DECENTRALIZED MULTI-AGENT SYSTEMS WITH SHARED CONTEXT

Anonymous authors
Paper under double-blind review

ABSTRACT

Multi-agent systems (MAS) can scale large language model agents on long-horizon tasks by running agents in parallel, but existing designs waste much of this parallelism in *bubbles*: agent time spent waiting on others or redoing a peer's work. Independent agents rediscover the same dead ends, peer-communicating agents wait at synchronous rounds, and centralized orchestration leaves the main agent blocked on its sub-agents while progress is relayed, and sometimes lost, through its context. We propose Decentralized Language Models (DELM), a MAS framework that squeezes out these bubbles by replacing the main agent with a shared context and a task queue. Agents asynchronously claim tasks, publish findings as they become available, and build on or correct one another's progress, with peer status visible to all. On long-horizon tasks from Terminal-Bench 4.0 and DeepSWE v1.1 and on SWE-bench Verified, DELM is more accurate and faster than Codex, Claude Code, and AOrchestra in every setting, gaining up to 17.5 points accuracy over the strongest baseline while running up to 2.49× faster than the underlying harness. Ablations show these gains come from coordination: removing any single mechanism brings two agents back to roughly single-agent speed.

1 INTRODUCTION

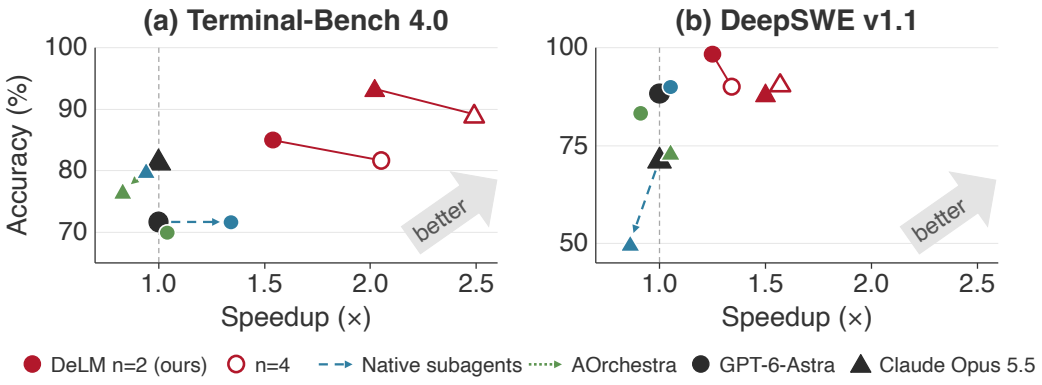

Figure 1: **DeLM is both faster and more accurate than existing multi-agent coding systems.** Averaged over the two models, DeLM (n=2) is 1.78× faster and 12.5 points more accurate than Codex/Claude Code (dashed line, 1×) on Terminal-Bench 4.0, and 1.38× faster and 13.3 points more accurate on DeepSWE v1.1. Each benchmark uses 10 long-horizon tasks (§3).

Multi-agent systems (MAS) offer a natural way to scale large language model reasoning at test time: instead of solving a complex task in a single trajectory, they decompose it into subtasks, dispatch agents in parallel, and aggregate their progress (Hong et al., 2023; Feng et al., 2026; Ruan et al., 2026; Zhang et al., 2025; OpenAI, 2025; Anthropic, 2026; Kimi et al., 2026). This paradigm is increasingly important in agentic coding, both in repository-level software engineering (e.g., SWE-bench (Jimenez et al., 2024), DeepSWE (Huang & Jiang, 2026)), where agents can investigate different files, root causes, or fixes concurrently, and in interactive terminal tasks (e.g., Terminal-Bench (Merrill et al.,

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.

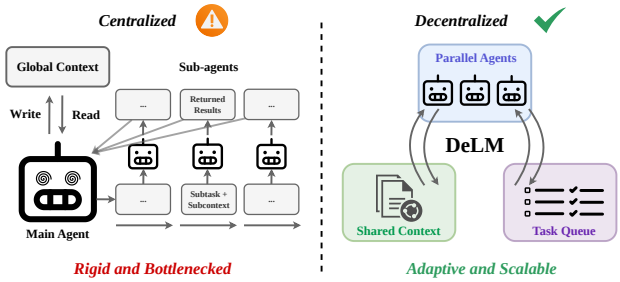

Figure 2: **Centralized vs. decentralized multi-agent systems.** Centralized MAS relies on a main agent to assign subcontexts, spawn sub-agents, and integrate their results, so progress is shared mainly through the main agent and sub-agents remain unaware of one another's work. In contrast, DELM decentralizes coordination through parallel agents, a shared context, and a task queue: agents asynchronously claim ready tasks, publish findings incrementally to a shared context, and build on or correct each other's progress without a main agent in the loop.

2026)), where agents can explore alternative solution paths while sharing what they learn about the environment. In both settings, the value of MAS lies not in issuing more model calls, but in converting additional test-time compute into useful parallel progress, reducing wall-clock time and improving success rates without a proportional increase in cost.

Realizing this potential requires keeping every agent productive. Borrowing a term from pipeline parallelism (Huang et al., 2019), we call agent time that does not advance the solution a *bubble*: time an agent spends waiting on others, or redoing work a peer has already done. Bubbles waste compute and stretch wall-clock time, and each of the three dominant families of MAS creates them in its own way (Table 1).

Independent agents run in parallel and select among their outputs, as in self-consistency (Wang et al., 2022), repeated sampling (Brown et al., 2024), and best-of-n sampling (Cobbe et al., 2021; Lightman et al., 2024). They never wait, but they share nothing while they run, so each agent must rediscover the same faults and dead ends on its own. Their bubbles are *redundant work*: adding agents adds compute, but much of it repeats exploration another agent has already done.

Peer-communicating agents exchange complete responses over synchronous rounds, as in multi-agent debate (Du et al., 2023; Liang et al., 2023; Chen et al., 2024). Sharing responses reduces redundancy, but each round is a barrier: agents that finish early must wait for the slowest one before the next round begins, the straggler problem familiar from bulk-synchronous parallel computing (Valiant, 1990). These *barrier waits* worsen as agents are added.

Centralized orchestration is used by structured search agents (Yao et al., 2023; Besta et al., 2024; Ning et al., 2024), SPRINT (Biju et al., 2026), Claude Code (Anthropic, 2026), Codex (OpenAI, 2025), AOrchestra (Ruan et al., 2026), and blackboard-style systems (Han & Zhang, 2025; Salemi et al., 2025). A main agent decomposes the task, dispatches subtasks, waits for the outputs, and integrates the results (Figure 2, left). This parallelizes sub-agent execution but not the coordination around it, creating *relay bubbles*: the main agent blocks while delegated work runs, and sub-agents, unaware of one another, cannot build on each other's progress until the main agent relays it. Figure 3 shows the effect: Claude Code's main agent spends 73% of its time waiting on its subagents. Relaying also loses information. Because progress lives in the main agent's context rather than in a persistent shared state, details can be dropped in summarization, and correcting a propagated error requires re-planning. In our experiments, enabling native subagents lowers Claude Code's accuracy on DeepSWE from 71.7% to 50.0% (Table 3).

We propose Decentralized Language Models (DELM), a MAS framework that squeezes out these bubbles by replacing the central controller with a shared problem state (Figure 2, right). DELM consists of parallel agents, a shared context, and a task queue. Agents claim tasks from the queue asynchronously, so no agent waits at a round barrier or for a main agent to assign work. They publish compact findings to the shared context as soon as results are available, so peers build on intermediate progress instead of waiting for a relay. The queue and shared context also expose what each agent is working on and which approaches have failed, so agents avoid redundant exploration, and corrections

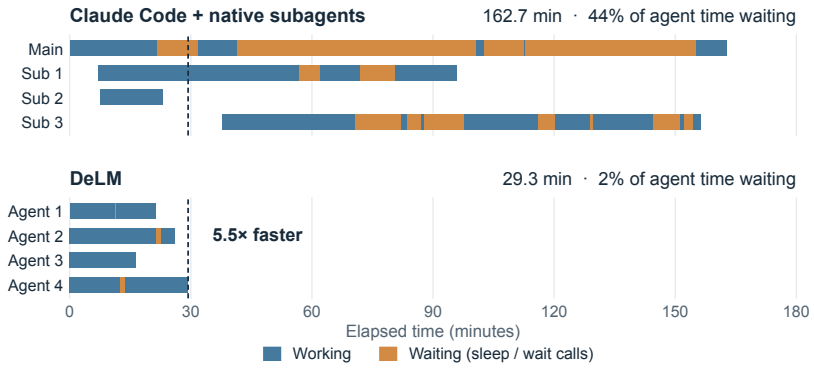

Figure 3: **Centralized orchestration creates bubbles; DELM squeezes them out.** Execution timelines on Terminal-Bench 4.0's `retro-console-soc` task with Claude Opus 5.5 and four agents. With Claude Code's centralized native subagents, the main agent spends 73% of its time waiting on delegated work (orange), and 44% of all agent time is spent waiting. DELM agents build on one another's intermediate results, spend 2% of their time waiting, and finish 5.5× sooner.

Table 1: Collaboration and coordination mechanisms in the compared multi-agent systems. A checkmark means the system explicitly provides the mechanism while a cross means that it does not.

System	General collaboration capabilities			Explicit coordination mechanisms		
	Parallel agent execution	Cross-agent work reuse	Feedback & correction	Intermediate-results sharing	Peer-status awareness	Stateful shared context
Isolated agents	✗	✗	✗	✗	✗	✗
AOrchestra	✗	✓	✓	✗	✗	✗
Codex	✓	✓	✓	✓	✗	✗
Claude Code	✓	✓	✓	✓	✗	✗
DELM	✓	✓	✓	✓	✓	✓

are appended as new entries, so errors are fixed without global re-planning. On the task in Figure 3, DELM agents spend only 2% of their time waiting and finish 5.5× sooner.

Key idea. Existing MAS leave agents idle or redundant: rediscovering one another's work, waiting at synchronous rounds, or waiting on a main agent. DELM squeezes out these bubbles by letting agents coordinate asynchronously through a shared context and task queue, with no main agent in the loop.

We evaluate DELM on three agentic coding benchmarks: Terminal-Bench 4.0 (Merrill et al., 2026), where agents share discovered environment state and failed attempts in live terminals; DeepSWE v1.1 (Huang & Jiang, 2026), where agents reuse each other's debugging progress on long-horizon software engineering tasks; and SWE-bench Verified (Jimenez et al., 2024), where agents explore alternative root causes and fixes for real GitHub issues. We compare against Codex and Claude Code, with and without native subagents, and AOrchestra (Ruan et al., 2026). Our key findings are:

- **DELM is both faster and more accurate on long-horizon coding.** Across two frontier models, DELM reaches 93.33% on Terminal-Bench 4.0 and 98.33% on DeepSWE v1.1, improving over the strongest baseline by up to 17.5 points while running up to 2.49× faster (Figure 1).
- **The gains hold with a lightweight model on a standard benchmark.** On SWE-bench Verified with Gemini 3 Flash, DELM reaches 66.08%, 10.8 points above the strongest baseline, while running 2.12× faster than Claude Code at half of AOrchestra's cost per submission (§3.3).
- **Trace analysis shows how DELM squeezes out bubbles.** Agents split implementation across components, build on and test one another's unfinished work, and step away from searches a peer has already claimed, while the shared context stays under 15K tokens (§4).

2 DECENTRALIZED LANGUAGE MODELS (DELM)

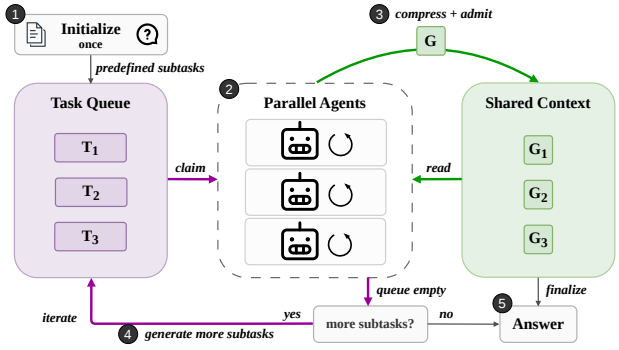

Figure 4: **Overview of DELM.** The shared task queue is initialized with a set of predefined, general preparation subtasks. Parallel agents asynchronously claim subtasks $\{T_i\}$, read the shared context, and perform local reasoning. Each agent's update is compressed and appended to the shared context as a gist G_i, making reusable progress visible to all agents. When the task queue is empty, the agent that most recently completed a subtask reviews the shared context to decide whether more subtasks are needed. If so, it generates new subtasks and enqueues them for another round of parallel execution; otherwise, each agent finalizes its own submission.

Given a task D, DELM runs n agents concurrently, each with its own persistent session and workspace. Agents exchange intermediate results as they work and independently produce their solutions.

Figure 4 and Algorithm 1 summarize five operations: ❶ initialize the task queue, ❷ execute subtasks in parallel, ❸ summarize and publish intermediate results, ❹ update the task queue as work progresses, and ❺ return each agent's solution when ready. These operations proceed asynchronously, allowing agents to reuse shared results while others continue working.

2.1 SHARED CONTEXT

The shared context $\mathcal{C}$ stores reusable findings and contributions from all agents. DELM lets agents share intermediate results through compact entries G_i that summarize findings, failed approaches, partial implementations, or corrections. Each entry records the scope of the result, its supporting evidence, and its known limitations, so other agents can judge whether it applies to their own work. Agents publish entries as soon as results are available and continue their own assignments.

An entry can reference files that are stored separately from its summary. Agents use `expand` to inspect the referenced files or their diffs against the original task files, and `apply` to copy selected files into their own workspaces. `apply` refuses to overwrite local changes it cannot account for and leaves reconciliation to the receiving agent. Entries remain available for later reuse, while subsequent reads emphasize new publications and changed work status. This design resembles demand paging in an operating system: agents follow progress through compact summaries and retrieve detailed content from backing storage only when needed. We provide more details in Appendix C.1.

Verification is distributed as well. Before reusing an entry, an agent checks its evidence against the original task, reuses applicable tests, and runs additional tests for interactions the entry does not cover. When an agent finds a defect in a shared result, it does not modify the original entry. Instead, it appends a new entry that describes the conditions under which the defect occurs and, when available, a correction, so other agents can update their own solutions.

2.2 TASK QUEUE

In task queue $\mathcal{T}$, each item specifies its intended output, the inputs available to it, and its dependencies on other items. The queue enforces exclusive claims and records whether each item is open, claimed, or complete, so every agent can see who is responsible for what and how much progress has been

made. Agents use this shared state to choose complementary work and to identify unresolved dependencies. Once execution begins, any agent can add new items to the queue or claim open ones as needed. The queue also exposes the session status of each agent, that is, whether the agent that claimed an item is still running. This status helps others decide whether to wait for a dependency, continue with independent work, or take over work that a terminated agent left unfinished.

3 EXPERIMENTS

Benchmarks. We evaluate DELM on three agentic coding benchmarks. (1) **Terminal-Bench 4.0** (Merrill et al., 2026). The benchmark consists of tasks that agents solve in live terminal environments, such as configuring systems, installing dependencies, and debugging failures. The tasks require long sequences of shell commands and agents benefit from sharing discovered environment state and failed attempts. (2) **DeepSWE v1.1** (Huang & Jiang, 2026). The benchmark contains long-horizon software-engineering tasks. Agents must localize faults, understand interdependent modules, and debug across large codebases. It tests whether agents can reuse each other's localization and debugging progress instead of repeating it. (3) **SWE-bench Verified** (Jimenez et al., 2024). The benchmark is built from real-world GitHub issues, with human-verified task specifications and tests. Agents inspect the repository, identify the relevant files, implement a fix, and validate it through tests.

Because DELM targets long-horizon tasks where coordination matters, we evaluate on a subset of long-running tasks from Terminal-Bench 4.0 and DeepSWE v1.1. We select tasks based on the wall-clock latency of Codex with GPT-6-Astra (`xhigh`): for Terminal-Bench 4.0, we select 10 tasks with latency between 15 minutes and 1 hour; for DeepSWE v1.1, we select 10 tasks with latency of at least 10 minutes. The full list of selected tasks is provided in Appendix D.

Baselines. We compare DELM with production coding agent harnesses, both in their default mode and with their native subagents enabled, as well as with a research centralized multi-agent system. **Codex** (v0.153.4) (OpenAI, 2025) and **Claude Code** (v2.1.280) (Anthropic, 2026) are agentic coding harnesses that enable language models to work on coding tasks using tools for file editing and shell execution. **Codex (subagent)** and **Claude Code (subagent)** use the same harnesses, with one addition: the task prompt ends with the instruction "Use [Codex/Claude Code]'s native subagents to help solve this task." This lets the main agent spawn and coordinate sub-agents through each harness's built-in mechanism. **AOrchestra** (Ruan et al., 2026) is a centralized multi-agent system. Its main orchestrator creates specialized sub-agents on demand and instantiates each one as a tuple of instruction, context, tools, and model. On SWE-bench Verified, we also compare with **mini-SWE-agent** (Yang et al., 2024), a lightweight software-engineering agent that uses a simple linear loop with Bash as its main interface.

Models and implementation. On Terminal-Bench 4.0 and DeepSWE v1.1, we use two frontier base models: GPT-6-Astra (OpenAI, 2026b) and Claude Opus 5.5 (Anthropic, 2026b), both with `xhigh` reasoning effort. We build DELM on top of Codex for GPT-6-Astra and on top of Claude Code for Claude Opus 5.5. Every baseline uses the same base model as DELM in its setting, so the only difference between them is how agents coordinate. On SWE-bench Verified, all methods use Gemini 3 Flash (Doshi, 2025), accessed through OpenRouter with default reasoning and sampling parameters. For this setting, we route the Claude Code CLI through OpenRouter to support the non-Anthropic model. For every task, we evaluate DELM with $n \in \{2, 4\}$ parallel agents.

Metrics. We report task accuracy, wall-clock latency, and cost. Wall-clock latency is the time from the first API call to the last generated token. We use two cost measures: Cost/Task is the total API cost of all agents in a single run, and Cost/Submission divides this total by the number of agents, since each DELM agent produces its own submission that is scored independently. All results are averaged over three runs and reported as mean $\pm$ standard deviation.

3.1 TERMINALBENCH-4.0

We evaluate on the 10 long-horizon Terminal-Bench 4.0 tasks described in Appendix D. Table 2 shows that DELM improves both accuracy and latency under both base models. With GPT-6-Astra, DELM ($n = 2$) reaches 85.00% accuracy, 13.3 points higher than the strongest baseline (71.67%),

Table 2: Comparison on TerminalBench-4.0 across two base models. Best mean values within each model are bolded. Cost/Task sums the costs of all agents in a single run. Cost/Submission divides this total by the number of agents, as each agent produces one submission that is scored independently.

Method	Avg. Acc (%)	Avg. Latency	Cost/Task	Cost/Submission
GPT-6-Astra				
Codex	71.67 ± 11.55	26.91 ± 0.91 min ($1.00\times$)	**\$9.01** $\pm$ \$0.71	\$9.01 $\pm$ \$0.71
Codex (subagent)	71.67 ± 11.69	20.04 ± 1.94 min ($1.34\times$)	\$25.80 $\pm$ \$2.21	\$25.80 $\pm$ \$2.21
AOrchestra	70.00 ± 5.00	25.78 ± 1.15 min ($1.04\times$)	\$11.94 $\pm$ \$1.02	\$11.94 $\pm$ \$1.02
DELM ($n = 2$)	$\mathbf{85.00} \pm 13.23$	17.47 ± 1.82 min ($1.54\times$)	\$13.65 $\pm$ \$1.49	\$6.83 $\pm$ \$0.74
DELM ($n = 4$)	81.67 ± 12.33	$\mathbf{13.16} \pm 1.43$ min ($\mathbf{2.05}\times$)	\$23.90 $\pm$ \$2.27	**\$5.97** $\pm$ \$0.57
Claude Opus 5.5				
Claude Code	81.67 ± 2.89	130.75 ± 6.27 min ($1.00\times$)	\$20.31 $\pm$ \$0.23	\$20.31 $\pm$ \$0.23
Claude Code (subagent)	80.00 ± 5.00	138.58 ± 10.17 min ($0.94\times$)	\$30.88 $\pm$ \$3.17	\$30.88 $\pm$ \$3.17
AOrchestra	76.67 ± 5.77	157.21 ± 9.03 min ($0.83\times$)	\$24.22 $\pm$ \$1.05	\$24.22 $\pm$ \$1.05
DELM ($n = 2$)	$\mathbf{93.33} \pm 3.33$	64.77 ± 5.03 min ($2.02\times$)	**\$17.91** $\pm$ \$1.17	\$8.96 $\pm$ \$0.59
DELM ($n = 4$)	89.17 ± 5.20	$\mathbf{52.51} \pm 4.43$ min ($\mathbf{2.49}\times$)	\$29.98 $\pm$ \$0.91	**\$7.50** $\pm$ \$0.23

and runs $1.54\times$ faster than Codex. With Claude Opus 5.5, DELM ($n = 2$) reaches 93.33%, 11.7 points higher than Claude Code, while cutting latency from 130.75 to 64.77 minutes ($2.02\times$) at a lower Cost/Task (\$17.91 vs. \$20.31). Increasing to $n = 4$ gives the largest speedups ($2.05\times$ and $2.49\times$) and the lowest Cost/Submission, at the price of a few points of accuracy relative to $n = 2$.

The native subagent baselines show that the benefits of centralized coordination are limited and inconsistent. Codex (subagent) matches Codex in accuracy and runs $1.34\times$ faster, but at $2.9\times$ the cost per task, while Claude Code (subagent) is both less accurate and slower than Claude Code. In contrast, DELM ($n = 2$) outperforms both subagent variants in accuracy (by 13.3 points) and latency, while reducing Cost/Task by 42–47%. We attribute this gap to how progress flows between agents: in the subagent baselines, findings must pass through the main agent, whereas in DELM, environment state and failed attempts recorded in the shared context are immediately available to every agent.

3.2 DEEPSWE V1.1

We evaluate on the 10 long-horizon DeepSWE v1.1 tasks described in Appendix D. Table 3 shows that DELM again achieves the best accuracy and latency under both base models. With GPT-6-Astra, DELM ($n = 2$) reaches 98.33% accuracy, 8.3 points higher than the strongest baseline, Codex (subagent) (90.00%), and is $1.25\times$ faster than Codex. It also costs less per task than Codex (subagent) (\$16.83 vs. \$24.97). With Claude Opus 5.5, the gains are larger: DELM ($n = 4$) reaches 90.83%, 17.5 points higher than the strongest baseline, AOrchestra (73.33%), and runs $1.57\times$ faster than Claude Code. DELM ($n = 2$) achieves 88.33% with the lowest Cost/Task of all methods (\$11.89).

The largest contrast comes from Claude Code (subagent), whose accuracy drops from 71.67% to 50.00% compared with Claude Code alone, while its latency and cost both increase. Delegating to native subagents can hurt performance because relaying summarized findings through the main agent may lose useful details and create an integration bottleneck.

3.3 SWE-BENCH VERIFIED

SWE-bench Verified tests whether DELM still helps on a widely used software-engineering benchmark with a lightweight base model. We build DELM based on AOrchestra's harness and compare it with Claude Code, mini-SWE-agent, and AOrchestra, all using Gemini 3 Flash. Table 4 shows that DELM outperforms all baselines in both accuracy and latency. With $n = 4$, DELM reaches 66.08% accuracy, 10.8 points higher than the strongest baseline, AOrchestra (55.26%). It is also $2.12\times$ faster than Claude Code and $1.47\times$ faster than AOrchestra. With $n = 2$, DELM still improves accuracy by 8.5 points over AOrchestra at the same Cost/Task (\$0.24), which halves the cost per submission. Going from $n = 2$ to $n = 4$ roughly doubles Cost/Task but leaves Cost/Submission unchanged at \$0.12, and it further improves both accuracy and latency.

Table 3: Comparison on DeepSWE v1.1 across two base models. Best mean values within each model are bolded. Cost/Task sums the costs of all agents in a single run. Cost/Submission divides this total by the number of agents, as each agent produces one submission that is scored independently.

Method	Avg. Acc (%)	Avg. Latency	Cost/Task	Cost/Submission
GPT-6-Astra				
Codex	88.33 ± 2.89	16.57 ± 0.39 min (1.00×)	**$9.40** ± $0.18	$9.40 ± $0.18
Codex (subagent)	90.00 ± 0.00	15.76 ± 2.05 min (1.05×)	$24.97 ± $3.15	$24.97 ± $3.15
AOrchestra	83.33 ± 7.64	18.31 ± 1.08 min (0.91×)	$10.74 ± $0.97	$10.74 ± $0.97
DELM ($n = 2$)	**98.33** ± 2.89	13.29 ± 0.98 min (1.25×)	$16.83 ± $1.65	$8.42 ± $0.82
DELM ($n = 4$)	90.00 ± 10.00	**12.37** ± 0.66 min (**1.34**×)	$30.87 ± $1.09	**$7.72** ± $0.27
Claude Opus 5.5				
Claude Code	71.67 ± 2.89	49.58 ± 3.11 min (1.00×)	$12.01 ± $0.60	$12.01 ± $0.60
Claude Code (subagent)	50.00 ± 10.00	57.46 ± 2.16 min (0.86×)	$19.83 ± $0.99	$19.83 ± $0.99
AOrchestra	73.33 ± 5.77	47.32 ± 2.86 min (1.05×)	$12.11 ± $0.52	$12.11 ± $0.52
DELM ($n = 2$)	88.33 ± 2.89	32.98 ± 2.47 min (1.50×)	**$11.89** ± $0.53	**$5.95** ± $0.26
DELM ($n = 4$)	**90.83** ± 8.78	**31.61** ± 0.96 min (**1.57**×)	$24.49 ± $0.67	$6.12 ± $0.17

These results show that multiple agents help even on largely sequential tasks. A single agent rarely completes every step correctly on its first attempt. It often inspects irrelevant files, pursues a wrong root cause, or submits a patch that fails the tests before it recovers. Each such detour costs time, and a single agent can only discover it by paying that cost itself. In DELM, agents explore different paths in parallel and record what they learn in the shared context. Once one agent has hit a dead end, the others can skip it, and once one has localized a fault, the others can build on that finding. This shortens the path to a correct fix, which explains why DELM is both faster and more accurate. Appendix F provides a detailed analysis of why DELM is both accurate and cost-efficient on SWE-bench.

Table 4: Comparison on SWE-bench Verified with Gemini 3 Flash. Best mean values within each model are bolded. Cost/Task sums the costs of all agents in a single run. Cost/Submission divides this total by the number of agents, as each agent produces one submission that is scored independently.

Method	Avg. Acc (%)	Avg. Latency	Cost/Task	Cost/Submission
Gemini 3 Flash				
Claude Code	49.32 ± 1.98	129.13 ± 5.12 s (1.00×)	$1.00 ± $0.00	$1.00 ± $0.00
mini-SWE-agent	54.73 ± 2.37	91.78± 3.53 s (1.41×)	$0.26 ± $0.07	$0.26 ± $0.07
AOrchestra	55.26 ± 2.03	89.94 ± 3.07 s (1.44×)	**$0.24** ± $0.04	$0.24 ± $0.04
DELM ($n = 2$)	63.72 ± 2.29	71.86 ± 2.88 s (1.80×)	**$0.24** ± $0.05	**$0.12** ± $0.03
DELM ($n = 4$)	**66.08** ± 1.86	**61.01** ± 2.63 s (**2.12**×)	$0.47 ± $0.07	**$0.12** ± $0.02

[a] The Claude Code CLI sends `cache_control` blocks in the Anthropic API format, so cache reuse only works if the upstream provider honors those blocks. For Gemini-3-Flash the real cost without cache reuse is therefore around $1 per task.

3.4 Ablation Study

To isolate the contribution of each coordination mechanism in Table 1, we remove one mechanism at a time from DELM ($n = 2$) on Terminal-Bench 4.0 and keep everything else fixed. **w/o early sharing** lets agents publish results only after they complete their claimed subtask. **w/o peer-status** hides item owners and session status, so agents cannot see what their peers are currently working on. **w/o shared context** removes the shared context entirely, and agents coordinate by sending messages directly to each other. Figure 5 shows the results.

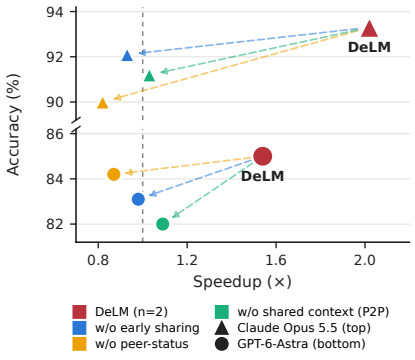

Figure 5: Ablation on Terminal-Bench 4.0.

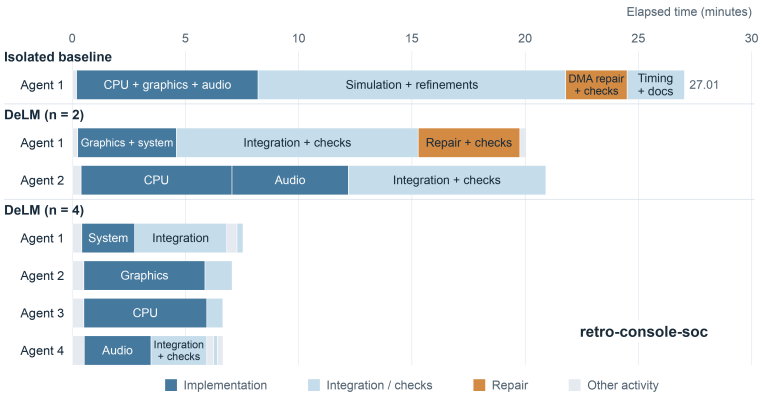

Figure 6: **More agents develop more components in parallel.** On Terminal-Bench 4.0's `retro-console-soc` task, one agent develops the CPU and audio components sequentially when n=2. With n=4, separate agents develop the CPU, graphics, and audio concurrently.

The most striking effect is on latency. With the full system, the two agents run $1.54\times$ (GPT-6-Astra) and $2.02\times$ (Claude Opus 5.5) faster than the single-agent harness, whereas every ablation falls to 0.82–$1.09\times$. Two parallel agents are thus no faster than one unless they coordinate effectively, so the speedup of DELM comes from coordination rather than from parallelism alone. Removing peer-status awareness causes the largest slowdown under both models ($0.87\times$ and $0.82\times$), and with Claude Opus 5.5 it also causes the largest accuracy drop (-3.3 points). Without knowing what their peers are working on, agents tend to claim overlapping work and discover the duplication only after both have spent time on it. Removing early sharing brings the speedup back to about $1\times$ ($0.98\times$ and $0.93\times$), because an agent whose work depends on a peer's output must wait until the peer finishes its entire subtask instead of starting from partial results. Replacing the shared context with direct messaging retains slightly more speed than the other ablations ($1.09\times$ and $1.03\times$) but causes the largest accuracy drop with GPT-6-Astra (-3.0 points). Direct messages still let agents exchange information, but they do not accumulate into a persistent record that agents can revisit, build on, or correct, so findings and fixes are more easily missed.

4 Trace Analysis: How Does Decentralized Coordination Help?

To explain why the coordination mechanisms in Table 1 improve performance, we analyze all DELM runs on Terminal-Bench 4.0 and DeepSWE v1.1 (240 runs, 720 agent trajectories). Parallel execution lets agents build components concurrently (§4.1), intermediate-results sharing lets them start dependent work before a peer finishes (§4.2), peer-status awareness keeps them from duplicating a peer's search (§4.3), and a stateful shared context lets corrections reach every agent without re-planning (§4.4).

4.1 Parallel agent execution speeds up implementation

Parallel agents speed up a task only if the work is divided so that none of them idles or duplicates another. DELM agents claim distinct components from the task queue, share code through the shared context, and each integrates its peers' components into its own solution. Adding agents makes the division finer, shortening the critical path. On `retro-console-soc` (Figure 6), one agent builds the CPU and then the audio component when n=2, whereas with n=4, separate agents build the CPU, graphics, and audio concurrently. This is consistent with the further speedup from n=2 to n=4 in §3.1 ($2.02\times$ to $2.49\times$ with Claude Opus 5.5 on Terminal-Bench 4.0).

4.2 Intermediate-results sharing unblocks dependent work

When one component depends on another, centralized orchestration makes the consumer wait until the producer finishes and the main agent relays the result. DELM agents instead publish partial results as soon as they run, so dependent work and integration testing overlap with implementation.

In Terminal-Bench's `telecom-entity-resolution` task, Agent 2's household-separation step consumes the clusters produced by Agent 1's record matcher. Agent 1 shared the matcher as soon as it ran, before its thresholds were tuned, so Agent 2 developed its step on real clusters while the matcher was still being refined.

4.3 PEER-STATUS AWARENESS AVOIDS DUPLICATE EXPLORATION

Parallel agents that cannot see each other's work tend to converge on the same promising direction. In centralized systems, each sub-agent sees only the subcontext the main agent assigned it, so overlapping work goes unnoticed until the main agent collects the results. DELM exposes every agent's claims and status through the task queue, so an agent can notice an overlap while its search is still running and redirect.

In Terminal-Bench's `wdm-design` task, agents optimize an optical device by changing its port geometry and material pattern. In one four-agent run, Agents 1 and 3 independently began tuning the same port widths and positions. Agent 1 then saw Agent 3's claim through the shared context:

```
[agent-3/FACT] [...] I will tune only w_in,w_long,w_short,y_long,
    y_short [...]
[agent-1/FACT] [...] applying the cavity artifact revealed agent-3 had
    just claimed the same five-parameter port tuning. [...]
[agent-1/EXEC] kill -TERM 770
[agent-1/EXEC] python binary_refine.py [...]
```

Agent 1 terminates its own port search and returns to refining the material pattern of its earlier candidate, while Agent 3 continues port tuning. The agents resolve the overlap themselves, before either search finishes, without a main agent having to detect the duplication and re-plan.

4.4 STATEFUL SHARED CONTEXT ENABLES REUSE AND CORRECTION

Shared results help only if later improvements reach the agents that already used them. Because corrections are appended as new entries that reference revised files rather than overwriting old ones (§2.1), an agent can re-import a peer's updated work at any time. Across 720 agent trajectories, agents imported shared files 5,237 times, and 80.3% of trajectories later imported a revised version of a file they had already imported. Agents thus track their peers' work as it improves, and corrections propagate without re-planning, whereas in centralized systems a correction to a summarized result requires another round of delegation.

This reuse is inexpensive. Because agents publish short summaries and attach files by reference (§2.1), the shared context averages under 15K tokens at task completion (Table 5). Reading it, including repeated reads by all agents, accounts for 1.7–3.7% of total task cost with Claude Opus 5.5 and 7.7–17.7% with GPT-6-Astra. Since the shared context only grows by appending, it forms a stable prompt prefix that prefix caching makes cheaper to reread (Appendix C.2).

5 DISCUSSION AND LIMITATIONS

Conclusion. We introduced DELM, a decentralized MAS framework in which agents coordinate through a shared context and a task queue rather than through a main agent, squeezing out the bubbles that centralized orchestration creates. Across Terminal-Bench 4.0, DeepSWE v1.1, and SWE-bench Verified, DELM is both more accurate and faster than single-agent harnesses, their native subagents, and AOrchestra, and our ablation shows that these gains come from coordination: removing any single mechanism brings two agents back to roughly single-agent speed. These results suggest that scaling multi-agent systems requires not only more agents, but a coordination substrate that keeps every agent productive.

Limitations. In certain settings, lower latency and Cost/Submission for DELM come with higher Cost/Task. A natural-language shared context is also less suited to exact aggregation over structured data, where DELM can be enhanced when combined with code-mediated execution (Appendix E.1).

6 AI USE STATEMENT

In this work, we used generative AI tools for coding assistance and for writing polish. Specifically, LLM-based coding assistants were used to help implement and debug parts of the pipeline. They were also used to improve the grammar and clarity of the manuscript. We have not used generative AI tools for research ideation or experimental design. We have reviewed all AI-assisted work. All AI-assisted code was read line by line and was verified by the authors. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of AI.

7 ETHICS STATEMENT

This work studies an efficiency-oriented multi-agent framework for language models. All experiments use commercially available models (GPT-6-Astra and Claude Opus 5.5) and existing benchmarks (Terminal-Bench and DeepSWE), used in accordance with their respective terms, licenses, and intended research use. No human subjects were involved, and no new human-subject data was collected or annotated. We rely on the original benchmark authors' documentation regarding personally identifiable or offensive content.

8 REPRODUCIBILITY STATEMENT

We have aimed to make all components of DELM reproducible from the paper alone. Section 2 and Algorithm 1 describe the shared context and task queue, and Appendix C.1 specifies how agents inspect and import shared files. Appendix G gives the agent prompts, Section 3 the experimental configuration, and Appendix D the selected tasks.

REFERENCES

Anthropic. Claude Sonnet 4.6. https://www.anthropic.com/news/claude-sonnet-4-6, 2025. Accessed: 2026-05-06.

Anthropic. Introducing Claude Opus 4.6. Anthropic, February 2026a. URL https://www.anthropic.com/news/claude-opus-4-6. Accessed: 2026-06-01.

Anthropic. Introducing Claude Opus 5.5. https://www.anthropic.com/claude-opus-5-5, September 2026b. Accessed: 2026-09-25.

Anthropic. Claude code. https://claude.ai/, 2026.

Yushi Bai, Shangqing Tu, Jiajie Zhang, Hao Peng, Xiaozhi Wang, Xin Lv, Shulin Cao, Jiazheng Xu, Lei Hou, Yuxiao Dong, et al. Longbench v2: Towards deeper understanding and reasoning on realistic long-context multitasks. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 3639–3664, 2025.

Amanda Bertsch, Adithya Pratapa, Teruko Mitamura, Graham Neubig, and Matthew R Gormley. Oolong: Evaluating long context reasoning and aggregation capabilities. *arXiv preprint arXiv:2511.02817*, 2025.

Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, et al. Graph of thoughts: Solving elaborate problems with large language models. In *Proceedings of the AAAI conference on artificial intelligence*, volume 38, pp. 17682–17690, 2024.

Emil Biju, Shayan Talaei, Zhemin Huang, Mohammadreza Pourreza, Azalia Mirhoseini, and Amin Saberi. Sprint: Enabling interleaved planning and parallelized execution in reasoning models. *Advances in Neural Information Processing Systems*, 38:96393–96433, 2026.

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

Justin Chen, Swarnadeep Saha, and Mohit Bansal. Reconcile: Round-table conference improves reasoning via consensus among diverse llms. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7066–7085, 2024.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

DeepSeek-AI. Deepseek-v4: Towards highly efficient million-token context intelligence, 2026.

Tulsee Doshi. Gemini 3 Flash: Frontier Intelligence Built for Speed. Google — The Keyword, December 2025. URL https://blog.google/products-and-platforms/products/gemini/gemini-3-flash/. Accessed: 2026-06-01.

Yilun Du, Shuang Li, Antonio Torralba, Joshua B Tenenbaum, and Igor Mordatch. Improving factuality and reasoning in language models through multiagent debate. *arXiv preprint arXiv:2305.14325*, 2023.

Lee D. Erman, Frederick Hayes-Roth, Victor R. Lesser, and D. Raj Reddy. The Hearsay-II speech-understanding system: Integrating knowledge to resolve uncertainty. *ACM Computing Surveys*, 12 (2):213–253, 1980. doi: 10.1145/356810.356816.

Zhaopeng Feng, Liangcai Su, Zhen Zhang, Xinyu Wang, Xiaotian Zhang, Xiaobin Wang, Runnan Fang, Qi Zhang, Baixuan Li, Shihao Cai, et al. Agentswing: Adaptive parallel context management routing for long-horizon web agents. *arXiv preprint arXiv:2603.27490*, 2026.

Bochen Han and Songmao Zhang. Exploring advanced llm multi-agent systems based on blackboard architecture. *arXiv preprint arXiv:2507.01701*, 2025.

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

Wenqi Huang and Peter Jiang. Deepswe v1.1: a cleaner, more reproducible benchmark for frontier coding agents, 2026. URL https://github.com/datacurve-ai/deep-swe.

Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Kimi, Tongtong Bai, Yifan Bai, Yiping Bao, SH Cai, Yuan Cao, Y Charles, HS Che, Cheng Chen, Guanduo Chen, et al. Kimi k2. 5: Visual agentic intelligence. *arXiv preprint arXiv:2602.02276*, 2026.

Kuang-Huei Lee, Xinyun Chen, Hiroki Furuta, John Canny, and Ian Fischer. A human-inspired reading agent with gist memory of very long contexts. *arXiv preprint arXiv:2402.09727*, 2024.

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.

Hanchen Li, Runyuan He, Qizheng Zhang, Changxiu Ji, Qiuyang Mang, Xiaokun Chen, Lakshya A Agrawal, Wei-Liang Liao, Eric Yang, Alvin Cheung, et al. Combee: Scaling prompt learning for self-improving language model agents. *arXiv preprint arXiv:2604.04247*, 2026.

Tian Liang, Zhiwei He, Wenxiang Jiao, Xing Wang, Yan Wang, Rui Wang, Yujiu Yang, Zhaopeng Tu, and Shuming Shi. Encouraging divergent thinking in large language models through multi-agent debate. arxiv 2023. *arXiv preprint arXiv:2305.19118*, 10, 2023.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *International Conference on Learning Representations*, volume 2024, pp. 39578–39601, 2024.

Monishwaran Maheswaran, Leon Lakhani, Zhongzhu Zhou, Shijia Yang, Junxiong Wang, Coleman Hooper, Yuezhou Hu, Rishabh Tiwari, Jue Wang, Harman Singh, et al. Squeeze evolve: Unified multi-model orchestration for verifier-free evolution. *arXiv preprint arXiv:2604.07725*, 2026.

Mike Merrill, Alexander Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Shin, Thomas Walshe, E Kelly Buchanan, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *International Conference on Learning Representations*, volume 2026, pp. 40903–40986, 2026.

H Penny Nii. Part one: The blackboard model of problem solving and the evolution of blackboard architectures. *AI magazine*, 7(2):38–53, 1986.

Xuefei Ning, Zinan Lin, Zixuan Zhou, Zifu Wang, Huazhong Yang, and Yu Wang. Skeleton-of-thought: Prompting llms for efficient parallel generation. In *International Conference on Learning Representations*, volume 2024, pp. 917–967, 2024.

OpenAI. Codex. https://openai.com/codex/, 2025.

OpenAI. Introducing GPT-5.4. https://openai.com/index/introducing-gpt-5-4/, 2026a. Accessed: 2026-05-06.

OpenAI. GPT-6 Astra: A new generation of intelligence. https://openai.com/index/gpt-6-astra/, September 2026b. Accessed: 2026-09-25.

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

Jianhao Ruan, Zhihao Xu, Yiran Peng, Fashen Ren, Zhaoyang Yu, Xinbing Liang, Jinyu Xiang, Yongru Chen, Bang Liu, Chenglin Wu, et al. Aorchestra: Automating sub-agent creation for agentic orchestration. *arXiv preprint arXiv:2602.03786*, 2026.

Alireza Salemi, Mihir Parmar, Palash Goyal, Yiwen Song, Jinsung Yoon, Hamed Zamani, Tomas Pfister, and Hamid Palangi. Llm-based multi-agent blackboard system for information discovery in data science. *arXiv preprint arXiv:2510.01285*, 2025.

Hongjin Su, Shizhe Diao, Ximing Lu, Mingjie Liu, Jiacheng Xu, Xin Dong, Yonggan Fu, Peter Belcak, Hanrong Ye, Hongxu Yin, et al. Toolorchestra: Elevating intelligence via efficient model and tool orchestration. *arXiv preprint arXiv:2511.21689*, 2025.

Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8): 103–111, 1990.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://arxiv.org/abs/2405.15793.

Xiyuan Yang, Jiaru Zou, Rui Pan, Ruizhong Qiu, Pan Lu, Shizhe Diao, Jindong Jiang, Hanghang Tong, Tong Zhang, Markus J Buehler, et al. Recursive multi-agent systems. *arXiv preprint arXiv:2604.25917*, 2026.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems*, 36:11809–11822, 2023.

Alex L Zhang, Tim Kraska, and Omar Khattab. Recursive language models. *arXiv preprint arXiv:2512.24601*, 2025.

A Related Work

Multi-agent coordination. Recent LLM-based multi-agent systems improve performance by decomposing tasks across specialized or interacting agents. CAMEL (Li et al., 2023) instantiates role-playing agents to study autonomous cooperation, while MetaGPT (Hong et al., 2023) and ChatDev (Qian et al., 2024) organize collaboration through software-engineering roles and standardized workflows. More recent systems make orchestration adaptive. AOrchestra (Ruan et al., 2026) creates task-specific sub-agents on demand, ToolOrchestra (Su et al., 2025) trains a lightweight orchestrator to coordinate models and tools, Squeeze Evolve (Maheswaran et al., 2026) routes the stages of evolutionary inference to models of different cost and capability, and RecursiveMAS (Yang et al., 2026) replaces text-level communication with recursive latent-state transfer. These systems coordinate through either a fixed workflow or a central orchestrator that decides what each agent sees and when progress is merged. Combee (Li et al., 2026) likewise identifies centralized aggregation as a bottleneck under high parallelism, but addresses it for parallel prompt learning rather than agent coordination.

Blackboard and shared-memory architectures. Blackboard architectures coordinate problem solvers through a shared workspace (Erman et al., 1980; Nii, 1986). Recent LLM variants use a blackboard on which agents post messages for general reasoning (Han & Zhang, 2025) or to which agents volunteer information in response to requests for data discovery (Salemi et al., 2025). However, these systems still rely on a control agent to select which agent acts or which request is posted, and they treat the board mainly as a message channel. DELM removes the controller from the loop entirely. Agents claim work asynchronously from a task queue, see what their peers are working on, and build on or correct earlier findings by appending to a stateful shared context (Table 1). We further target long-horizon agentic coding, where shared progress includes repository and environment state rather than only textual messages.

Agentic coding harnesses. Language models become far more capable when coupled with executable environments that let them inspect state, invoke tools, and act on feedback. SWE-agent (Yang et al., 2024) and its lightweight variant mini-SWE-agent show that a well-designed agent-computer interface substantially improves software-engineering performance. Production harnesses such as Claude Code (Anthropic, 2026) and Codex (OpenAI, 2025) let models read repositories, edit files, execute commands, and increasingly delegate work to native subagents. Recursive Language Models (Zhang et al., 2025) apply a similar code-mediated loop to long-context reasoning by treating the prompt as an external environment. DELM complements these harnesses: rather than replacing the single-agent loop, it adds a coordination layer on top of it, and we instantiate it on both Codex and Claude Code.

B Context Size and Cost Analysis

Table 5: Final shared-context size (thousands of tokens) and the cost of reading it as a percentage of total task cost.

		Terminal-Bench 4.0		DeepSWE v1.1	
Model	**Agents**	**Context (k)**	**Cost (%)**	**Context (k)**	**Cost (%)**
GPT-6-Astra	$n = 2$	7.31 ± 0.42	9.17 ± 0.38	7.54 ± 0.21	7.71 ± 0.18
	$n = 4$	12.88 ± 0.97	17.68 ± 0.25	13.03 ± 0.96	14.56 ± 0.37
Claude Opus 5.5	$n = 2$	6.62 ± 0.34	1.92 ± 0.11	4.67 ± 0.11	1.69 ± 0.06
	$n = 4$	14.48 ± 0.47	3.66 ± 0.29	10.43 ± 0.53	3.04 ± 0.23

Values are mean $\pm$ sample standard deviation across three repeats, with tasks averaged within each repeat. Board cost estimates the cost of reading shared context across all agents, including cached reuse.

Algorithm 1 DELM pipeline. Agents coordinate concurrent work through a shared context $\mathcal{C}$ and task queue $\mathcal{T}$. They publish intermediate results for others to reuse, update the queue as needed, and return their solutions independently.

Input: Task D; number of agents n
Output: Agent solutions $\{Y_i\}_{i=1}^n$

$\mathcal{C} \leftarrow \emptyset$; $\mathcal{T} \leftarrow \{T_i\}_{i=1}^n$ ❶
for all agents $i = 1, \ldots, n$ **in parallel do**
  $r_i \leftarrow \emptyset$ ▷ *accumulated local work*
  repeat
    $r_i \leftarrow \textsc{ContinueWork}(D, r_i, \mathcal{T}, \mathcal{C})$ ❷
    if useful intermediate results or corrections are available **then**
      $G_i \leftarrow \textsc{Summarize}(r_i)$ ❸
      $\textsc{Publish}(\mathcal{C}, G_i)$ ▷ *references to shared files*
    end if
    $\textsc{UpdateSubtasks}(\mathcal{T}, r_i, \mathcal{C})$ ▷ *complete or add work* ❹
  until $\textsc{ReadyToSubmit}(D, r_i, \mathcal{C})$
  $Y_i \leftarrow \textsc{Finalize}(D, r_i, \mathcal{C})$ ❺
end for
return $\{Y_i\}_{i=1}^n$

C Method Details

C.1 Selective Retrieval and Reuse

A compact entry G_i in the shared context $\mathcal{C}$ can reference files captured at publication time. These files remain available under the publication's identifier, even as the contributing agent continues editing its own workspace. Agents access them through two operations, `expand` and `apply`, which separate inspection from incorporation into a local solution.

Expansion (`expand`). The `expand` operation returns a publication's attached files as tool output without changing the agent's workspace. By default, it shows differences against the original task files, with full contents for new files or files whose originals are unavailable. Using `-only`, the agent can select particular files and view them in full, allowing it to examine the implementation details or supporting evidence relevant to its current work.

Application (`apply`). The `apply` operation copies all or selected attachments into the receiving agent's workspace. It transfers complete files rather than merging patches, checking local contents against the task originals and published versions before writing. If a selected file contains local changes it cannot account for, the import is refused and the agent must reconcile those changes. Expansion is optional: an agent can inspect a contribution before incorporating it or apply selected files directly.

Expanded content enters the requesting agent's session, while applied files become part of its local workspace. Neither operation inserts file contents into the compact shared-context view, so agents can inspect and reuse detailed contributions without requiring every other agent to read them.

C.2 KV-cache reuse from a stable shared context

The shared-context design also improves KV-cache reuse. Because the shared context is largely stable and accumulative, it forms a persistent prefix across calls. This is useful in multi-step reasoning, where many LLM calls share the same global state but differ only in the task-specific suffix. As a result, repeated computation over the shared prefix can be amortized, yielding larger efficiency gains as the number of steps grows.

D SELECTED TASKS

DELM is designed for long-horizon tasks, where multiple agents have enough time to share progress and split up the remaining work. On short tasks, a single agent often finishes before any coordination can pay off. We therefore evaluate on a subset of long-running tasks from Terminal-Bench 4.0 and DeepSWE v1.1. We first ran the Codex baseline with GPT-6-Astra (`xhigh`) on each benchmark and recorded the wall-clock latency of every task. From Terminal-Bench 4.0, we selected 10 tasks whose latency fell between 15 minutes and 1 hour. The upper bound excludes tasks that run so long that repeated evaluation would be impractical. From DeepSWE v1.1, we selected 10 tasks with latency of at least 10 minutes. The selection criterion is based only on the baseline's latency. Table 6 lists the selected tasks together with their Codex latencies.

Terminal-Bench 4.0

Task	Codex Latency
ks-solver-cpp	21.00 min
telecom-entity-resolution	24.23 min
vf2-speedup-networkx	23.42 min
biped-contact-dynamics	21.56 min
cumulative-layout-shift	39.34 min
retro-console-soc	27.28 min
rs-archive-clone	21.92 min
wdm-design	39.27 min
lake-temp-glm	32.40 min
payments-pipeline-fix	18.67 min
Average	**26.91 min**

DeepSWE v1.1

Task	Codex Latency
oxvg-structural-selector-preservation	19.71 min
wasmi-trap-coredumps	14.55 min
happy-dom-abort-pending-body-reads	12.52 min
scriggo-method-declarations	17.35 min
dynamodb-toolbox-lazy-recursive-schemas	22.19 min
dynamodb-toolbox-conditional-attribute	16.78 min
opa-template-string-reconstruction	13.99 min
boa-hierarchical-evaluation-cancellation	13.28 min
numba-stencil-boundary-modes	15.91 min
pebble-durability-wait-apis	19.38 min
Average	**16.57 min**

Table 6: Selected tasks from Terminal-Bench 4.0 and DeepSWE v1.1. Baseline latency is the average execution time of the Codex baseline using GPT-6-Astra at xhigh reasoning effort.

E EXPERIMENTS ON LONGBENCH-V2 MULTI-DOC QA

LongBench-v2 (Bai et al., 2025) is a long-context benchmark designed to evaluate deep understanding and reasoning over realistic tasks. We focus on its multi-document QA setting, which contains 125 samples across diverse domains: Financial (15 samples), Government (23 samples), Multi-News (23 samples), Legal (14 samples), and Academic (50 samples). This setting is also suitable for our evaluation because its questions require multi-hop reasoning over evidence distributed across long contexts, rather than extracting a single local span.

For the compared baselines, we additionally pick ReadAgent (Lee et al., 2024) which is an agentic long-context baseline that compresses the input into short gist memories and looks up the original passages when a subtask needs details the gists omit. For base models, we use Gemini 3 Flash (Doshi, 2025) and Claude Opus 4.6 (Anthropic, 2026a) for the SWE-bench experiments; and GPT-5.4 (OpenAI, 2026a), Claude Sonnet 4.6 (Anthropic, 2025), Gemini 3 Flash, and DeepSeek-V4-Pro (DeepSeek-AI, 2026) for the LongBench-v2 experiments.

As shown in Table 7, DELM achieves the highest average accuracy across all four model families: 60.1% with GPT-5.4, 59.8% with Claude Sonnet 4.6, 61.5% with Gemini 3 Flash, and 67.5% with DeepSeek-V4-Pro. Compared with the strongest baseline for each model family, these correspond to gains of 5.7, 5.3, 4.4, and 3.6 percentage points, respectively. DELM also achieves the best or tied-best result in most domain-model combinations, suggesting that the benefit comes from a general improvement in evidence selection and reuse rather than from a single model or domain.

E.1 COMBINING DELM WITH RLM

Recursive Language Models (RLMs) (Zhang et al., 2025) process long contexts by recursively selecting context segments, issuing sub-calls, and composing the resulting partial answers. Crucially, RLM performs this process through a code-mediated Read-Eval-Print Loop (REPL): the long input is stored as an external variable that the model can inspect, parse, and aggregate programmatically.

Table 7: Accuracy (%) comparison on LongBench-v2 Multi-Doc QA across four base models. Results are mean±std over 3 independent runs. Numbers in parentheses denote the number of samples in each domain. Best values are bolded.

Method	**Fin.** (15)	**Govern.** (23)	**MNews** (23)	**Legal** (14)	**Acad.** (50)	**Avg.**
GPT-5.4						
Base	62.2±9.3	39.1±1.7	50.5±8.6	62.0±4.4	56.0±1.2	53.9±4.9
Claude Code	65.0±8.3	36.2±2.0	50.7±2.0	59.5±3.4	**60.7**±0.9	54.4±3.1
ReadAgent	57.8±3.9	34.8±5.0	52.2±8.7	61.9±4.4	58.0±1.2	53.0±3.3
DELM	**71.1**±3.1	**43.5**±2.0	**65.2**±2.0	**64.3**±5.8	56.0±2.8	**60.1**±1.2
Claude Sonnet 4.6						
Base	68.9±3.9	34.8±1.1	46.4±2.0	52.4±3.4	64.0±0.9	53.3±1.9
Claude Code	66.7±1.1	42.0±2.0	47.8±4.4	50.0±3.4	52.0±3.4	51.7±2.4
ReadAgent	62.2±3.9	44.9±4.4	43.5±4.4	57.1±4.1	**64.7**±2.3	54.5±3.8
DELM	**73.3**±3.1	**46.4**±2.0	**55.1**±2.0	**59.5**±3.4	**64.7**±0.9	**59.8**±1.5
Gemini 3 Flash						
Base	80.0±0.0	30.4±0.0	43.5±0.0	**71.4**±0.0	60.0±2.3	57.1±3.3
Claude Code	73.3±3.1	26.1±1.5	39.1±0.0	57.1±2.1	52.0±1.9	49.5±2.2
ReadAgent	75.6±2.3	21.7±4.4	46.4±2.9	69.0±3.4	62.0±2.3	54.9±1.1
DELM	**82.2**±3.1	**37.8**±1.6	**52.2**±1.5	**71.4**±0.0	**64.0**±0.0	**61.5**±0.6
DeepSeek-V4-Pro						
Base	**86.7**±3.1	47.8±2.0	44.9±2.0	**76.2**±3.4	64.0±0.9	63.9±1.3
Claude Code	84.4±3.1	36.2±2.0	56.5±4.4	64.3±2.1	48.7±0.9	58.0±0.9
ReadAgent	71.1±3.9	43.5±4.4	30.4±2.3	38.1±4.1	50.7±1.2	46.8±1.8
DELM	**86.7**±5.4	**52.2**±2.0	**59.4**±4.4	73.8±3.4	**65.3**±0.9	**67.5**±1.2

Table 8: Comparison of RLM, DELM, and their hybrid on LongBench-v2 Multi-Doc QA and OOLONG using GPT-5.

Method	**Acc.**	**Cost/Task**
OOLONG		
RLM	56.0%	$0.43
DELM	53.3%	$0.47
DELM+RLM	**64.0%**	**$0.40**
LongBench-v2 Multi-Doc QA		
RLM	55.8%	$0.29
DELM	57.9%	$0.30
DELM+RLM	**60.3%**	**$0.24**

This makes RLM a natural fit for OOLONG (Bertsch et al., 2025), an aggregation-heavy benchmark where each sample consists of timestamped, user-attributed entries and answering a question often requires classifying many entries before computing a distributional answer.

We first evaluate DELM and RLM separately on both LongBench-v2 Multi-Doc QA and OOLONG using GPT-5 with medium reasoning (RLM uses GPT-5-mini for sub-calls and recursion depth=1). As shown in Table 8, the two methods have complementary strengths. On LongBench-v2, DELM achieves higher average accuracy than RLM at roughly the same cost per task. On OOLONG, however, DELM underperforms RLM in both average accuracy and cost. This is because OOLONG is closer to a structured data-processing benchmark than a conversational reasoning benchmark (such as LongBench-v2): many questions require exact counting, filtering, comparison, and tie handling.

In this setting, the natural-language shared context used by DELM is not reliable. RLM performs better because its core interface is code-mediated: through a REPL environment, the model can inspect, parse, transform, and aggregate the input using executable programs and programmatic sub-calls. However, RLM still relies on centralized coordination, where recursive calls report back to a root process that must reconcile and aggregate their outputs. This centralized aggregation is less suitable for natural-language multi-document reasoning, helping explain RLM's weaker LongBench-v2 performance. These results motivate a hybrid design that combines RLM's precise REPL-based execution with DELM's decentralized coordination.

We therefore combine the two methods by retaining RLM as the underlying reasoner while adding two DELM components: a shared context and a decentralized task queue. In this hybrid setting, there is no main-agent/sub-agent hierarchy; all agents are equally ranked RLM instances that coordinate through shared state. An initial RLM call produces a compact work plan, after which subtasks are placed into the task queue and claimed by parallel RLM workers. Each worker's output is admitted into the shared context, and the system then generates any additional subtasks from the current problem state, and computes the final answer from the shared context once no further subtasks are needed. This design preserves RLM's strength in precise code-mediated execution while adding DELM's decentralized coordination. As shown in Table 8, the combined method achieves the best accuracy and lowest cost on both benchmarks, indicating that RLM and DELM are complementary rather than competing approaches.

F WHY DELM IS BOTH ACCURATE AND COST-EFFICIENT ON SWE-BENCH

Three trace-level mechanisms explain why DELM improves both accuracy and cost efficiency. In the DELM traces below, each shared entry is one thread's note, tagged with its thread id (`t0`, `t1`, ...) and a type such as `FACT`, `FAIL`, or `PATCH_SUMMARY`.

(1) Parallel agents complement each other by sharing failures. The first mechanism is that failed hypotheses become reusable state rather than private dead ends. In isolated forks, a negative result remains local to one trajectory, so other attempts may spend their own budget rediscovering the same failure. In DELM, once such a failure is admitted to the shared context, later agents can treat it as a constraint and redirect their search.

This mechanism appears in the trace below. The task fixes `lambdify` for single-element tuples. The natural guess, the Python printer layer, is a red herring; the real bug is on a separate tuple-building path in `sympy/utilities/lambdify.py`:

```
[t0/FACT] AbstractPythonCodePrinter change did not affect lambdify
    output
[t1/FACT] sympy/utilities/lambdify.py:964 _recursive_to_string manual
    tuple join bypasses printer fix
[t0/FACT] sympy/utilities/lambdify.py:964 manual join for tuples lacks
    trailing comma logic
```

The decisive update is the negative result in the first line: thread `t0` shows that changing the printer does not affect the output. After reading this failure, `t1` avoids repeating the same detour and localizes the actual bypass in `_recursive_to_string`; `t0` then records the concrete defect, the missing trailing comma. Thus, DELM turns a failed hypothesis into shared progress, improving later agents' search efficiency.

(2) Admitted constraints remain binding shared state. The second mechanism is that DELM preserves important constraints as shared state rather than routing them through a central controller. In centralized coordination, a main agent may soften, omit, or reopen constraints discovered by sub-agents before they can guide later work.

This mechanism appears in the trace below. Folding multiple search filters into one `.filter()` call is unsafe for multi-valued relations, where `.filter(A).filter(B)` and `.filter(A, B)` differ. In AOrchestra, a sub-agent finds exactly this danger, but the constraint must pass through the main agent, which reopens the optimization and softens it into a tradeoff about reducing joins "for

many use cases"; the run fails. The relevant fact was found, but not preserved as binding state. In contrast, DELM keeps the constraint explicit and reusable:

```
[t3/FAIL] django/contrib/admin/options.py:1041 single .filter() breaks
    M2M multi-term search
[t3/FACT] Django ORM .filter(Q1, Q2) vs .filter(Q1).filter(Q2) differs
    for multi-valued relations
[t3/FACT] lookup_spawns_duplicates determines if single .filter() is
    safe
[t1/FACT] lookup_spawns_duplicates preserves M2M search semantics while
     optimizing FKs
```

Thread `t3` records the unsafe case (`FAIL`), the semantic reason, and the predicate `lookup_spawns_duplicates`, which determines when the optimization is valid. Thread `t1` then builds on this state to preserve M2M semantics while still optimizing foreign-key cases. Thus, later threads inherit the binding constraint rather than reopening a globally invalid simplification.

(3) Compact patch summaries carry discoveries. The third mechanism is compact sharing, which lowers cost by preventing peer communication from becoming another long raw transcript. Sharing full traces would preserve information, but would also expose every worker to command history, file dumps, failed edits, and intermediate reasoning. DELM instead shares a compressed version of the useful discovery, so later workers can reuse the result without reading the entire trajectory.

This mechanism appears in the trace below. Both the shared and no-share settings solve the task, but compact sharing reduces the cost from $0.399 to $0.125. The difference is not that DELM shares more context; rather, it shares a compressed version of the useful search result (labeled as `PATCH_SUMMARY`). One thread first rules out the ordinary tuple-printer path and localizes the actual bypass in `lambdify.py`:

```
[t1/FACT] StrPrinter._print_tuple at sympy/printing/str.py:868 already
    includes trailing comma logic
[t1/FACT] sympy/utilities/lambdify.py:964 lacks trailing comma for
    single-element tuples
[t1/PATCH_SUMMARY] files=sympy/utilities/lambdify.py | idea=Modify
    _recursive_to_string to add a trailing comma for single-element
    tuples | evidence=reproduce_issue.py PASSED
```

This patch summary turns a multi-step debugging trajectory into a short, evidence-backed handoff. Later workers do not need the full command history, file dumps, failed edits, or intermediate reasoning. Thus, compact sharing preserves the reusable discovery while avoiding the cost of exposing every peer to the raw trace, explaining why DELM solves the task at much lower cost.

Together, these examples explain the trend in Table 4: DELM keeps reusable facts, failures, constraints, and patch summaries visible to all peers, while avoiding both redundant isolated search and the high token cost of raw trace sharing. This improves per-thread success while reducing cost.

G PROMPTS

DeLM Prompt ($n = 2$, Part 1): Initial Instructions

`[DeLM AGENT A]`

Workspace.
Your task workspace is {workspace}. Do not spawn extra agents.

Role.
You are {worker}, one of two agents solving this task in parallel. Roster: {roster}.

Objective.

Minimize elapsed time to a correct, complete submission. Start useful work promptly, unblock your peer with usable results, and reuse applicable evidence.

Coordination.

- Divide work and exchange findings and text files through the board. It is the only connection between workers. Each worker has a separate container, so edits and installs are local. Every worker submits a complete solution from its own workspace.
- The board is ready immediately. The first worker creates the initial queue: it reserves its own scope, then publishes actionable peer work or states the concrete dependency preventing a split. After that handoff, both can publish, reserve, and claim distinct work. If the first worker ends before the handoff, the other takes over. Both can share findings immediately.

Task requirements.
The original task remains authoritative for both workers. Queue items divide work; their summaries never replace or weaken its requirements. Preserve every required observable effect, including state changes and nonvisual side effects.

Performance.

- For performance-constrained work, choose the complete first-call path before fixing component interfaces. In an existing initial item or FACT, state what the producer prepares, what the consumer receives directly, and what substantial work remains on the first required invocation. Retained state is not directly usable if that invocation must rebuild another representation. Design preparation and updates within the task's required lifecycle, preserving fresh-start, mutation and lifetime semantics.
- For decision or search operations, consider cheap sufficient conditions for exact acceptance and rejection before general preparation or search. An early answer must satisfy every required condition; a heuristic or necessary condition is not enough. Use the complete method when no such condition establishes an answer.
- Share the interface early and implement independent parts in parallel. When a specific interface cost could prevent a required target, resolve that concern with existing evidence or a small probe before expanding code dependent on that choice. Keep unaffected work moving. Address a measured bottleneck before extending an expensive comparison; reuse compatible components.

Your initial work:
Inspect the task and existing files to choose the simplest approach that can satisfy all stated requirements and identify complementary, reusable implementation outputs. Read the board and incorporate useful peer findings. Establish the initial split: reserve your scope and publish actionable, nonoverlapping work for your peer, with available inputs, interfaces and the first usable dependency. Do not wait to complete a whole-task plan before publishing useful work and implementing your part in this same session. If no independent item is clear yet, state your scope and the unresolved dependency so your peer can help resolve it. Add remaining work only as needed.

DeLM Prompt ($n = 2$, Part 2): Board Commands

`[DeLM AGENT A]`

Board commands:
`sh /tools/context.sh ACTION ...` through your ordinary shell tool.

- `read [--all] [--wait SECONDS] [--for ID]`
  The first view is complete. Later views show active work, new publications and changed statuses since your last served view. Unchanged active items show their identity and owner. Session states (starting/running/ended) are always shown. A session ending does not establish task correctness. Use read --all for older entries or a missed response. A waiting read returns on a peer publication or a peer ending, otherwise after at most 25 seconds. Status changes do not end a wait. --for names a needed item or publication and returns if it is available already.

- `publish --items JSON`
  Publish open items as a list of objects with subject and detail, following the startup roles above. Publishing does not claim ownership. Matching subjects reuse existing items without changing their owner, detail, or completion state. Subject target: 80, detail: 300 characters.
- `claim --item ID`
  First claim wins; a loss names the owner. One unfinished claim per worker.
- `reserve --subject TEXT --detail TEXT`
  Atomically create and claim one scope if your claim slot is free. Matches reuse open items, report claimed owners, or remain done. Subjects match ignoring case and extra whitespace. Different titles can still overlap, so respect visible scopes.
- `done --item ID --summary TEXT [--attach PATH...]`
  Finish your claim with results, affected files/functions, limitations, and check outcomes already obtained (summary target: 800 characters). Attach reusable text files from your workspace or /tmp. Publication returns its immutable ID.
- `note --type FACT|FAIL --text TEXT --scope TEXT --basis TEXT [--attach PATH...]`
  FACT means a finding supported by observations. FAIL means a contradicted approach or hypothesis. State scope and reproducible evidence, including the inputs, conditions and actual results behind measurements. Identify unfinished parts. For measurements, include the input construction, sample count, starting state, metric and aggregation. Attach the runnable command or script when sharing its result. Targets in characters: text 400, scope 120, basis 200. Keep attachments focused (target: 20 files and 200,000 bytes per bundle).
- `apply --id ID [--only PATH...]`
  Copy attachments over original or registered peer versions. Unknown local edits refuse the selected batch; inspect and reconcile locally. Identical files and newer peer versions are kept.
- `expand --id ID [--only PATH...]`
  Show attachment diffs against task originals, not your current files. New files and --only paths are shown in full.
- `status --text TEXT`
  State a changed question, configuration, dependency or scope (target: 200 characters). Claim, reserve and done already set work status.

DeLM Prompt ($n = 2$, Part 3): Coordination

`[DeLM AGENT A]`

Reading the board.
Read the board at startup, before a new implementation subproblem or substantial diagnostic hypothesis, before final integration, and before submitting. Choose a new diagnostic after receiving the view, not in the invocation that first requests it. If a peer owns or answered that question, choose a distinct unresolved one unless changed conditions or contradictory evidence justify repetition. Command responses already include fresh board views, which replace a separate read at those boundaries. Routine local checks do not each require a read, status, or reservation.

Sharing results.
Share usable changes promptly when they can unblock a peer. State unfinished parts, known limitations, interfaces, and existing check evidence. New checks are not required merely to share partial work or close a subtask. Closing a subtask does not establish that the complete solution meets the requirements. Do not hold reusable work until unrelated work is finished. When outputs or service state cannot be attached, share the generating or replay code and exact command and inputs to reproduce them locally. When no decision depends on a returned view, combine a board publication and the next local command in one shell call. Avoid repeating evidence across board fields and the final answer. Do not add documentation or a new build wrapper unless the task requires it or it is needed to produce or check the submitted output.

Waiting:

do useful independent implementation, integration or diagnosis when available. Wait only for a specific required input from a starting or running peer when none remains. Name the missing input in status and inspect the returned board view before deciding to wait. Do not batch that response with a blocking wait that hides it. If still needed, use read --wait 25 (--for ID when known) with yield_time_ms: 30000, reassessing each returned view before waiting again. Never use sleep, repeated polling, duplicate probes, redundant checks or paperwork to fill the gap. Never wait for an ended peer or a nonrequired addition.

Long commands.
Run necessary long diagnostics asynchronously when useful independent work remains, exposing intermediate results. Preserve their inputs and code; results apply to those versions. Read new output when it can change a decision instead of repeatedly polling. If a peer update could make a tool wait unnecessary, read the board before blocking, not after the wait in the same call. Avoid competing workload during timing measurements. Wait only for evidence still required for completion.

The queue:

- Queue items are for required implementation, repair of a demonstrated defect, or a concrete unanswered question needed to implement the solution. Do not create separate verification, testing, review, reproduction, or solution-checker assignments, including at a peer's request. This rule applies to queue items created with publish or reserve. Diagnosis and measurements needed to choose or repair an implementation remain allowed. A checker that is itself the requested product is implementation.
- Name each item's output, available inputs, shared interface and first usable dependency. Prefer complementary outputs that both workers can implement independently. Preserve required public behavior, interfaces and side effects. Reuse compatible components. Pursue a different approach to resolve a concrete limitation, sharing compatible parts. Keep related shared-file edits together. Do not invent work just to occupy a peer.
- After the initial handoff, claim an existing item when it fits; otherwise reserve a distinct required implementation scope. Respect the returned owner. Ownership covers an output, not a fixed algorithm. Publish a needed component when a peer can implement it independently, stating its interface, inputs and nonoverlapping scope while continuing your part. Keep small local adjustments under the current claim. Routine local integration and necessary checking need no separate item or active claim. Respect ownership and interfaces while the owner is running. Report concrete defects through scoped FAIL notes. To hand off incomplete work, close the claim with its remaining scope and reusable files. An ended worker's claim stays with it, but cannot block you from completing required work. Keep completed items closed. Do not assign verification roles through notes or status. Requests for existing evidence and interface clarification remain useful. Initial planning does not make either worker a permanent coordinator or checker. There is no global plan-closure step.

DeLM Prompt ($n = 2$, Part 4): Validation and Completion

`[DeLM AGENT A]`

Validation and completion:

- Derive necessary checks from the original task, not the queue summary or the current implementation. Preserve its measurement domain and apply each stated exception only to the requirement it modifies. Assess your own and peer evidence against those requirements. Reuse applicable results; adoption alone does not require repeating a check. Evidence must match the relevant code, inputs, configuration and environment of the final local output.
- For a new comparison, obtain matched candidate and reference observations before collecting repeats. Repeat as required by the task or where uncertainty could change the implementation or acceptance decision, preserving the required starting conditions. Preserve inputs, expected results and reference measurements. After candidate-only changes, rerun affected candidate checks; reuse reference measurements only under matching inputs, reference version, procedure and operating conditions.

- Publish an observed relevant failure promptly as FAIL with its conditions. Keep it unresolved until corrected code or a task-based explanation addresses it; changing the sample or averaging it with unrelated cases is not a repair. When ground truth is unavailable, use available references, convergence or invariants and state what remains unmeasured. Never turn that limitation into a claim of measured success.

Submission.
Before submitting, read the board and address published corrections relevant to your final local solution. A required repair is a dependency even if an earlier artifact is usable. Do not finish with a known unresolved failure of a required behavior or performance target. Once applicable evidence supports satisfying all requirements and no relevant failure remains, preserve that candidate and submit. An optional stronger result or an already-running search is not a reason to delay. Prevent exploratory work from changing the submitted output. Close any claim you still hold with done. If your item is already done, publish required corrections in a note rather than opening a claim merely to finish it. Submit a concise answer with required explanations and known limitations. A peer still working does not block a complete local submission.

Four-agent configuration. For $n = 4$, the two additional agents initially identify reusable components and interfaces, and resolve a concrete design uncertainty. These directions yield to available implementation work. The prompt also clarifies cross-author file conflicts, interface changes, local state and artifact reuse, container resource limits, and measurement intervals. Shared instructions and the first two startup prompts also contain wording changes.