Dynamic Context Allocation for Concurrent Multi-Agent LLMs

Junho Na[1,2,*], **Daiki E. Matsunaga**[2], **Namho Koh**[2],
Jongmin Lee[3], **Pascal Poupart**[4], **Kee-Eung Kim**[2,†]
[1]KT Corporation [2]KAIST [3]Yonsei University [4]University of Waterloo
junho.na@kt.com {jhn9803,d.e.matsunaga,namhokoh,kekim}@kaist.ac.kr
jongminlee@yonsei.ac.kr ppoupart@uwaterloo.ca

Abstract

Concurrent multi-agent LLM systems increasingly rely on role-specialized Workers that act in parallel, but effective coordination depends not only on what Workers generate, but also on what task state each Worker observes before generating. Existing systems usually fix this observation interface through workflow rules, retrieval heuristics, or full-history sharing, so information flow cannot adapt as task state and feedback evolve. We introduce **Dynamic Context Allocation (DCA)**, which treats per-Worker observation control as a trainable coordination problem for a fixed-role team. An RL-trained *Context Allocator* maintains a bounded *Context Board* and decides before each round which evolving task-state entries should be retained, revised, removed, or exposed to each Worker, rather than learning agent selection, execution order, or workflow topology. We evaluate DCA on cooperative code generation, Customer Relationship Management (CRM) lead qualification, and **MultiWebGen**, a new benchmark for multi-agent full-stack web development. Across these settings, DCA improves over fixed-protocol coordination and fixed-context multi-agent RL baselines, and it outperforms all evaluated frontier single-agent LLMs on CRM lead qualification despite using smaller role-specialized Workers.

1 Introduction

Recent collaborative LLM systems increasingly decompose complex tasks across multiple role-specialized agents that act concurrently, as in code generation (Liu et al., 2026b; Shi et al., 2026), software development (Anthropic, 2025), and deep research (Hadfield et al., 2025; Perplexity, 2026). This design matches the structure of many tasks: different agents can focus on different sub-problems, while concurrent execution improves coverage and throughput. However, these sub-problems are rarely independent. One agent's action often changes what another agent should assume, implement, or verify. A central challenge is therefore not only what each agent should *generate*, but what task state each agent should *observe* at each step.

Context allocation determines which messages, task-states, memories, and intermediate artifacts are exposed to each role (Han et al., 2024; Tran et al., 2025). Poor context allocation can hide critical updates, preserve obsolete assumptions, or distract agents with irrelevant traces, leading to redundant work and inconsistent outputs (Wu and Shu, 2026; Xiong et al., 2026; Liu et al., 2025). Existing multi-agent LLM systems largely treat context allocation as a fixed design choice. Workflow-based systems route information through hand-specified roles, subscription rules, or heuristic relevance scores (Hong et al., 2024; Liu et al., 2025), while conversation- and review-based systems often expose agents to full dialogue histories or peer-generated artifacts (Wu et al., 2024; Du et al., 2024). Recent RL-trained concurrent-agent methods improve what agents generate, but still condition them on fixed observation interfaces, often full accumulated histories (Liu et al., 2026b,a). This static observation interface can leave agents with incomplete, stale, or overly broad context, so artifacts that are plausible for an individual role may still be misaligned with other agents' current decisions or newly observed feedback (Liu et al., 2025; Cemri et al., 2025).

We introduce **Dynamic Context Allocation (DCA)**, which makes the observation interface of a concurrent multi-agent LLM system a learned component of the policy. DCA does not learn a new workflow, choose which agents to invoke, or change the Workers' roles. Instead, it augments a fixed-role concurrent team with a *Context Allocator* that maintains a bounded *Context Board* and constructs role-specific Worker views before each generation round. Through context-editing opera-

tions, the Context Allocator learns which task-state entries should be retained, revised, discarded, or exposed to each Worker, so coordination is learned at the level of information access rather than only at the level of generated actions. We evaluate DCA on cooperative code generation, full-stack web development, and Customer Relationship Management (CRM) lead qualification. Our contributions are:

- We isolate *observation control* as a distinct coordination problem for concurrent multi-agent LLMs: when Workers act in parallel, the system must decide what evolving task state each Worker should observe before it generates. This differs from orchestration and conductor-style work, which primarily controls agent selection, sequencing, or topology.
- We propose **DCA**, an RL-trained Context Allocator that maintains a bounded Context Board and learns role-specific visibility over task-state entries for a fixed-role concurrent team. This separates learned information access from Worker task generation and from workflow construction.
- We introduce **MultiWebGen** and provide empirical evidence across code generation, web development, and CRM lead qualification that learned observation control improves over fixed-protocol coordination and fixed-context multi-agent RL baselines; ablations further show that the gains are not explained by merely adding more context, training Workers, or using role specialization alone.

2 Related Work

Context Allocation in Multi-Agent LLM Systems. Multi-agent LLM systems coordinate role-specialized agents through workflows, dialogue, tools, memory, and orchestration. Workflow-based systems such as MetaGPT (Hong et al., 2024) route artifacts through predefined stages or subscriptions, while conversation- and review-based systems such as AutoGen (Wu et al., 2024) and MAD (Du et al., 2024) expose agents to shared dialogue traces or manually configured message-passing policies. RCR-Router (Liu et al., 2025) is closer to our setting because it routes structured memory entries to roles, but its routing is still based on hand-designed relevance, stage, and recency heuristics. Memory-R1 (Yan et al., 2026) instead learns a Memory Manager that edits a memory bank for a downstream Answer Agent in multi-session question answering, but it constructs memory for a single consumer rather than role-specific views for multiple concurrent Workers. Recent orchestration and conductor-style methods optimize agentic workflows, communication topology, or natural-language worker instructions (Fourney et al., 2024; Nielsen et al., 2026; Wang et al., 2026; Hu et al., 2025; Zhang et al., 2025). These approaches decide who acts, in what order or topology, and under what instructions. In contrast, DCA targets a deployment-constrained coordination layer that prior orchestration methods largely leave fixed: for a given role structure, tool interface, and output contract, it learns how evolving task state should flow within that organization. This distinction matters because learning an entire workflow entangles coordination with agent design, role discovery, tool use, and output formatting, whereas many deployed or benchmarked multi-agent systems already specify these interfaces. Although DCA trains a task-specific Context Allocator in our experiments, it restricts learning to observation control rather than redesigning the agent organization itself. DCA therefore replaces static routes, retrieval heuristics, and workflow-level information flow with a learned policy over what each Worker observes before acting.

Learning to Coordinate Concurrent LLM Agents. Recent reinforcement-learning methods optimize multi-agent policies from shared task rewards, improving coordination in concurrent settings. MAGRPO (Liu et al., 2026b) extends GRPO (Guo et al., 2025) to multi-agent reinforcement learning by optimizing multiple LLM agents from shared rewards, while CoLLM-CC (Liu et al., 2026a) trains LLM agents with a learned centralized critic. These methods make Worker behavior or training-time evaluation more adaptive, but the execution-time observation structure remains largely fixed by the designer. DCA instead keeps the Worker roles fixed and learns a separate observation-control policy: the Context Allocator decides which information should be retained, removed, or exposed to each Worker before concurrent generation. This separation isolates a coordination bottleneck that prior multi-agent RL methods largely leave manual.

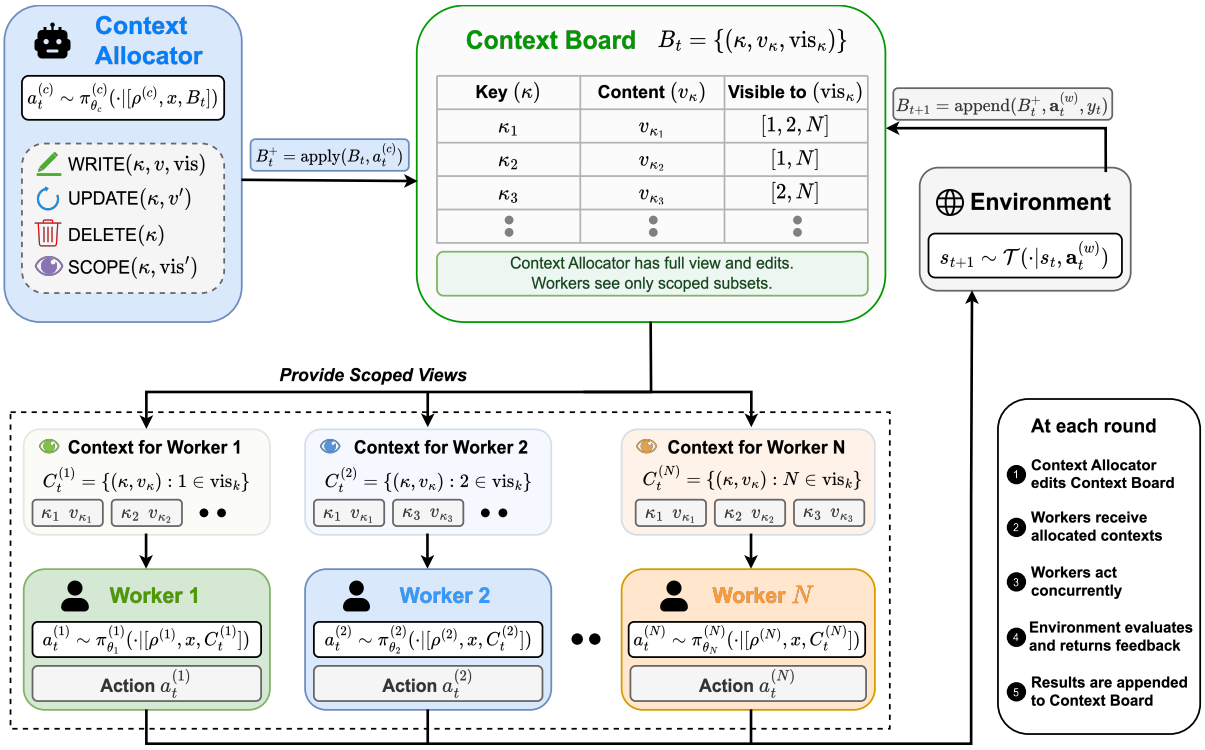

Figure 1: Overview of DCA. A Context Allocator edits a bounded Context Board through symbolic operations and constructs role-specific views for Workers. Workers act concurrently from their scoped contexts; their actions are assembled into a benchmark-specific team artifact and evaluated by the environment, and the resulting actions and feedback are appended to the Context Board for the next coordination round.

3 Preliminaries

3.1 Decentralized Partially Observable Markov Decision Process

We use a cooperative decentralized partially observable Markov decision process (Dec-POMDP) as the base formalism for concurrent multi-agent LLM collaboration (Oliehoek and Amato, 2016; Peralez et al., 2025; Liu et al., 2026b). A Dec-POMDP is defined as

$$\mathcal{M} = \langle \mathcal{N}, \mathcal{S}, p_0, \{\mathcal{A}^{(i)}\}_{i\in\mathcal{N}}, \{\Omega^{(i)}\}_{i\in\mathcal{N}}, \mathcal{T}, O, R, H, \gamma \rangle.$$

Here, $\mathcal{N} = \{1, \ldots, N\}$ is the agent set, $\mathcal{S}$ is the environment state space, $p_0 \in \Delta(\mathcal{S})$ is the initial-state distribution, H is the episode horizon, and $\gamma \in [0, 1]$ is the discount factor. Each agent i has its own action space $\mathcal{A}^{(i)}$ and observation space $\Omega^{(i)}$, yielding joint spaces

$$\mathcal{A} = \prod_{i\in\mathcal{N}} \mathcal{A}^{(i)}, \qquad \Omega = \prod_{i\in\mathcal{N}} \Omega^{(i)}.$$

At round t, agents choose a joint action $\mathbf{a}_t = (a_t^{(1)}, \ldots, a_t^{(N)}) \in \mathcal{A}$. The environment then evolves according to

$$s_{t+1} \sim \mathcal{T}(\cdot \mid s_t, \mathbf{a}_t), \quad \mathbf{o}_{t+1} \sim O(\cdot \mid s_t, \mathbf{a}_t, s_{t+1}),$$

where $\mathbf{o}_t = (o_t^{(1)}, \ldots, o_t^{(N)})$ is the joint observation. The reward $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is shared by all agents.

Because the state is only partially observed, each agent acts from its local action-observation history,

$$a_t^{(i)} \sim \pi^{(i)}(\cdot \mid h_t^{(i)}),$$

where $h_t^{(i)} = (o_0^{(i)}, a_0^{(i)}, o_1^{(i)}, \ldots, a_{t-1}^{(i)}, o_t^{(i)})$. The cooperative objective is to learn a joint policy $\boldsymbol{\pi} = (\pi^{(1)}, \ldots, \pi^{(N)})$ that maximizes expected shared return:

$$\max_{\boldsymbol{\pi}} \mathbb{E}_{\boldsymbol{\pi}} \left[\sum_{t=0}^{H-1} \gamma^t R(s_t, \mathbf{a}_t) \right].$$

3.2 Role-Specialized Multi-Agent LLM Collaboration and Context Allocation

We instantiate this formalism in round-based LLM teams with persistent task roles. Each episode begins with a task prompt x. Worker i is assigned a role prompt $\rho^{(i)}$ that specifies its responsibility

and output interface, such as producing a backend implementation, a frontend implementation, or evidence for one decision criterion. At round t, the system supplies Worker i with a role-specific context $C_t^{(i)}$, and the Worker generates

$$a_t^{(i)} \sim \pi_{\theta_i}^{(i)}(\cdot \mid [\rho^{(i)}, x, C_t^{(i)}]).$$

The action $a_t^{(i)}$ may be a natural-language message, program fragment, tool call, database query, or other benchmark-defined artifact. Workers act concurrently within a round, so $a_t^{(i)}$ cannot depend on another Worker's same-round action. After the Worker actions are assembled according to the task interface, such as concatenating main and auxiliary functions or placing backend and frontend artifacts into their designated project files, the environment returns a shared reward and may also emit feedback y_t, such as execution traces, test failures, or tool results.

The central design choice is therefore the construction of $C_t^{(i)}$. Fixed full-history or rule-based systems define $C_t^{(i)}$ through a predetermined observation protocol. DCA instead treats context allocation as a learned coordination problem: a Context Allocator maintains the task-state and determines which information should be retained, revised, removed, and exposed to each Worker before the next concurrent generation step.

4 Method

4.1 Dynamic Context Allocation (DCA)

In DCA, the *Context Allocator* serves as a learned context manager over a bounded *Context Board*, while fixed-role Workers produce task artifacts from their assigned views. We augment the N fixed-role Workers with a Context Allocator c:

$$\tilde{\mathcal{N}} = \{c, 1, \ldots, N\}.$$

The Workers produce benchmark-defined task actions, while the Context Allocator manages the information state from which those Workers act. The Context Allocator does not directly produce a benchmark-evaluated task artifact, and Workers do not directly edit the Context Board; this separation makes context allocation a distinct learnable policy.

At coordination round t, the augmented state is $\tilde{s}_t = (s_t, B_t)$, where $s_t \in \mathcal{S}$ is the environment state and $B_t \in \mathcal{B}$ is the Context Board. The Context Board stores the task-state, including task constraints, Worker actions, environment feedback, and Context Allocator guidance.

The Context Board contains at most K active slots. Let Λ be the universe of symbolic slot keys and $\Lambda_t \subseteq \Lambda$ the active keys at round t, with $|\Lambda_t| \leq K$. We write

$$B_t = \{(\kappa, v_\kappa, \mathrm{vis}_\kappa)\}_{\kappa \in \Lambda_t},$$

where κ is a Context Allocator-defined key, v_κ is textual content, and

$$\mathrm{vis}_\kappa \subseteq \{1, \ldots, N\}$$

specifies which Workers may access slot κ. The Context Allocator observes the full Context Board, while each Worker observes only the entries visible to its role. This separates global task-state maintenance from role-specific context construction.

At each round, the Context Allocator observes its instruction prompt $\rho^{(c)}$, the task prompt x, and the current Context Board, and generates a context-editing action

$$a_t^{(c)} \sim \pi_{\theta_c}^{(c)}(\cdot \mid [\rho^{(c)}, x, B_t]).$$

The action is an ordered sequence of symbolic operations:

- `WRITE`$(\kappa, v, \mathrm{vis})$: create a slot with key κ, content v, and visibility set vis.
- `UPDATE`(κ, v'): replace the content of an existing slot κ with updated content v'.
- `DELETE`(κ): remove slot κ and its content.
- `SCOPE`(κ, vis'): change the visibility set of slot κ to vis$'$ without changing its content.

These four operations form a compact interface that is equivalent to maintaining a separate context board per agent, but expressed as edits to a single task-state.

Executing the ordered operations in $a_t^{(c)}$ deterministically updates the Context Board while enforcing at most K active slots:

$$B_t^+ = \mathrm{apply}(B_t, a_t^{(c)}).$$

The capacity constraint is enforced by least-recently-edited eviction. `WRITE`, `UPDATE`, and `SCOPE` mark the affected key as recently edited, while `DELETE` removes it. Whenever the updated Context Board exceeds K active slots, apply evicts

the slots whose keys have gone longest without a Context Allocator edit until the capacity constraint is satisfied.

After these edits and possible evictions, B_t^+ is the information state from which the Workers act. Thus, the Context Allocator acts as part of the multi-agent policy, but its action updates the observation interface rather than directly producing a benchmark-evaluated task artifact.

Each Worker then receives its scoped context

$$C_t^{(i)} = \{(\kappa, v_\kappa) \mid (\kappa, v_\kappa, \mathrm{vis}_\kappa) \in B_t^+,\ i \in \mathrm{vis}_\kappa\},$$

and generates

$$a_t^{(i)} \sim \pi_{\theta_i}^{(i)}(\cdot \mid [\rho^{(i)}, x, C_t^{(i)}]).$$

Workers act concurrently, so no Worker conditions on another Worker's same-round action. The Worker actions are merged according to the benchmark-specific interface and evaluated by the environment:

$$s_{t+1} \sim \mathcal{T}(\cdot \mid s_t, \mathbf{a}_t^{(w)}),$$

where $\mathbf{a}_t^{(w)} = (a_t^{(1)}, \ldots, a_t^{(N)})$. The environment returns a shared reward r_t and may also emit feedback y_t, such as execution traces, verifier signals, test failures, or tool outputs. The system appends actions $\mathbf{a}_t^{(w)}$ and feedback y_t to the Context Board using a deterministic bookkeeping operation,

$$B_{t+1} = \mathrm{append}(B_t^+, \mathbf{a}_t^{(w)}, y_t).$$

Here, append enforces the same capacity rule as apply: when adding new $\mathbf{a}_t^{(w)}$ and y_t records would cause the Context Board to exceed K active slots, the least-recently-edited slots are evicted. These records become available to the Context Allocator at the next round, where they can be retained, revised, deleted, or scoped before Workers act again. Workers do not directly edit the Context Board because this would turn the board into a multi-writer memory whose contents depend on uncoordinated role-specific edits. DCA instead treats Worker actions and environment feedback as deterministic evidence appended to the board, and makes the Context Allocator solely responsible for learning which entries should persist, be removed, or be routed to each role. This design makes state maintenance and role-specific visibility the explicit learning problem, rather than a side effect of Worker-generated messages.

This defines one DCA coordination round: the Context Allocator edits a bounded Context Board, Workers generate concurrently from scoped role-specific views, the environment evaluates the merged team output, and the resulting evidence is appended to the Context Board for subsequent coordination. This process repeats until the fixed round budget is reached or the environment returns a success signal.

4.2 RL Training

Motivated by the recent success of reinforcement learning for LLM post-training (Guo et al., 2025), we cast DCA as a cooperative Multi-Agent RL problem where the Context Allocator and Workers are trained jointly from shared task rewards. We adopt MAGRPO (Liu et al., 2026b), a multi-agent extension of critic-free group policy optimization (Guo et al., 2025). For each task prompt x, we sample a group of G rollout trajectories $\{\tau^g\}_{g=1}^G$. Each trajectory consists of repeated Context Allocator edits, concurrent Worker generation, environment evaluation, and Context Board updates over horizon $T_g \le H$.

At round t of trajectory τ^g, the Context Allocator and Worker inputs are

$$o_t^{(c),g} = [\rho^{(c)}, x, B_t^g], \qquad o_t^{(i),g} = [\rho^{(i)}, x, C_t^{(i),g}],$$

where B_t^g is the current Context Board and $C_t^{(i),g}$ is the scoped context visible to Worker i. The Context Allocator samples a context-edit action $a_t^{(c),g}$, and each Worker samples a role-specific task action $a_t^{(i),g}$ following the DCA round defined above.

Let r_t^g denote the shared task reward at round t of trajectory τ^g. All Workers receive this shared reward. The Context Allocator additionally receives an auxiliary formatting reward $r_{t,\mathrm{fmt}}^g$ that encourages parseable and executable context-edit operations:

$$r_t^{(c),g} = r_t^g + r_{t,\mathrm{fmt}}^g, \qquad r_t^{(i),g} = r_t^g.$$

The corresponding discounted returns are

$$R_t^{(c),g} = \sum_{t'=t}^{T_g-1} \gamma^{t'-t} r_{t'}^{(c),g}, \quad R_t^{(i),g} = \sum_{t'=t}^{T_g-1} \gamma^{t'-t} r_{t'}^{(i),g}.$$

Following MAGRPO, returns are normalized within the rollout group to obtain advantages, yielding $A_{t,}^{(c),g}$ for the Context Allocator and $A_t^{(i),g}$ for Worker i.

We write the resulting update in policy-gradient form:

$$\mathcal{J}_c = \mathbb{E}\left[\sum_{g=1}^{G}\sum_{t=0}^{T_g-1} A_t^{(c),g} \log \pi_{\theta_c}^{(c)}\left(a_t^{(c),g} \mid o_t^{(c),g}\right)\right],$$

$$\mathcal{J}_w^{(i)} = \mathbb{E}\left[\sum_{g=1}^{G}\sum_{t=0}^{T_g-1} A_t^{(i),g} \log \pi_{\theta_i}^{(i)}\left(a_t^{(i),g} \mid o_t^{(i),g}\right)\right].$$

The training objective maximizes the Context Allocator objective and all Worker objectives jointly. Thus, Workers learn role-conditioned task policies, while the Context Allocator learns context edits that maintain and route the information needed for team-level success. Full prompts, qualitative results, a step-by-step running example of Context Board evolution, and pseudocode are provided in Appendix I–L.

5 Experiments

5.1 Benchmarks

CoopHumanEval. CoopHumanEval is a collaborative code generation benchmark introduced by Liu et al. (2026b). Given a programming problem, two agents generate complementary components: a *main* function and an *auxiliary* helper function. The final program is evaluated by test cases. We report the Test Pass Rate, defined as the proportion of test samples that pass their test cases. Although CoopHumanEval provides a controlled setting for basic two-agent role collaboration, its coordination demands are limited: even with guardrails requiring the main function to use the auxiliary function, many tasks can still be largely solved within a single function, and each task is centered on one function-level problem rather than realistic complex-interface or multi-step collaboration.

MultiWebGen. To evaluate more realistic team collaboration, we introduce MultiWebGen, a full-stack web development benchmark which extends the tasks from WebGen-Bench (Lu et al., 2025) to the multi-agent setting. Given a shared web development task and Product Requirements Document (PRD), *backend* and *frontend* Workers receive role-specific instructions and concurrently implement their respective components. The backend Worker is responsible for server-side endpoints and state logic, while the frontend Worker is responsible for user-facing interactions and API calls. MultiWebGen requires concrete coordination because end-to-end (E2E) tests pass only when backend endpoints and frontend API calls are mutually consistent: missing endpoints or mismatched request/response formats cause failures even if one side is correct. Each task involves multiple endpoint-level functions or interactions and produces substantially longer code than CoopHumanEval, making MultiWebGen a more realistic testbed for multi-agent collaboration. We report E2E Pass Rate, defined as the proportion of provided end-to-end tests passed by each test sample, averaged across all samples. Details of the MultiWebGen construction are provided in Appendix B.

CRM Lead Qualification. We also evaluate on the *lead qualification* task from CRMArena-Pro (Huang et al., 2026), a benchmark for realistic CRM agents over Salesforce-style business data. Given a lead identifier and access to Salesforce-style CRM records, the system must decide whether the lead is qualified; if not, it must identify which BANT criteria are not satisfied: *Budget*, *Authority*, *Need*, and *Timeline*. We focus on this task because it naturally decomposes into a concurrent multi-agent setting and allows us to test whether DCA remains effective as the number of Workers increases beyond two. This task naturally induces four fixed roles, where each Worker is responsible for querying relevant CRM records and deciding whether the lead satisfies its assigned criterion. We report Exact Match Rate, where a prediction is correct only if the qualification decision and the set of failed BANT criteria exactly match the ground truth.

5.2 Baselines

For CoopHumanEval and MultiWebGen, we compare DCA against three families of multi-agent baselines: non-coordinated methods, fixed-protocol coordination methods, and fixed-context RL methods. Detailed baseline instantiations are provided in Appendix A.

Non-coordination. These baselines do not exchange information across Workers during coordination; each Worker relies only on its own local history, feedback, or independent samples. They include *Independent*, where each Worker uses only its own history and feedback; *Self-Refine* (Madaan et al., 2023), where each Worker independently critiques and revises its generated action; and *Self-Consistency* (Wang et al., 2023), which samples multiple candidate actions from each Worker and

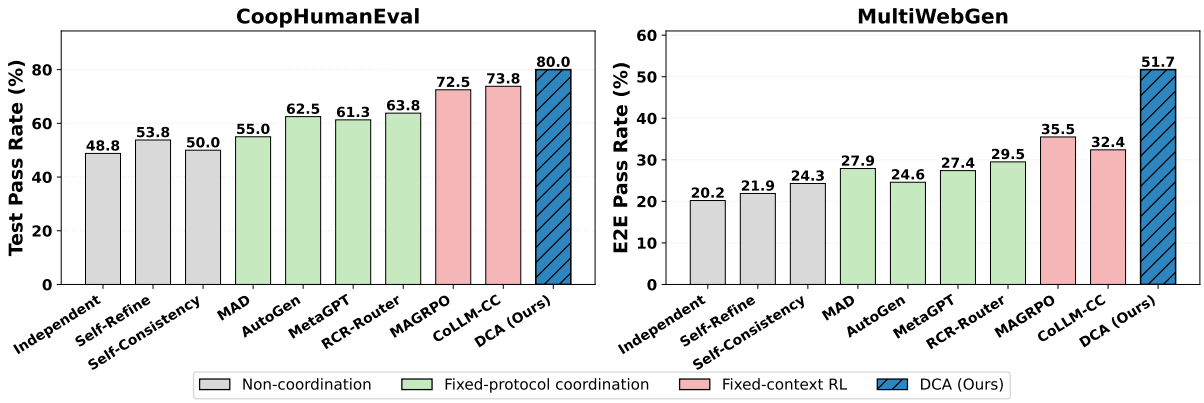

Figure 2: Main experimental results. DCA improves performance in our evaluated settings by learning how a Context Allocator should allocate task context to concurrent Workers. All reported values are averaged over five independent runs, with full statistics in Appendix E.

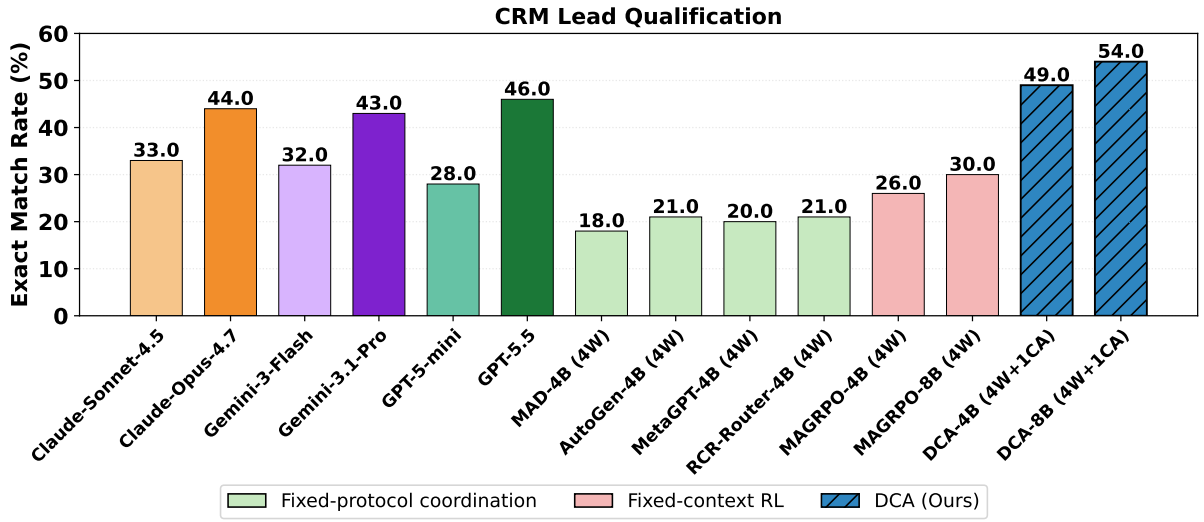

Figure 3: CRM lead qualification results. 4W denotes four role-specialized Workers and 1CA denotes one additional Context Allocator; −4B and −8B denote Qwen3-4B-Instruct and Qwen3-8B Worker backbones, respectively. DCA improves over multi-agent baselines and also outperforms frontier single-agent models despite using smaller backbone models. All reported values are averaged over five independent runs, with full statistics in Appendix E.

selects the best assembled team artifact using execution feedback.

Fixed-protocol coordination. These baselines allow cross-Worker information flow, but the observation protocol is fixed by hand-designed rules rather than learned from task rewards. For all fixed-protocol coordination baselines, we adapt the original methods to our concurrent generation setting while preserving their core observation protocols. *MAD* (Du et al., 2024) exposes each Worker to actions produced by the other Workers in the previous round before generation. *AutoGen* (Wu et al., 2024) formats previous Worker actions and environment feedback as a shared, role-attributed conversation history provided to all Workers. *MetaGPT* (Hong et al., 2024) routes task-state entries through hand-specified role-subscription rules. *RCR-Router* (Liu et al., 2025) selects structured memory entries with heuristic role-relevance, task-stage, and recency scores.

Fixed-context RL. These baselines train Worker policies with shared task rewards, but keep each Worker's input context fixed to its own role prompt, local action history, and feedback, without learning a separate context-allocation policy. They include *MAGRPO* (Liu et al., 2026b), which extends GRPO to multi-agent LLM training with shared rewards, and *CoLLM-CC* (Liu et al., 2026a), which uses a learned centralized critic to train decentralized cooperative LLM agents.

For CRM lead qualification, we compare DCA with larger frontier single-agent models and multi-agent baselines. Frontier models use the full context in a ReAct-style SQL tool-use loop without role decomposition or learned context allocation. Multi-agent baselines use four BANT-specific Workers that concurrently collect evidence, summarize it, and predict whether their assigned criterion is qualified or failed; the final prediction is qualified only if all Workers predict qualified, and otherwise fails the criteria predicted as failed.

5.3 Main Results

Our results for CoopHumanEval and MultiWebGen in Figure 2 show that DCA consistently improves over all evaluated baselines. DCA improves over the strongest fixed-context RL baseline on CoopHumanEval and the margin is substantially larger on MultiWebGen. This pattern supports our central claim: when coordination depends on evolving cross-role state, such as endpoint names, request and response schemas, and state-update conventions, learning the observation interface provides a benefit beyond training Workers or applying fixed routing rules.

The comparison against fixed-context RL baselines is especially important. MAGRPO and CoLLM-CC improve Worker behavior through shared rewards, but they keep each Worker's observation interface fixed. DCA's gains over these baselines show that multi-agent RL benefits from learning not only what Workers generate, but also what task state each Worker observes before generation. [1] The non-RL baselines follow the same trend: fixed-protocol coordination methods improve over non-coordinated methods by adding communication or shared-memory structure, but their information flow remains prompt-defined through broad broadcast, fixed routing, or heuristic scores.

In Figure 3, DCA also performs strongly on CRM lead qualification. DCA with 4B Workers and a Context Allocator outperforms all evaluated frontier single-agent models and further widens the gap with 8B models. The contrast with MAGRPO is particularly informative: DCA with smaller 4B Workers and a Context Allocator outperforms MAGRPO with larger 8B Workers. This indicates that the gain is not explained by role specialization or Worker scale alone, but by learning how task state should be maintained and routed to each role.

[1] Appendix C analyzes the Context Allocator's context-editing operations, and Appendix D shows that learned context allocation itself provides substantial gains with frozen Workers.

Worker / Context	Coop HumanEval	Multi WebGen
Independent / Full history	46.3	15.5
Independent / Previous round	48.8	20.2
MAGRPO / Full history	68.8	31.1
MAGRPO / Previous round	72.5	35.5
DCA	80.0	51.7

Table 1: Comparison with static history-based context policies. Reported values are Test Pass Rates for CoopHumanEval and E2E Pass Rates for MultiWebGen.

Method	E2E Pass Rate (%)
MAGRPO	35.5
DCA / random `SCOPE`	23.5
DCA / broadcast-to-all	40.0
DCA w/o `UPDATE`	47.3
DCA w/o `DELETE`	48.6
DCA w/o `SCOPE`	43.1
DCA w/o `UPDATE+DELETE`	41.8
DCA	51.7

Table 2: Context-editing ablations on MultiWebGen.

5.4 Ablations

Static History Policies vs. Dynamic Context Allocation. We test whether static history exposure can replace dynamic context allocation. Full history gives each Worker all of its previous actions and feedback, while previous-round history gives only the immediately preceding action and feedback. Table 1 shows that both MAGRPO variants underperform DCA, and full history does not improve over previous-round history. This suggests that DCA's gains come from learned task-state maintenance and role-specific allocation, not simply from exposing more context, or shortening the prompt.

Context-Editing Operations. We evaluate two complementary aspects of the Context Allocator's editing behavior on MultiWebGen: how it controls visibility and how it modifies task-state. First, to isolate the effect of learned visibility, random `SCOPE` replaces learned assignments with random ones while retaining the remaining system components and experimental settings, whereas broadcast-to-all exposes every active Context Board slot to both Workers. Both variants underperform DCA in Table 2. Second, we remove individual context-editing operations from the Allocator's ac-

tion space. Among the evaluated single-operation removals, the largest drop comes from removing `SCOPE`; `WRITE` can still assign initial visibility in this variant, but the Allocator cannot revise it later. Although each operation-removal ablation remains above MAGRPO, the performance drops show that the evaluated editing operations help DCA revise, discard, and scope task-state effectively. Together, these results indicate that DCA benefits both from learned role-specific visibility and from the ability to maintain evolving task-state.

6 Conclusion

This work frames per-Worker observation control as a learnable coordination problem for concurrent multi-agent LLMs. DCA instantiates this view with an RL-trained Context Allocator that maintains a bounded Context Board, edits evolving task-state entries, and constructs role-specific Worker views before each generation round. Across code generation, full-stack web development, and CRM lead qualification, DCA improves over diverse baselines, showing that effective multi-agent coordination depends on learning what each Worker observes, not only what it generates. For future work, we plan to expand our experimental setup to evaluate teams with 8–16 roles beyond our 2 and 4 agent setup, as well as cross-task zero-shot transfer to new roles, tools, output schemas, or feedback channels.

Limitations

DCA adds one sequential Context Allocator generation before the Workers act in each round. On four-Worker CRM lead qualification, this increases latency, although DCA remains faster than serial execution of the same roles (Appendix H).

Ethical Considerations

This work aims to improve coordination among role-specialized LLM agents, rather than to automate any particular high-stakes decision process. However, DCA-like systems may be applied to domains such as enterprise decision-making or software development, where incorrect outputs, biased evidence, or inappropriate handling of sensitive information could have downstream consequences. Since the Context Board stores the task-state, including task constraints, Worker outputs, execution feedback, and Context Allocator guidance, practical deployments should incorporate data minimization, access control, logging, and privacy-preserving handling of sensitive records. In business-facing tasks such as lead qualification, DCA should be used as a decision-support system with appropriate human oversight, especially when outputs may affect customers, organizations, or allocation of opportunities.

Acknowledgements

This work was partly supported by the Advanced GPU Utilization Support Program and the Institute of Information & Communications Technology Planning & Evaluation (IITP), both funded by the Government of the Republic of Korea (Ministry of Science and ICT) (No. RS-2024-00457882, AI Research Hub Project; No. RS-2024-00343989, Enhancing the Ethics of Data Characteristics and Generation AI Models for Social and Ethical Learning; No. RS-2020-II200940, Foundations of Safe Reinforcement Learning and Its Applications to Natural Language Processing; No. RS-2019-II190075, Artificial Intelligence Graduate School Program (KAIST)). This work was supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korean Government (MSIT) (RS-2020-II201361, Artificial Intelligence Graduate School Program (Yonsei University)) and National Research Foundation of Korea (NRF) grant funded by the Korea government(MSIT) (RS-2026-25505230).

References

Anthropic. 2025. Claude code: Subagents.

Mert Cemri, Melissa Z Pan, Shuyi Yang, Lakshya A Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. 2025. Why Do Multi-Agent LLM Systems Fail? In *NeurIPS Datasets and Benchmarks Track*.

Yilun Du, Shuang Li, Antonio Torralba, Joshua B Tenenbaum, and Igor Mordatch. 2024. Improving factuality and reasoning in language models through multiagent debate. In *ICML*.

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. 2024. Magentic-One: A Generalist Multi-Agent System for Solving Complex Tasks. *arXiv preprint arXiv:2411.04468*.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, and 175 others. 2025. DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning. *Nature*, 645(8081):633–638.

Jeremy Hadfield, Barry Zhang, Kenneth Lien, Florian Scholz, Jeremy Fox, and Daniel Ford. 2025. How we built our multi-agent research system. *Anthropic Engineering Blog*.

Shanshan Han, Qifan Zhang, Weizhao Jin, and Zhaozhuo Xu. 2024. LLM multi-agent systems: Challenges and open problems. *arXiv preprint arXiv:2402.03578*.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. 2024. MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework. In *ICLR*.

Shengran Hu, Cong Lu, and Jeff Clune. 2025. Automated Design of Agentic Systems. In *ICLR*.

Kung-Hsiang Huang, Akshara Prabhakar, Onkar Thorat, Divyansh Agarwal, Prafulla Kumar Choubey, Yixin Mao, Silvio Savarese, Caiming Xiong, and Chien-Sheng Wu. 2026. CRMArena-Pro: Holistic Assessment of LLM Agents Across Diverse Business Scenarios and Interactions. *TMLR*.

Jun Liu, Zhenglun Kong, Changdi Yang, Fan Yang, Tianqi Li, Peiyan Dong, Joannah Nanjekye, Hao Tang, Geng Yuan, Wei Niu, Wenbin Zhang, Pu Zhao, Xue Lin, Dong Huang, and Yanzhi Wang. 2025. RCR-Router: Efficient Role-Aware Context Routing for Multi-Agent LLM Systems with Structured Memory. In *AAAI*.

Shuo Liu, Tianle Chen, Ryan Amiri, and Christopher Amato. 2026a. Learning Decentralized LLM Collaboration with Multi-Agent Actor Critic. In *ICML*.

Shuo Liu, Tianle Chen, Zeyu Liang, Xueguang Lyu, and Christopher Amato. 2026b. LLM Collaboration With Multi-Agent Reinforcement Learning. In *AAAI*.

Zimu Lu, Yunqiao Yang, Houxing Ren, Haotian Hou, Han Xiao, Ke Wang, Weikang Shi, Aojun Zhou, Mingjie Zhan, and Hongsheng Li. 2025. WebGen-Bench: Evaluating LLMs on Generating Interactive and Functional Websites from Scratch. In *NeurIPS Datasets and Benchmarks Track*.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-Refine: Iterative Refinement with Self-Feedback. In *NeurIPS*.

Stefan Nielsen, Edoardo Cetin, Peter Schwendeman, Qi Sun, Jinglue Xu, and Yujin Tang. 2026. Learning to Orchestrate Agents in Natural Language with the Conductor. In *ICLR*.

Frans A. Oliehoek and Christopher Amato. 2016. *A Concise Introduction to Decentralized POMDPs*, 1st edition. Springer Publishing Company, Incorporated.

Johan Peralez, Aurélien Delage, Jacopo Castellini, Rafael F. Cunha, and Jilles Steeve Dibangoye. 2025. Optimally Solving Simultaneous-Move Dec-POMDPs: The Sequential Central Planning Approach. In *AAAI*.

Perplexity. 2026. Introducing Perplexity Computer.

Xi Shi, Mengxin Zheng, and Qian Lou. 2026. Learning Latency-Aware Orchestration for Parallel Multi-Agent Systems. *arXiv preprint arXiv:2601.10560*.

Khanh-Tung Tran, Dung Dao, Minh-Duong Nguyen, Quoc-Viet Pham, Barry O'Sullivan, and Hoang D Nguyen. 2025. Multi-agent collaboration mechanisms: A survey of llms. *arXiv preprint arXiv:2501.06322*.

Siyu Wang, Ruotian Lu, Zhihao Yang, Yuchao Wang, Yanzhou Zhang, Lei Xu, Qimin Xu, Guojun Yin, Cailian Chen, and Xinping Guan. 2026. AgentConductor: Topology Evolution for Multi-Agent Competition-Level Code Generation. In *ICML*.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-Consistency Improves Chain of Thought Reasoning in Language Models. In *ICLR*.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. 2024. AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversations. In *COLM*.

Shanglin Wu and Kai Shu. 2026. Memory in LLM-based Multi-agent Systems: Mechanisms, Challenges, and Collective Intelligence. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*.

Zidi Xiong, Yuping Lin, Wenya Xie, Pengfei He, Zirui Liu, Jiliang Tang, Himabindu Lakkaraju, and Zhen Xiang. 2026. How Memory Management Impacts LLM Agents: An Empirical Study of Experience-Following Behavior. In *ACL*.

Sikuan Yan, Xiufeng Yang, Zuchao Huang, Ercong Nie, Zifeng Ding, Zonggen Li, Xiaowen Ma, Jinhe Bi, Kristian Kersting, Jeff Z. Pan, Hinrich Schütze, Volker Tresp, and Yunpu Ma. 2026. Memory-R1: Enhancing Large Language Model Agents to Manage and Utilize Memories via Reinforcement Learning. In *ACL*.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xiong-Hui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. 2025. AFlow: Automating Agentic Workflow Generation. In *ICLR*.

Appendix

A Detailed Implementation of Baselines

Shared Experimental Setup. For CoopHumanEval and MultiWebGen, we control the evaluation setup across all multi-agent methods. Each method uses the benchmark-specific task format, the same Worker backbone, a round budget of three rounds, the same decoding configuration, the same output extraction rule, and the same evaluator. The fixed roles are main/auxiliary Workers for CoopHumanEval, backend/frontend Workers for MultiWebGen, and budget/authority/need/timeline Workers for CRM lead qualification. Unless otherwise stated, baselines in the main results use previous-round history. Full-history conditioning is evaluated separately in the ablation study in Section 5.4. DCA uses the same Worker backbone and role definitions, but additionally introduces the learned Context Allocator and structured Context Board described in Section 4.2. For CRM lead qualification, we evaluate DCA under the role-specialized multi-agent setting described in Section 5.1, while frontier-model baselines use the single-agent tool-use protocol detailed below.

Independent. Each Worker operates without any cross-agent communication. The input to each Worker consists of the task description, its role instruction, its own previous output, and the latest environment feedback.

Self-Refine. Each Worker performs round-based iterative self-improvement. The input to each Worker consists of the task description, its role instruction, its previous output, the latest environment feedback, and an instruction to first critique its own previous output is appended to the end of follow-up prompt before producing a revised output. This process is performed independently for each Worker without any cross-agent interaction.

Self-Consistency. Each Worker generates multiple candidate outputs per round by sampling diverse responses. All combinations of candidate outputs across Workers are evaluated using the environment, and the best-performing combination is selected according to execution-based metrics. The selected outputs are then used as the input for the next round. This baseline relies on diversity and selection rather than explicit coordination between Workers.

MAD. Each Worker is allowed to observe other Workers' outputs from the previous round and revise its own output. The input to each Worker consists of the task description, its role instruction, its own previous output, the previous outputs of other Workers, and the latest environment feedback. This baseline captures collaborative refinement through reviewing other agents' outputs.

AutoGen. A conversation-formatting agent converts previous Worker outputs and environment feedback into a structured sequence of role-attributed dialogue turns, following the message-list format. Each Worker output is cast as an `assistant`-role turn and each environment feedback as a `user`-role turn, preserving full content without condensation. All Workers then receive this same dialogue history, together with their role instruction and previous output, when generating their responses. This baseline directly instantiates AutoGen-style multi-agent coordination via a shared message history, without additional summarization or filtering.

MetaGPT. A shared structured workspace is maintained across external rounds. At each round, a prompted LLM labeler converts previous Worker actions and environment feedback into structured workspace entries under a fixed metadata schema. The labeler annotates entries with metadata such as producer role, message type, affected interface, dependency type, failure category, and relevant role tags, but does not choose recipient Workers. Instead, each Worker retrieves entries whose metadata match fixed, hand-specified subscription rules for its role, together with its own previous action and environment feedback. This baseline adapts publish–subscribe communication: messages are labeled by a prompted LLM, but visibility is determined by predefined subscription rules rather than learned or reward-optimized context allocation.

RCR-Router. A shared structured memory is maintained across external rounds, adapting role-aware context routing. At each round, previous Worker outputs, evaluator feedback, and extracted interface or failure information are converted into structured memory entries. Each entry is annotated using a fixed metadata schema, including the producer role, round index, message type, task stage, affected interface, failure category, and relevant role tags. Unlike MetaGPT, Workers do not retrieve entries through hand-designed subscription rules. Instead, before each Worker is invoked, a router scores all memory entries using fixed role-relevance, task-stage, and recency features, and greedily selects the highest-scoring entries under a fixed token budget. For scoring, we follow the heuristic form of RCR-Router and compute

$$\begin{aligned}\alpha(m; R^{(i)}, S_t) = {} & w_{\text{role}}\text{Score}_{\text{role}}(m, R^{(i)}) \\ & + w_{\text{stage}}\text{Score}_{\text{stage}}(m, S_t) \\ & + w_{\text{rec}}\text{Score}_{\text{rec}}(m, t).\end{aligned}$$

$\text{Score}_{\text{role}}$ is 1 if the memory entry contains role-relevant tags or interface keywords for Worker i, and 0 otherwise. $\text{Score}_{\text{stage}}$ is 1 if the entry type matches the current stage, such as requirement, implementation, feedback, interface, or failure, and 0 otherwise. $\text{Score}_{\text{rec}}$ is an exponential recency weight based on the entry round index. Unless otherwise stated, we set all weights to 1.0 and greedily select entries in descending score order until the token budget is exhausted. The selected entries are inserted into the corresponding Worker prompt together with the task description, role instruction, the Worker's own previous output, and evaluator feedback. This baseline uses the heuristic scorer-based routing mechanism where structured memory is selectively exposed to each Worker under fixed scoring and token-budget rules rather than a reward-optimized context allocation policy.

MAGRPO. Workers are trained using shared task-level rewards and executable feedback, while the input to each Worker follows the same fixed context construction as the Independent baseline. No Context Allocator, shared memory, or cross-agent communication mechanism is introduced. This baseline isolates the effect of reinforcement learning on Worker policies without any explicit coordination mechanism.

CoLLM-CC. CoLLM-CC uses the same fixed-context multi-agent setting as MAGRPO: Workers are trained from shared task-level rewards and executable feedback, and each Worker receives the same context as in the Independent baseline. Unlike MAGRPO, CoLLM-CC introduces one learned centralized critic shared across Workers to estimate returns and compute policy advantages during training. The critic is used only during training; at execution time, there is no Context Allocator, shared memory, or cross-agent communication mechanism.

Baselines of CRM Lead Qualification. For the frontier-model baselines, we use the same single-agent tool-use protocol for all API models. The model receives the task instance, task metadata, target lead id, and a compact SQLite schema summary. At each round, the model may issue up to four SQL queries to gather evidence, observing the executed results after each query. In subsequent rounds, the model receives all the results including the accumulated SQL queries and observations from previous round. The model alternates between generating SQL queries and observing query results until it either has enough evidence to answer or reaches the SQL budget. The baseline does not use role decomposition, a Context Board, or multi-agent communication; a single model is responsible for both evidence retrieval and the final BANT decision.

Listing 1: Self-Refine Critique Prompt

```
Critique your previous {role} output above.
    Identify what is wrong, missing, or could be
     improved given the execution diagnostics.
    Be concise. Do NOT output any code; output
    ONLY the critique text.
```

Listing 2: AutoGen-Formatting Agent Prompt

```
You are the conversation-formatting agent of a
    shared-conversation
multi-worker system.

Your task is to convert previous Worker outputs
    and environment
feedback into a structured sequence of role-
    attributed dialogue
turns. This dialogue history will be shown to
    all Workers in the
next round as their shared conversational
    context.

Benchmark: {benchmark_name}
Required output slots: {role_slots}

Task description:
{task_description}

Role instructions:
{role_instructions}

Previous Worker outputs:
{previous_worker_outputs}

Environment feedback:
{environment_feedback}

Previous shared dialogue turns:
{previous_dialogue_turns}

Formatting rules:
1. Each Worker output must be cast as a single
    dialogue turn with
   role "assistant" and speaker attributed to
       the producing Worker
   (e.g., "[WorkerA]: ...").
2. Each environment feedback item must be cast
    as a dialogue turn
   with role "user" and speaker "environment".
3. Preserve the original content of each output
    or feedback
   verbatim. Do not paraphrase, condense, or
       omit information.
4. Append new turns after the previous shared
    dialogue turns in
   chronological order.
5. Do not invent content not present in the
    inputs.
6. Do not merge multiple outputs into a single
    turn.
7. Do not add commentary, interpretation, or
    meta-information
   beyond the structured turns.

Return only a valid JSON object in the following
     format:

{
  "dialogue_turns": [
    {
      "turn_id": "string",
      "role": "assistant | user",
      "speaker": "one of {role_slots} |
          environment",
      "content": "string"
    }
  ]
}
```

Listing 3: MetaGPT Message Labeler Prompt

```
You are the message-labeling agent of a role-
    subscription multi-worker system.

Your task is to convert previous Worker outputs
    and environment feedback into structured
    workspace entries.

Benchmark: {benchmark_name}
Required output slots: {role_slots}

Task description:
{task_description}

Role instructions:
{role_instructions}

Previous Worker outputs:
{previous_worker_outputs}

Environment feedback:
{environment_feedback}

Existing workspace entries:
{previous_workspace_entries}

Use the following fixed metadata schema.

Allowed producer_role values:
{role_slots}, environment, shared

Allowed message_type values:
```

```
implementation_decision, interface_contract,
    dependency, unresolved_question,
bug_report, test_feedback, algorithmic_edge_case,
     data_format,
api_signature, constraint, clarification,
    obsolete

Allowed dependency_type values:
none, requires_consistency, blocks_other_role,
    depends_on_other_role,
shared_assumption, evaluator_required_change

Allowed failure_category values:
none, syntax_error, runtime_error, test_failure,
     interface_mismatch,
missing_requirement, wrong_output_format,
    incomplete_implementation,
logic_error, uncertain

Allowed role tags:
{role_slots}, all

You must:
1. Create structured entries from previous
    Worker outputs and environment feedback.
2. Preserve source attribution.
3. Label each entry using only the allowed
    metadata values.
4. Mark information that is relevant to all
    Workers with the role tag "all".
5. Mark role-specific information with the
    relevant role tags.
6. Mark stale or superseded information as
    message_type "obsolete" when appropriate.

You must not:
1. Directly choose recipient Workers.
2. Create new output slots.
3. Change the benchmark task decomposition.
4. Invent information not supported by the
    inputs.
5. Perform semantic retrieval or learned routing.


The actual visibility of entries will be
    determined only by fixed subscription rules
    outside this prompt.

Return only a valid JSON object in the following
     format:

{
  "workspace_entries": [
    {
      "entry_id": "string",
      "source": "previous_worker_output |
          environment_feedback |
          previous_workspace",
      "producer_role": "one of the allowed
          producer_role values",
      "message_type": "one of the allowed
          message_type values",
      "content": "string",
      "affected_interface": "string or none",
      "dependency_type": "one of the allowed
          dependency_type values",
      "failure_category": "one of the allowed
          failure_category values",
      "role_tags": [
        "one or more of the allowed role tags"
      ],
      "status": "active | unresolved | resolved |
           obsolete"
    }
  ]
}
```

Listing 4: MetaGPT Entry Selection Prompt

```
You are the entry-selection module of a role-
    subscription multi-worker system.

Your task is to select workspace entries for a
    Worker using fixed role-specific
    subscription rules.

Worker slot:
{worker_slot}

Workspace entries:
{workspace_entries}

Fixed subscription rules for this Worker:
{subscription_rules}

You must select entries only if their metadata
    match the fixed subscription rules.
You must not select entries based on semantic
    judgment beyond the given rules.
You must not rewrite, merge, or reinterpret
    entries.
You must not create new entries.
You must not directly route information
    adaptively.

Always include:
1. Entries tagged with the Worker slot.
2. Entries tagged with "all".
3. Environment feedback entries allowed by the
    subscription rules.
4. Active unresolved dependencies involving the
    Worker slot.

Exclude:
1. Entries marked as obsolete, unless the
    subscription rules explicitly request
    obsolete entries.
2. Entries whose role tags and message types do
    not match the subscription rules.

Return only a valid JSON object in the following
     format:

{
  "selected_entry_ids": [
    "string"
  ],
  "selected_entries": [
    {
      "entry_id": "string",
      "producer_role": "string",
      "message_type": "string",
      "content": "string",
      "affected_interface": "string or none",
      "dependency_type": "string",
      "failure_category": "string",
      "role_tags": [
        "string"
      ],
      "status": "string"
```

```
    }
  ]
}
```

Listing 5: RCR-Router routed-memory Worker prompt prefix.

```
You are the {role} Worker in a fixed-role
    concurrent generation system.

You will receive routed memory entries selected
    for your role from a shared structured
    memory.
Use them to revise your role-specific output and
     maintain consistency with the other Worker.

[ROUTED MEMORY]
{routed_memory_entries}

Guidelines:
- Use the routed memory to address relevant
    feedback, errors, interfaces, and
    dependencies.
- Respect your assigned role.
- Do not directly produce the other Worker's
    output.
- Ensure your output remains compatible with the
     other Worker's role-specific output.
- If routed memory conflicts with evaluator
    feedback, prioritize evaluator feedback.

[TASK]
{task_description}

[ROLE INSTRUCTION]
{role_instruction}

[YOUR PREVIOUS OUTPUT]
{own_previous_output}

[LATEST EVALUATOR FEEDBACK]
{evaluator_feedback}

Now produce the next version of your role-
    specific output.
```

Listing 6: Single-agent frontier-model System Prompt

```
You are a careful CRM analyst solving lead
    qualification tasks.
Use the SQLite CRM database observations to
    decide whether the lead is qualified.
If the lead is not qualified, identify exactly
    which BANT factors fail:
Budget, Authority, Need, Timeline.

You can request one SQL query at a time.
Do not invent evidence not supported by the task
     or SQL observations.
The final answer must be a JSON list: ["
    Qualified"] or a canonical-order subset
of ["Budget", "Authority", "Need", "Timeline"].
```

The initial user message is instantiated from the following template:

Listing 7: Single-agent frontier-model Initial User Prompt

```
Task:
{task.query}

Task metadata:
{task_metadata}

Lead Id: {lead_id}

SQLite schema summary:
{schema_summary}

Important hints:
- The lead id in metadata is the lead to
    evaluate.
- Voice call transcripts are usually in
    VoiceCallTranscript__c with LeadId__c
  and Body__c.
- The Lead table gives the lead identity/title/
    status.
- Knowledge__kav may contain product or
    competitor context if the transcript
  mentions it.
- Use multiple SQL steps when needed; finalize
    as soon as you have enough
  evidence.
- If more evidence is needed, ask for another
    SQL query instead of guessing.
- Use at most {max_steps} SQL steps before
    finalizing.

Return only JSON. For a SQL step:
{
  "sql": "SELECT ... LIMIT 20",
  "rationale": "short reason"
}

For the final answer:
{
  "final_response": ["Qualified"],
  "reasoning": "brief evidence-based explanation
      "
}

If not qualified, use a non-empty canonical-
    order subset, for example:
{"final_response": ["Budget", "Timeline"], "
    reasoning": "..."}
```

After each model-generated SQL query, the evaluator executes the query against the local CRMArena-Pro SQLite database and appends a follow-up message:

Listing 8: Single-agent frontier-model Follow-Up Prompt

```
SQL execution result:
SQL:
{sql}

status: {status}
observation:
{observation}

Continue with one more SQL query if more
    evidence is needed, or finalize if the
evidence is sufficient.
Return only JSON in one of these forms:
```

```
{"sql": "SELECT ... LIMIT 20", "rationale":
    "..."}
{"final_response": ["Qualified"], "reasoning":
    "..."}
{"final_response": ["Budget", "Authority"], "
    reasoning": "..."}
```

If the model reaches the query budget without producing a final answer, the model is forced to generate a final decision with:

Listing 9: Single-agent frontier-model Final Decision Prompt

```
You have reached the SQL budget. Produce the
    final answer now.
Return only valid JSON using one of the
    following formats.
If the lead satisfies all BANT criteria, return:
{"final_response": ["Qualified"], "reasoning":
    "..."}
Otherwise, return:
{"final_response": ["<
    unmet_or_unverified_criterion>", "..."], "
    reasoning": "..."}
```

B MultiWebGen

We introduce MultiWebGen, a benchmark for evaluating multi-agent collaborative code generation in full-stack web development. Each task simulates a realistic development setting in which a backend agent and a frontend agent jointly implement a web application. Both agents are provided with a shared Product Requirements Document (PRD), agent-specific private observations, and executable test suites for evaluation. This setup enforces collaboration under partial information and enables evaluation of coordination, role specialization, and end-to-end functionality.

Source and Task Selection. MultiWebGen is constructed from the training split of WebGen-Bench (Lu et al., 2025). From this pool, we select 120 tasks using a two-stage process combining heuristic scoring and diverse selection. Tasks are filtered to 15 application categories (e.g., e-commerce, dashboards, CRM/ERP, social platforms) while excluding domains such as games, animations, or streaming that are less aligned with structured web workflows. Scoring prioritizes instructions containing CRUD operations, form interactions, and API/database concepts, while penalizing overly short or excessively long descriptions. To ensure diversity, tasks are grouped by category and selected in a round-robin manner before filling the remaining quota with top-scoring candidates. The final set is split into 80 training, 20 valid, and 20 test tasks using a fixed random seed.

Endpoint Extraction and Feature Construction. For each task, we use a LLM-based pipeline to extract a compact set of API endpoints that define a coherent mini-application centered around a primary resource. Each endpoint specifies its HTTP method, path (normalized to `/api/...`), description, resource schema, input fields (for POST/PUT). The model also produces a simplified instruction and a scenario identifier. This instruction is further rewritten into an implementation-ready feature description that removes framework-specific details, assumes a fixed stack (React + Express.js), and describes concrete data fields and user interactions. The result forms the core specification used by both agents. All extraction and rewriting stages use deterministic decoding to improve consistency across generated specifications.

PRD and Agent Observations. From the extracted endpoints and rewritten description, we generate three documents using deterministic templates. The PRD contains the feature description, endpoint specifications, and the fixed technical stack, intentionally describing what to build without prescribing how. The backend observation instructs the backend agent to implement an Express.js router with in-memory storage and proper HTTP semantics, outputting only route handler code. The frontend observation instructs the frontend agent to implement a React functional component using the fetch API, including UI elements and user feedback.

Test Generation and Refinement. Each task includes two executable test suites. Backend tests are written in Jest with supertest and verify endpoint existence, valid and invalid request handling, and appropriate status codes, without assuming strict response schemas. Frontend tests are written using React Testing Library with fetch mocking, verifying API calls triggered by rendering and user interactions. Importantly, rather than relying on LLM-as-a-judge for evaluation, we adopt programmatically executable tests to enable scalable reinforcement learning. LLM-based judging is prohibitively expensive and slow when used as a reward signal during RL training, especially in multi-agent settings. In contrast, our test-based evaluation provides a lightweight, deterministic, and fast reward signal that is suitable for large-scale training. While these tests do not enforce full end-to-end integration or strict semantic correctness, they are sufficient to ensure that agents

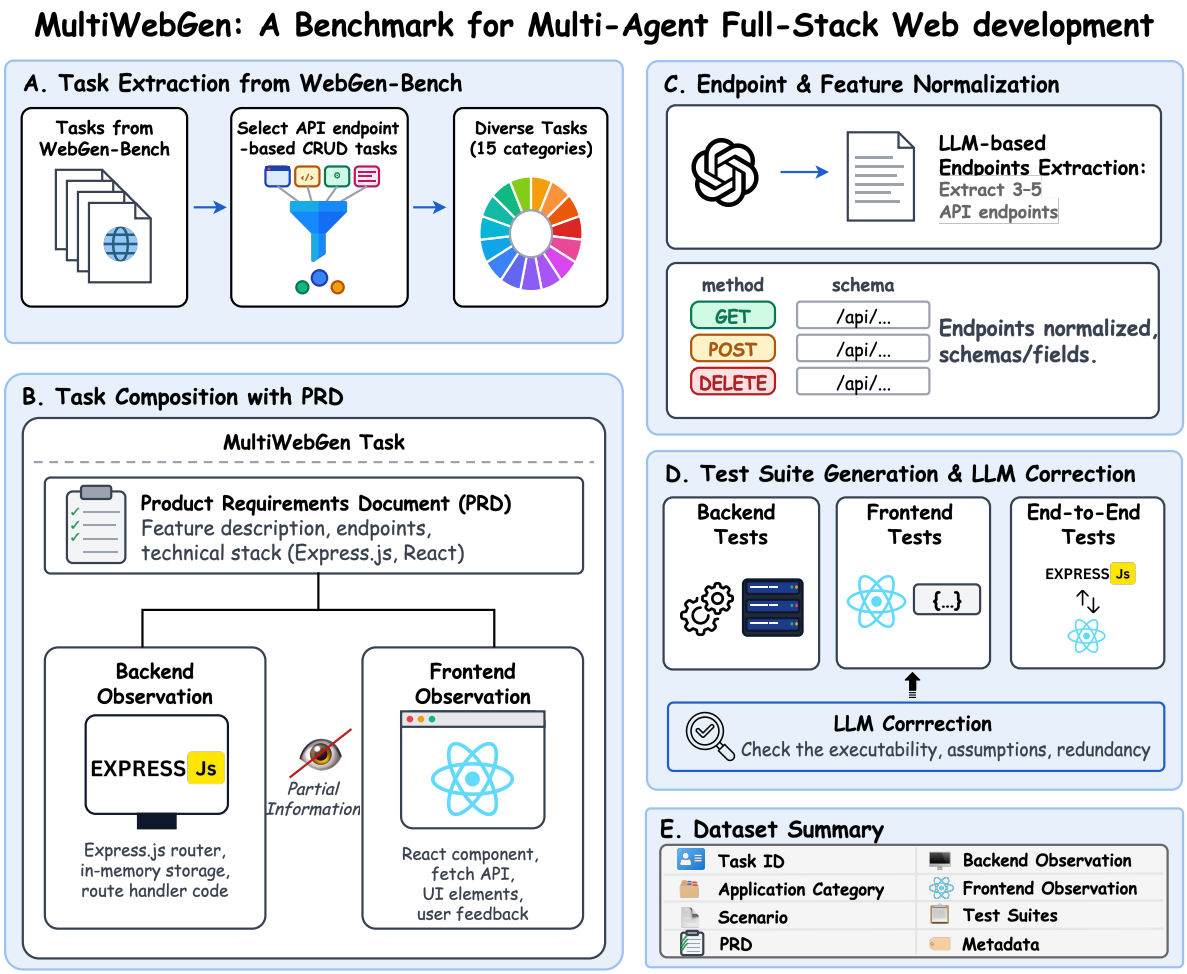

Figure 4: Overview of MultiWebGen.

align on shared API contracts and produce a minimally functional web application, which is the primary objective in collaborative code generation. To improve robustness, all tests undergo a LLM (GPT-5.2) fix pass that corrects unrealistic assumptions, ensures proper mocking, removes redundant checks, and aligns expectations with in-memory implementations.

Dataset Format and Statistics. Each record contains a unique task identifier, application category, scenario name, PRD, backend and frontend observations, corresponding test suites, endpoint definitions, and metadata. MultiWebGen consists of 120 tasks (80 train, 20 valid, 20 test), each containing 3–5 endpoints across 15 application categories. The dataset provides a controlled yet realistic environment for studying multi-agent collaboration in full-stack code generation. All task specifications are substantially rewritten through endpoint extraction, feature reconstruction, and agent-specific decomposition. The benchmark redistributes only newly generated specifications, prompts, and executable evaluation assets derived from publicly available source tasks. For reproducibility, the full benchmark and prompts accompany the paper. We will publicly release the dataset under a permissive research license.

Frontier Workers and Allocator Transfer. To calibrate MultiWebGen, we first evaluate frontier models under the *Independent* protocol. Each model uses a 4,096-token budget per Worker, top-p 1.0, temperature 0.0, three rounds, and previous-round conditioning; the independent results are averaged over three runs. We then reuse the MultiWebGen-trained Context Allocator checkpoint with each frontier Worker backbone, without further training either the Allocator or the Workers. This tests cross-backbone transfer on the same task distribution, rather than cross-task zero-shot generalization.

Worker model	w/o CA	w/ CA	Δ
GPT-5.5	48.4	57.3	+8.9
Claude-Opus-4.7	47.5	56.8	+9.3
Gemini-3.1-Pro	45.4	51.0	+5.6

Table 3: E2E Pass Rate (%) on MultiWebGen with and without the transferred Context Allocator.

C Analysis of Context Allocator Operations

We further analyze how the learned Context Allocator uses the context-editing operation space. This analysis complements the main results by examining whether the Context Allocator behaves as a static planner or as an active context manager over the course of multi-round collaboration.

C.1 Counting Protocol

For each evaluation episode, we parse the Context Allocator output between <OPS> and </OPS> and count each valid executable operation. Invalid operations, malformed JSON entries, and operations dropped by the reserved-key policy are excluded from the operation counts. Importantly, we do not count the automatic environment append step as a Context Allocator WRITE. After each Worker generation step, Worker outputs and executable feedback are appended to the context board by the environment update. These automatic additions are not Context Allocator-authored memory edits. However, if the Context Allocator later applies SCOPE or DELETE to environment-appended Worker-output or feedback slots, we count those operations because they are explicit Context Allocator decisions about what information should remain available and which Workers should see it.

C.2 Aggregate Operation Frequency

Table 4 reports aggregate operation frequencies on CoopHumanEval and MultiWebGen. The two benchmarks induce different Context Allocator behaviors. CoopHumanEval is centered on a compact function-level interface between a helper function and a main function. As a result, the Context Allocator often creates a small number of initial coordination slots and then performs limited revision. MultiWebGen, in contrast, requires backend and frontend Workers to maintain agreement over endpoint paths, HTTP methods, request bodies, response schemas, and state-update behavior across several interactions. This produces a moderately larger number of Context Allocator edits per episode, especially in repair turns where the Context Allocator must route feedback and revise shared interface assumptions.

The aggregate distribution shows that MultiWebGen requires more active context management than CoopHumanEval, but the increase is concentrated in revision and visibility-control operations rather than in additional initial planning. On MultiWebGen, UPDATE and SCOPE together account for more than sixty percent of all Context Allocator operations. This indicates that the Context Allocator frequently revises existing coordination state and adjusts role-specific visibility, rather than only writing new instructions.

The relative decrease of WRITE from CoopHumanEval to MultiWebGen is also informative. In CoopHumanEval, many episodes can be coordinated by writing a helper-side plan, a main-side plan, and a shared function contract. In MultiWebGen, the initial API contract is only the starting point: subsequent execution feedback often reveals mismatches between frontend assumptions and backend behavior, requiring the Context Allocator to update the shared contract, revise a role-specific plan, or scope specific feedback to the responsible Worker. Thus, the higher operation count on MultiWebGen reflects continued context management over the course of collaboration, not simply a larger initial plan.

C.3 Round-wise Operation Profile on MultiWebGen

To distinguish initial planning from later context management, we further break down MultiWebGen operations by Context Allocator turn. Table 5 shows that WRITE operations are concentrated in the initial turn, where the Context Allocator creates the backend plan, frontend plan, and shared API contract. These initial writes define the division of labor and the interface assumptions that both Workers should follow.

In the repair turns, the Context Allocator's behavior shifts from creating new slots to maintaining existing coordination state. After Worker outputs and executable feedback are appended to the context board, the Context Allocator mainly uses UPDATE to revise role-specific plans or interface contracts, and SCOPE to route feedback to the Workers that need it. For example, frontend rendering or missing-field failures can be scoped to the frontend Worker, while endpoint, status-code, or request-body fail-

Benchmark	Valid Ops.	Ops./Episode	WRITE	UPDATE	DELETE	SCOPE
CoopHumanEval	98.7%	7.1 ± 1.9	3.6 (50.7%)	1.5 (21.1%)	0.5 (7.0%)	1.5 (21.1%)
MultiWebGen	97.9%	11.4 ± 3.2	3.8 (33.3%)	3.2 (28.1%)	0.5 (4.4%)	3.9 (34.2%)

Table 4: Frequency of Context Allocator context-editing operations in DCA. We count valid executable operations emitted during evaluation. Ops./Episode denotes the average number of Context Allocator operations per task episode. Percentages are computed within each benchmark.

Context Allocator Turn	Ops./Episode	WRITE	UPDATE	DELETE	SCOPE
Round 1	3.1	3.1 (100.0%)	0.0 (0.0%)	0.0 (0.0%)	0.0 (0.0%)
Round 2	6.0	0.6 (10.0%)	2.1 (35.0%)	0.3 (5.0%)	3.0 (50.0%)
Round 3	2.3	0.1 (4.3%)	1.1 (47.8%)	0.2 (8.7%)	0.9 (39.1%)

Table 5: Round-wise operation profile on MultiWebGen.

ures can be scoped to the backend Worker or to both Workers when they indicate an interface mismatch.

This round-wise pattern suggests that DCA does not merely act as an initial planning module. The small number of later `WRITE` operations correspond to genuinely new coordination slots, such as backend-status or unresolved-issue notes, while `DELETE` is used conservatively to remove stale or interfering information. Overall, the MultiWebGen profile shows that the learned Context Allocator performs active repair-time context management: it updates task state, controls role-specific visibility, and selectively prunes the buffer as the joint solution evolves.

D DCA with Frozen Workers

To isolate the effect of learning context allocation from the effect of improving Worker policies, we evaluate a frozen-Worker variant of DCA on CoopHumanEval and MultiWebGen. In this setting, the Worker models are kept fixed throughout training, and only the Context Allocator is optimized with reinforcement learning from shared task rewards. The Context Allocator still observes the Context Board, applies the same symbolic context-editing operations, and constructs role-specific Worker views, but the Workers themselves are not updated.

Table 6 reports mean and standard deviation over five independent runs. The frozen-Worker results show that learning context allocation alone provides substantial gains, but does not fully explain DCA's performance. On CoopHumanEval, frozen-Worker DCA improves over fixed-protocol baselines but remains below fixed-context RL baselines, suggesting that Worker policy learning is still important for function-level code generation. On MultiWebGen, frozen-Worker DCA surpasses worker-only RL baselines, indicating that cross-role observation control is especially valuable when backend and frontend implementations must maintain consistent API contracts. The remaining gap to full DCA shows that context allocation and Worker policy learning are complementary.

Method	**Coop HumanEval**		**Multi WebGen**	
	Mean	Std.	Mean	Std.
Independent	48.8	2.8	20.2	0.5
RCR-Router	63.8	2.8	29.5	1.3
MAGRPO	72.5	3.4	35.5	0.6
CoLLM-CC	73.8	2.8	32.4	0.9
DCA w/ Frozen Workers	67.5	2.8	40.7	1.4
DCA	80.0	2.8	51.7	0.5

Table 6: Performance of DCA with frozen Workers on CoopHumanEval and MultiWebGen. Only the Context Allocator is trained, while Worker models are kept fixed. CoopHumanEval reports Test Pass Rate (%), and MultiWebGen reports E2E Pass Rate (%).

E Detailed Experimental Results

This section reports the full numerical results of CoopHumanEval, MultiWebGen, and CRM lead qualification, including mean performance and standard deviation across five independent runs, to provide statistical support for the comparisons in the main text.

Method	CoopHumanEval		MultiWebGen	
	Mean	Std.	Mean	Std.
Independent	48.8	2.8	20.2	0.5
Self-Refine	53.8	3.4	21.9	0.4
Self-Consistency	50.0	4.4	24.3	2.2
MAD	55.0	2.8	27.9	0.8
AutoGen	62.5	0.0	24.6	0.6
MetaGPT	61.3	2.8	27.4	0.6
RCR-Router	63.8	2.8	29.5	1.3
MAGRPO	72.5	3.4	35.5	0.6
CoLLM-CC	73.8	2.8	32.4	0.9
DCA	80.0	2.8	51.7	0.5

Table 7: Detailed results on CoopHumanEval and MultiWebGen. CoopHumanEval reports Test Pass Rate (%), and MultiWebGen reports E2E Pass Rate (%).

Model	Mean	Std.
Llama-3.1-70B	13.0	2.7
Claude-Sonnet-4.5	33.0	2.7
Claude-Opus-4.5	41.0	2.2
Claude-Opus-4.7	44.0	2.2
Gemini-3-Flash	32.0	2.7
Gemini-3.1-Pro	43.0	2.7
GPT-5-mini	28.0	2.7
GPT-5.2	39.0	4.2
GPT-5.5	46.0	2.2
Qwen3-4B-Inst (4W)	15.0	3.5
Qwen3-8B (4W)	18.0	2.7
MAD-4B (4W)	17.0	2.7
AutoGen-4B (4W)	21.0	2.2
MetaGPT-4B (4W)	20.0	2.2
RCR-Router-4B (4W)	21.0	2.2
MAGRPO-4B (4W)	26.0	2.2
MAGRPO-8B (4W)	30.0	3.5
DCA-4B (4W+1CA)	49.0	2.2
DCA-8B (4W+1CA)	54.0	2.2

Table 8: Detailed results on CRM lead qualification. We report Exact Match Rate (%). Qwen3-4B-Inst (4W) and Qwen3-8B (4W) denote four-Worker baselines using the corresponding base models.

F Additional Experiments

F.1 Memory-R1-Style Multi-Agent Adaptation

Memory-R1 (Yan et al., 2026) was designed for memory-augmented question answering with a Memory Manager and one downstream Answer Agent. To adapt its allocation pattern to MultiWebGen, one Memory Manager edits a shared bank with Memory-R1's operation set, but does not construct role-specific views before Worker execution. Each Worker instead receives the full bank and uses its Worker policy both to select role-relevant entries and to generate its benchmark-defined output. This comparison directly evaluates DCA's dedicated per-Worker view construction against answer-side filtering; Section F reports the result.

Method	**E2E Pass Rate (%)**
MAGRPO	35.5
Memory-R1-style adaptation	43.4
DCA	**51.7**

Table 9: Comparison with a Memory-R1-style adaptation on MultiWebGen.

Model	**Original**	**Role-decomposed**
Gemini-3.1-Pro	43.0	45.0
Claude-Opus-4.7	44.0	48.0
GPT-5.5	46.0	49.0

Table 10: Exact Match Rate (%) on CRM lead qualification. Role decomposition changes the single-agent prompt, not its SQL budget or number of model instances.

The adaptation exceeds fixed-context MAGRPO on MultiWebGen, showing that learned memory management is useful in this setting. DCA is 8.3 E2E points higher than the adaptation (Table 9); the comparison evaluates this architectural alternative but does not identify which implementation difference causes the gap.

F.2 Role-Decomposed Single-Agent CRM Baselines

The original frontier baseline lets one model retrieve evidence and directly predict the final label. The role-decomposed variant instead instructs the same model to solve the four BANT criteria explicitly and aggregate them, while retaining the same SQL budget and single-model tool-use protocol. Table 10 shows that this stronger prompt improves all three frontier models. The best result, GPT-5.5 at 49.0%, matches DCA-4B but remains 5.0 points below DCA-8B.

G Training Curves

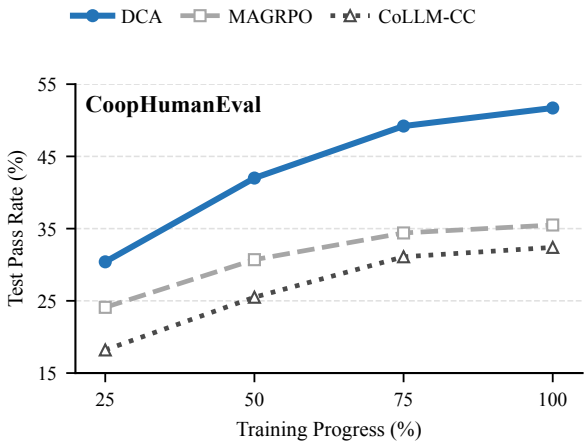

Figure 5: Training curves on CoopHumanEval for DCA and the fixed-context RL baselines.

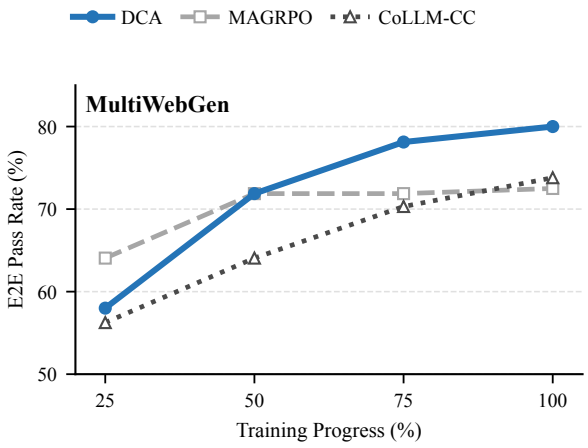

Figure 6: Training curves on MultiWebGen for DCA and the fixed-context RL baselines.

H Latency Comparison

We additionally measure end-to-end wall-clock latency on a test split of CRM lead qualification. This analysis focuses on whether concurrent role execution reduces latency relative to a sequential BANT decomposition under the same per-call model size. Each example requires a BANT decision over four criteria.

To avoid confounding latency with early stopping, all systems use a fixed three-round inference protocol. No system is allowed to emit a final BANT prediction before the end of round 3. Intermediate outputs from rounds 1 and 2 are treated only as internal evidence and may be carried forward to later rounds, but they are not normalized, evaluated, or used as an early stopping condition.

System	Schedule	Gen.	Mean (s)
Sequential	Serial Workers	24	11.80
Concurrent	Concurrent Workers	24	5.05
DCA	Context Allocator + Concurrent Workers	27	8.50

Table 11: Latency on CRM lead qualification. "Gen." denotes the number of LLM generations per example.

System	Round 1 (s)	Round 2 (s)	Round 3 (s)	Total (s)
Sequential	3.80	3.90	4.10	11.80
Concurrent	1.45	1.51	2.09	5.05
DCA	3.20	2.44	2.86	8.50

Table 12: Mean per-round wall-clock latency under the fixed three-round protocol.

Timing starts when the system receives the lead-qualification input and stops only after the final round-3 BANT prediction has been produced.

All systems use Qwen3-4B-Instruct-2507 served with the same `vLLM` backend on the same B200 hardware. Each criterion-level Worker call consists of two model generations: first a SQL proposal, then a criterion decision after SQL execution.

We compare three fixed-round schedules. *Sequential* uses one active model stream and evaluates the four BANT criteria serially in each round, in the fixed order Budget–Authority–Need–Timeline. To keep the observation protocol comparable to concurrent Workers, outputs produced earlier within the same round are not exposed to later criterion calls until the next round. *Concurrent* removes the Context Allocator and executes the four role-specific BANT Workers in parallel in each round. This schedule is the CRM lead qualification analogue of the *Independent* baseline used in CoopHumanEval and MultiWebGen: each Worker follows its own role, history, and feedback, without learned context allocation or access to other Workers' same-round outputs. DCA uses the same four Workers plus one Context Allocator. At each round, the Context Allocator first edits and scopes the context board, after which the four Workers act concurrently from their role-specific views.

The reported wall-clock time includes model decoding, SQL tool calls, Context Allocator generation when applicable, context-buffer operations when applicable, and deterministic final aggregation. It excludes offline costs such as model loading, training, and data preprocessing.

Table 11 shows that concurrent Worker execution reduces latency relative to serializing the same BANT decomposition with the same model size. Concurrent execution reduces mean latency from

11.80 to 5.05 seconds per task. DCA adds one Context Allocator generation per round, raising latency by 3.45 seconds over concurrent execution, but remains 3.30 seconds faster than serial role execution. Table 12 shows the same pattern at the round level: both concurrent schedules shorten the serial BANT critical path, while DCA trades part of this latency gain for learned context allocation.

I Prompts

I.1 CoopHumanEval

Listing 10: Context Allocator System prompt

```
You are the Controller of a collaborative coding
     team. You coordinate workers by editing a
    board with JSON ops.
```

Listing 11: Context Allocator initial instruction prompt

```
You are the Controller for a collaborative
    coding task.
Two workers use a Controller-maintained board:
  - Worker 0 writes a helper function named `aux
      ()`.
  - Worker 1 writes the main function `{
      entry_point}` which must call aux().

Problem:
{problem}

Initial context board (read-only env slots,
    visible to both workers):
{initial_context_board}

Your job is to seed the board with coordination
    slots so that each worker only sees the
    information it actually needs to solve its
    part.
You do NOT modify reserved environment slots --
    your only authoring is via WRITE on
    coordination slots you create yourself.
Output an ordered list of edit operations inside
     <OPS>...</OPS> as JSON.

Available ops: WRITE, UPDATE, DELETE, SCOPE.

Operation semantics:
- WRITE(k, v, vis): create a NEW coordination
    slot with key k, value v, and visibility vis
    .
- UPDATE(k, v): replace the value of an existing
     coordination slot with v, preserving its
    visibility.
- DELETE(k): remove a coordination slot you
    previously wrote.
- SCOPE(k, vis'): change the visibility list of
    slot k to vis'. The value is preserved
    verbatim.

Reserved-key policy:
- WRITE / UPDATE on `problem`, `entry_point`,`
    worker_*_code_t*`, `feedback_*` are dropped.
- DELETE on `problem` / `entry_point` is dropped.
- DELETE on `worker_*_code_t*` / `feedback_t*`
    IS allowed.
- SCOPE on reserved keys is allowed.

Rules:
- `vis` must be a list of ints in [0, 1]. vis
    =[0] = Worker 0 only, vis=[1] = Worker 1
    only, vis=[0,1] = shared, vis=[] = hide.
- Pick visibility deliberately.
- WRITE values must be plain natural-language
    coordination notes -- never emit code.

Example:
<OPS>
[
{"op":"WRITE","key":"contract","value":"...","
    vis":[0,1],}
{"op":"WRITE","key":"aux_focus","value":"...","
    vis":[0],}
{"op":"WRITE","key":"main_focus","value":"...","
    vis":[1]}
]
</OPS>
```

Listing 12: Context Allocator follow-up instruction prompt (round > 0)

```
You are the Controller for a collaborative
    coding task. Two workers
use a Controller-maintained context board:
  - Worker 0 writes a helper function named `aux
      ()`.
  - Worker 1 writes the main function `{
      entry_point}` which must call aux().

Original problem:
{problem}

Current board:
{rendered_board}

Output an ordered list of edit operations inside
     <OPS>...</OPS> as JSON.

Available ops: WRITE, UPDATE, DELETE, SCOPE.

Operation semantics:
- WRITE(k, v, vis): create a NEW coordination
    slot with key k, value v (natural-language
    plan / contract / hint), and visibility vis.
- UPDATE(k, v): replace the value of an existing
     coordination slot with v, preserving its
    visibility.
- DELETE(k): remove a coordination slot you
    previously wrote.
- SCOPE(k, vis'): change the visibility list of
    slot k to vis'. The value is preserved
    verbatim.

Reserved-key policy:
- WRITE / UPDATE on `problem`, `entry_point` are
     dropped.
- DELETE on `problem` / `entry_point` is dropped.

- DELETE on `worker_*_code_t*` / `feedback_t*`
    IS allowed.
- SCOPE on reserved keys is allowed.

Rules:
```

```
- `vis` must be a list of ints in [0, 1]. vis
    =[0] = Worker 0 only, vis=[1] = Worker 1
    only, vis=[0,1] = shared, vis=[] = hide.
- Pick visibility deliberately.
- WRITE / UPDATE values must be plain natural-
    language coordination notes -- never emit
    code.
- Use UPDATE to tighten an existing slot without
     changing its audience; use SCOPE to change
    only the audience.
- Keep total slot count below {max_slots}.
- Keep total slot count below {max_slots}.

Example:
<OPS>
[
  {"op":"UPDATE","key":"contract","value":"..."}
  {"op":"UPDATE","key":"aux_focus","value
      ":"..."},
]
</OPS>
```

Listing 13: Worker-0 aux function instruction prompt

```
Create a helper function for this coding problem.


Problem:
{problem}

Coordination instructions:
{filtered_context_view}

IMPORTANT INSTRUCTIONS:
- Output ONLY the function code, no explanations
     or examples
- Do NOT include markdown code blocks
- Do NOT include any text before or after the
    function
- Do NOT include test cases or example usage
- Create a helper function named 'aux' that can
    assist the main function
- The function should return useful data for
    solving the problem

Your output should follow this format:

def aux(...):
 # your function code here
 return result
```

Listing 14: Worker-1 main function instruction prompt

```
Solve this coding problem by implementing the
    required function.

Problem:
{problem}

Coordination instructions:
{filtered_context_view}

You have access to a helper function: aux(...)

IMPORTANT INSTRUCTIONS:
- Output ONLY the function code, no explanations
     or examples
- Do NOT include markdown code blocks
- Do NOT include any text before or after the
    function
- Do NOT include test cases or example usage
- Implement ONLY the '{entry_point}' function as
     specified
- You MUST call aux() to assign value to a
    variable within your function
- Do NOT redefine the aux() function

Your output should follow this format:

def {entry_point}({params}):
 # your function code with aux() call here
 return result
```

I.2 MultiWebGen

Listing 15: Context Allocator System prompt

```
You are the Controller of a collaborative full-
    stack web development team. You coordinate
    workers by editing a board with JSON ops.
```

Listing 16: Context Allocator initial instruction prompt

```
Two workers use a Controller-maintained board
    and will build a full-stack web application
    together:
  - Worker 0 (Backend) writes Express.js route
      handlers.
  - Worker 1 (Frontend) writes React components
      that call the backend via fetch().

Instruciton:
{simplified_instruction}

Product Requirements:
{prd}

Initial context board (read-only env slots,
    visible to both workers):
{initial_context_board}

Your job is to seed the board with coordination
    slots so that each worker only sees the
    information it actually needs to solve its
    part.
You do NOT modify reserved environment slots --
    your only authoring is via WRITE on
    coordination slots you create yourself.
Output an ordered list of edit operations inside
     <OPS>...</OPS> as JSON.

Available ops: WRITE, UPDATE, DELETE, SCOPE.

Operation semantics:
- WRITE(k, v, vis): create a NEW coordination
    slot with key k, value v (natural-language
    plan / API contract / hint), and visibility
    vis.
- UPDATE(k, v): replace the value of an existing
     coordination slot with v, preserving its
    visibility.
- DELETE(k): remove a coordination slot you
    previously wrote.
- SCOPE(k, vis'): change the visibility list of
    slot k to vis'. The value is preserved
    verbatim. This is the channel for exposing
    reserved slots across the backend/frontend
    boundary.
```

```
Reserved-key policy:
- WRITE / UPDATE on `instruction`, `prd`, `
    worker_*_code_t*`, `feedback_*` are dropped.
- DELETE on `instruction` / `prd` is dropped.
- DELETE on `worker_*_code_t*` / `feedback_t*`
    IS allowed.
- SCOPE on reserved keys IS allowed.

Rules:
- `vis` must be a list of ints in [0, 1]. vis
    =[0] = backend only, vis=[1] = frontend only
    , vis=[0,1] = shared, vis=[] = hide.
- WRITE values must be plain natural-language
    coordination notes, never JavaScript or JSX.

Example:
<OPS>
[
{"op":"WRITE","key":"contract","value":"...","
    vis":[0,1]}
{"op":"WRITE","key":"backend_focus","value
    ":"...","vis":[0]}
{"op":"WRITE","key":"frontend_focus","value
    ":"...","vis":[1]}
]
</OPS>
```

Listing 17: Context Allocator follow-up instruction prompt (round > 0)

```
You are the Controller for a collaborative full-
    stack web development task. Two workers use
    a Controller-maintained board:
  - Worker 0 writes Express.js route handlers (
      the backend).
  - Worker 1 writes React components (the
      frontend) that call the backend via fetch
      ().

Instruciton:
{simplified_instruction}

Product Requirements:
{prd}

Current board:
{rendered_board}

Output an ordered list of edit operations inside
     <OPS>...</OPS> as JSON.

Available ops: WRITE, UPDATE, DELETE, SCOPE.

Rules:
- `vis` must be a list of ints in [0, 1]. vis
    =[0] = backend only, vis=[1] = frontend only
    , vis=[0,1] = shared, vis=[] = hide.
- Pick visibility deliberately.
- Use UPDATE to tighten an existing slot without
     changing its audience; use SCOPE to change
    only the audience.
- WRITE / UPDATE values must be plain natural-
    language coordination notes, never
    JavaScript, JSX, or any other code.
- Keep total slot count below {max_slots}.

Example:
<OPS>
[
{"op":"UPDATE","key":"contract","value":"..."}
{"op":"SCOPE","key":"feedback_t1","vis":[0]}
]
```

Listing 18: Worker System prompt

```
You are an expert full-stack JavaScript
    developer. Write clean, correct code for
    Express.js backends and React frontends.
    Follow REST conventions and ensure API
    contracts between backend routes and
    frontend fetch calls are consistent. Return
    ONLY the implementation code without
    explanations or markdown formatting.
```

Listing 19: Worker-0 backend instruction prompt

```
Implement an Express.js backend for a web
    application.

Product Requirements:
{prd}

Your specific task:
{backend_observation}

{filtered_context_view}

IMPORTANT INSTRUCTIONS:
- Output ONLY the Express.js route handler code,
     no explanations
- Do NOT include markdown code blocks
- Do NOT include any text before or after the
    code
- Implement all required API endpoints as
    Express.js router handlers
- Use EXACT endpoint paths and HTTP methods from
     the provided task context
- Start with: const express = require('express')
    ; const router = express.Router();
- End with: module.exports = router;
```

Listing 20: Worker-1 frontend instruction prompt

```
Implement a React frontend component for a web
    application.

Product Requirements:
{prd}

Your specific task:
{frontend_observation}

Coordination instructions:
{filtered_context_view}

IMPORTANT INSTRUCTIONS:
- Output ONLY the React component code, no
    explanations
- Do NOT include markdown code blocks
- Do NOT include any text before or after the
    code
- Use fetch() calls to communicate with the
    backend API
- Match fetch endpoint paths and HTTP methods
    EXACTLY to the backend contract
- End with: export default ComponentName;
```

I.3 CRM Lead Qualification

Listing 21: Context Allocator System prompt

```
You are the Controller for a CRM lead-
    qualification team. You coordinate workers
    by editing a board with JSON ops.
```

Listing 22: Context Allocator instruction prompt

```
You are the Controller / Context Allocator for a
     CRM lead-qualification task.

Workers and visibility ids:
- 0 = budget
- 1 = authority
- 2 = need
- 3 = timeline

Task query:
{task_query}

Task metadata:
{task_metadata}

Task Persona:
{task_persona}

Current context board:
{full_context_buffer}

Current round: {round_idx} of {max_rounds}
{terminal_rule}

Decision ownership:
- The Controller allocates context and routes
    information across workers.
- Each worker writes SQL, observes the query
    result, summarizes role-specific evidence,
    and makes the decision for its own BANT
    criterion.
- The final lead-qualification answer is derived
     from worker decisions.

Output only edit operations inside <OPS>...</OPS
    > as JSON.
Available ops: WRITE, UPDATE, DELETE, SCOPE.

Operation schema:
- WRITE(k, v, vis): create a NEW coordination
    slot with key k, value v (natural-language
    plan / contract / hint), and visibility vis.
- UPDATE(k, v): replace the value of an existing
     coordination slot with v, preserving its
    visibility.
- DELETE(k): remove a coordination slot you
    previously wrote.
- SCOPE(k, vis'): change the visibility list of
    slot k to vis'. The value is preserved
    verbatim.

Rules:
- Use worker visibility ids deliberately. Budget
    =0, Authority=1, Need=2, Timeline=3.
- WRITE/UPDATE values should be concise natural-
    language coordination notes, not SQL,
    evidence judgments, or BANT decisions.
- You may SCOPE worker_* and feedback_* slots to
     expose exact prior evidence to specific
    workers.
- Do not write final_response or failed BANT
    labels; workers are responsible for role-
    specific decisions.
- SQL errors, empty observations, and
    missing_information are uncertainty, not
    failure evidence. Use context allocation to
    help workers gather better evidence.

Example:
<OPS>
[
  {"op":"WRITE","key":"budget_focus","value
      ":"...","vis":[0]},
  {"op":"WRITE","key":"authority_focus","value
      ":"...","vis":[1]},
  {"op":"WRITE","key":"need_focus","value
      ":"...","vis":[2]},
  {"op":"WRITE","key":"timeline_focus","value
      ":"...","vis":[3]}
]
</OPS>
```

Listing 23: Context Allocator follow-up instruction prompt (round > 0)

```
You are the Controller / Context Allocator for a
     CRM lead-qualification task.

Workers and visibility ids:
- 0 = budget
- 1 = authority
- 2 = need
- 3 = timeline

Task query:
{task_query}

Task metadata:
{task_metadata}

Task Persona:
{task_persona}

Current context board:
{full_context_buffer}

Current round: {round_idx} of {max_rounds}
{terminal_rule}

Decision ownership:
- The Controller allocates context and routes
    information across workers.
- Each worker writes SQL, observes the query
    result, summarizes role-specific evidence,
    and makes the decision for its own BANT
    criterion.
- The final lead-qualification answer is derived
     from worker decisions.

Output only edit operations inside <OPS>...</OPS
    > as JSON.
Available ops: WRITE, UPDATE, DELETE, SCOPE.

Operation schema:
- WRITE(k, v, vis): create a NEW coordination
    slot with key k, value v, and visibility vis
    .
- UPDATE(k, v): replace the value of an existing
     coordination slot with v, preserving its
    visibility.
```

```
- DELETE(k): remove a coordination slot you
    previously wrote.
- SCOPE(k, vis'): change the visibility list of
    slot k to vis'. The value is preserved
    verbatim.

Reserved-key policy:
- WRITE / UPDATE on `task_query`, `task_metadata
    `,`task_persona`, `worker_*_code_t*`, `
    feedback_*` slots are dropped.
- DELETE on `task_query`, `task_metadata`, and `
    task_persona` is dropped.
- DELETE on `worker_*_code_t*` / `feedback_*`
    slots IS allowed.
- SCOPE on reserved keys IS allowed.

Rules:
- Use worker visibility ids deliberately. Budget
    =0, Authority=1, Need=2, Timeline=3.
- Pick visibility deliberately.
- Use UPDATE to tighten an existing coordination
     slot without changing its audience; use
    SCOPE to change only the audience.
- Use SCOPE to expose exact prior worker
    evidence, SQL results, or feedback only to
    the workers that need it.
- WRITE / UPDATE values must be concise natural-
    language coordination notes, not SQL,
    evidence judgments, BANT decisions,
    final_response values, or failed BANT labels
    .
- Do not decide whether the lead is qualified;
    workers are responsible for role-specific
    decisions.
- SQL errors, empty observations, and
    missing_information are uncertainty, not
    failure evidence. Use context allocation to
    help workers gather better evidence.
- Keep total slot count below {max_slots}.

Example:
<OPS>
[
  {"op":"SCOPE","key":"worker_budget_summary_t0
      ","vis":[0]},
  {"op":"UPDATE","key":"authority_focus","value
      ":"..."}
]
</OPS>
```

Listing 24: Worker System prompt

```
You are a role-specific CRM BANT evidence worker.
     Query only columns shown in schema_summary,
     avoid hallucinated Salesforce fields,
    summarize only evidence for your assigned
    criterion, make a decision for that
    criterion from the SQL result, and return
    the requested JSON exactly.
```

Listing 25: Worker role instruction blocks

```
Budget worker:
Find evidence about whether the lead has budget.

Authority worker:
Find evidence about whether the contact has
    buying authority.

Need worker:
Find evidence about whether the lead has a clear
     business need or pain point that the
    product can address.

Timeline worker:
Find evidence about whether the lead has a clear
     buying, evaluation, or deployment timeline.
```

Listing 26: Worker SQL instruction prompt

```
You are a role-specific evidence collector for a
     CRM lead qualification task.
In this SQL step, you do not decide whether the
    lead is qualified.
In this SQL step, you do not output BANT failure
     labels.
Your job is to write a SQLite query that
    retrieves evidence relevant to your assigned
     criterion.

Use only tables and columns that appear in the
    visible schema_summary slot.
Do not invent Salesforce fields. If a useful
    column is absent, query the available
    transcript/lead/opportunity fields instead
    of guessing a column name.

Role Instruction:
{role_instruction}

Visible context:
{visible_context}

Return only one compact JSON object, on one line,
     with no markdown and no extra keys:
{
  "sql": "SELECT ...;"
}
```

Listing 27: Worker evidence-summary instruction prompt

```
You are a role-specific evidence collector for a
     CRM lead qualification task.
You decide only whether your assigned BANT
    criterion is satisfied, failed, or uncertain
    .
You do not decide the full lead qualification
    answer.
Your job is to summarize the SQL result and make
     a role-specific decision from retrieved
    evidence.

Role Instruciton:
{role_instruction}

Visible context:
{visible_context}

SQL you executed:
{sql_text}

DB status: {db_status}
DB observation:
{db_observation}

Summarize only role-specific evidence. Decide
    only your assigned criterion, not the final
    answer.
```

```
If DB status is 0, or the observation is an
    error/empty because the SQL failed, set
    decision to "uncertain"; do not treat query
    failure as proof that the criterion failed.
Set decision to "failed" only when retrieved
    records contain explicit negative evidence
    for this criterion.
Set decision to "satisfied" only when retrieved
    records contain explicit positive evidence
    for this criterion and no explicit blocker.
Return only JSON with keys: role, criterion,
    evidence_summary, supporting_records,
    decision, confidence.
```

J Qualitative Results

We present qualitative examples using the actual outputs generated by the models in our experiments, showing the first two rounds for readability. In the Context Allocator outputs, the fields `op`, `key`, `vis`, and `value` correspond to the operation, κ, vis, and v introduced in the method section, respectively.

J.1 CoopHumanEval

Task `calculate_rectangle_area(dimensions)`

Listing 28: Round 1: Context Allocator output

```
<OPS>
[
{"op":"WRITE","key":"contract","value":"aux(
    dimensions: list[int]) -> bool validates
    whether the input satisfies the task
    constraints. calculate_rectangle_area(
    dimensions) is the main function that
    returns the final area.","vis":[0,1]}
{"op":"WRITE","key":"aux_focus","value":"
    Implement helper function aux(dimensions) to
     validate the input list and return True for
     valid inputs, otherwise False.","vis":[0]},
{"op":"WRITE","key":"main_focus","value":"
    Implement calculate_rectangle_area(
    dimensions). Use aux(dimensions) for
    validation, then compute square or rectangle
     area.","vis":[1]},
]
</OPS>
```

Listing 29: Round 1: Worker-0 aux output

```
def aux(dimensions):
    if not dimensions or len(dimensions) > 2:
        return False
    return all(isinstance(x, (int, float)) and x
        > 0 for x in dimensions)
```

Listing 30: Round 1: Worker-1 main output

```
def calculate_rectangle_area(dimensions):
    if not aux(dimensions):
        return -1
    if len(dimensions) == 1:
        return dimensions[0] ** 2
    return dimensions[0] * dimensions[1]
```

Listing 31: Round 2: Context Allocator output

```
<OPS>
[
{"op":"DELETE","key":"worker_0_code_t1"},
{"op":"UPDATE","key":"contract","value":"aux(
    dimensions: list[int]) -> bool validates
    whether the input is a list of 1 or 2
    positive integers only.
    calculate_rectangle_area(dimensions) must
    call aux(dimensions); if valid, it computes
    and returns the geometric area, otherwise
    returns -1."},
{"op":"UPDATE","key":"aux_focus","value":"Revise
     aux(dimensions) to strictly enforce the
    specification: accept only Python lists
    containing 1 or 2 positive integers, and
    reject empty lists, lists longer than 2, non
    -integers, and non-positive values."},
]
</OPS>
```

Listing 32: Round 2: Worker-0 aux output

```
def aux(dimensions):
    if not isinstance(dimensions, list):
        return False
    if not dimensions or len(dimensions) > 2:
        return False
    return all(isinstance(x, int) and x > 0 for x
         in dimensions)
```

Listing 33: Round 2: Worker-1 main output

```
def calculate_rectangle_area(dimensions):
    if not aux(dimensions):
        return -1
    if len(dimensions) == 1:
        side = dimensions[0]
        return side * side
    length, width = dimensions
    return length * width
```

J.2 MultiWebGen

Task `manageProductionWorkOrders`

Express.js backend with five endpoints (`POST /api/users/login`, `POST/GET/PUT/DELETE /api/work-orders`) and a React frontend that consumes them. Worker-0 implements the backend; Worker-1 implements the frontend.

Listing 34: Round 1: Context Allocator output

```
<OPS>
[
{"op":"WRITE","key":"contract","value":"Wire
    format: every work-order record is a JSON
    object {id:number, orderNumber, productCode,
     quantity, dueDate, priority, status}. POST/
    PUT responses MUST echo all submitted fields
     verbatim in the top-level body (not nested)
    . GET /api/work-orders returns a top-level
    JSON array of such records. Empty POST/PUT
    body -> 400 JSON {error:string}. Frontend
    renders every field of every record as
    visible text so the response payload is
    observable in the DOM.","vis":[0,1]}
```

```
{"op":"WRITE","key":"backend_focus","value":"
    Implement an Express router exposing all
    five endpoints. Use an in-memory array `
    workOrders` keyed by an auto-incrementing `
    id`. POST /api/work-orders must echo every
    submitted field back in the response body so
     the frontend can render it without a second
     GET. Reject empty bodies on POST/PUT with
    400. GET must always return 200 with a JSON
    array (empty if none).","vis":[0]},
{"op":"WRITE","key":"frontend_focus","value":"
    Implement a single React component with
    three sections: a login form, a work-order
    creation form, and a work-order list. On
    mount, GET /api/work-orders and render every
     returned record's fields directly into the
    DOM (id, orderNumber, productCode, quantity,
     dueDate, status, priority). Provide a Login
     button that POSTs to /api/users/login, and
    a Create button that POSTs to /api/work-
    orders. After a successful create, append
    the echoed record to the list state so the
    UI updates without refetching.","vis":[1]},
]
</OPS>
```

Listing 35: Round 1: Worker-0 backend output

```
const express = require('express');
const router = express.Router();

let workOrders = [];
let nextId = 1;

router.post('/api/users/login', (req, res) => {
  const { email, password } = req.body || {};
  if (!email || !password) {
    return res.status(400).json({ error: 'email
        and password required' });
  }
  res.status(200).json({ email, token: 'session_
      ' + Date.now() });
});

router.post('/api/work-orders', (req, res) => {
  const body = req.body || {};
  if (Object.keys(body).length === 0) {
    return res.status(400).json({ error: 'empty
        body' });
  }
  const record = { id: nextId++, status: '
      pending', ...body };
  workOrders.push(record);
  res.status(201).json(record);
});

router.get('/api/work-orders', (req, res) => {
  res.status(200).json(workOrders);
});

router.put('/api/work-orders', (req, res) => {
  const body = req.body || {};
  if (Object.keys(body).length === 0) {
    return res.status(400).json({ error: 'empty
        body' });
  }
  const idx = workOrders.findIndex(w => w.id ===
       body.id);
  const updated = idx >= 0 ? { ...workOrders[idx
      ], ...body } : { ...body };
  if (idx >= 0) workOrders[idx] = updated;
  res.status(200).json(updated);
});

router.delete('/api/work-orders', (req, res) =>
    {
  const removed = workOrders.length;
  workOrders = [];
  res.status(200).json({ removed });
});

module.exports = router;
```

Listing 36: Round 1: Worker-1 frontend output

```
import React, { useState, useEffect } from '
    react';

function App() {
  const [orders, setOrders] = useState([]);
  const [email, setEmail] = useState('');
  const [password, setPassword] = useState('');
  const [form, setForm] = useState({});

  useEffect(() => {
    fetch('/api/work-orders')
      .then(r => r.json())
      .then(setOrders);
  }, []);

  const login = () => {
    fetch('/api/users/login', {
      method: 'POST',
      headers: { 'Content-Type': 'application/
          json' },
      body: JSON.stringify({ email, password })
    });
  };

  const create = () => {
    fetch('/api/work-orders', {
      method: 'POST',
      headers: { 'Content-Type': 'application/
          json' },
      body: JSON.stringify(form)
    })
      .then(r => r.json())
      .then(rec => setOrders(prev => [...prev,
          rec]));
  };

  return (
    <div>
      <input value={email} onChange={e =>
          setEmail(e.target.value)} />
      <input value={password} onChange={e =>
          setPassword(e.target.value)} />
      <button onClick={login}>Login</button>
      <input onChange={e => setForm({ ...form,
          orderNumber: e.target.value })} />
      <button onClick={create}>Create</button>
      <ul>
        {orders.map(o => (
          <li key={o.id}>{o.orderNumber} {o.
              productCode} {o.status}</li>
        ))}
      </ul>
    </div>
  );
}
```

```
export default App;
```

Listing 37: Round 2: Context Allocator output

```
<OPS>
[
{"op": "SCOPE","key": "feedback_t1","vis": [1]
},
{"op": "UPDATE","key": "frontend_focus","value":
     "The current frontend submits only
    orderNumber and displays only orderNumber,
    productCode, and status. Revise the create
    form to collect orderNumber, productCode,
    quantity, dueDate, and priority, and submit
    them as a JSON body to POST /api/work-orders
    . Revise the work-order list to display the
    created/returned work-order fields needed by
     the PRD: orderNumber, productCode, quantity
    , dueDate, priority, and status. Keep the
    existing /api/users/login and /api/work-
    orders paths unchanged."
},
{"op": "UPDATE","key": "contract","value": "Wire
     format unchanged. A work-order record is a
    top-level JSON object with fields {id,
    orderNumber, productCode, quantity, dueDate,
     priority, status}. POST /api/work-orders
    returns the created work-order object. GET /
    api/work-orders returns an array of work-
    order objects. The frontend should use these
     same field names in create requests and
    display the PRD-relevant work-order fields
    in the list view."
},
{"op": "SCOPE","key": "frontend_focus","vis":
    [1]
},
{"op": "SCOPE","key": "contract","vis": [0, 1]
}
]
</OPS>
```

Listing 38: Round 2: Worker-1 frontend output

```
import React, { useState, useEffect } from '
    react';

function App() {
  const [orders, setOrders] = useState([]);
  const [creds, setCreds] = useState({ email: '',
       password: '' });
  const [form, setForm] = useState({
    orderNumber: '', productCode: '', quantity:
        '', dueDate: '', priority: ''
  });
  const [msg, setMsg] = useState('');

  useEffect(() => {
    fetch('/api/work-orders')
      .then(r => r.json())
      .then(data => setOrders(Array.isArray(data)
          ? data : []));
  }, []);

  const login = () => {
    fetch('/api/users/login', {
      method: 'POST',
      headers: { 'Content-Type': 'application/
          json' },
      body: JSON.stringify(creds)
    })
      .then(r => r.json())
      .then(() => setMsg('logged in'))
      .catch(() => setMsg('login failed'));
  };

  const create = () => {
    fetch('/api/work-orders', {
      method: 'POST',
      headers: { 'Content-Type': 'application/
          json' },
      body: JSON.stringify(form)
    })
      .then(r => r.json())
      .then(rec => setOrders(prev => [...prev,
          rec]));
  };

  return (
    <div>
      <h2>Login</h2>
      <input value={creds.email}
             onChange={e => setCreds({ ...creds,
                 email: e.target.value })} />
      <input value={creds.password}
             onChange={e => setCreds({ ...creds,
                 password: e.target.value })} />
      <button onClick={login}>Login</button>
      <span>{msg}</span>

      <h2>Create Work Order</h2>
      {Object.keys(form).map(field => (
        <input key={field}
               value={form[field]}
               onChange={e => setForm({ ...form, [
                   field]: e.target.value })} />
      ))}
      <button onClick={create}>Create</button>

      <h2>Work Orders</h2>
      <ul>
        {orders.map((o, i) => (
          <li key={o.id ?? i}>
            {Object.entries(o).map(([k, v]) => (
              <span key={k}>{String(v)} </span>
            ))}
          </li>
        ))}
      </ul>
    </div>
  );
}

export default App;
```

J.3 CRM Lead Qualification

Task (`LeadId=00QWt000008989sMAA`)

The gold BANT failure label is `Authority`.

Listing 39: Round 1: Context Allocator output

```
<OPS>
[
  {"op":"WRITE","key":"budget_focus", "value":"
       Check approved budget, cost objections,
       procurement blockers, or ROI concerns.","
       vis":[0]},
```

```
  {"op":"WRITE","key":"authority_focus","value
      ":"Check whether the contact can approve
      purchase or requires manager, executive,
      procurement, or committee approval.","vis
      ":[1]},
  {"op":"WRITE","key":"need_focus","value":"
      Check whether the lead has a concrete pain
       point and product fit.","vis":[2]},
  {"op":"WRITE","key":"timeline_focus","value":"
      Check whether a buying, evaluation, or
      deployment timeline is explicit.","vis
      ":[3]}
]
</OPS>
```


Listing 40: Round 1: Worker-Budget output


```
{
  "role": "Budget Evidence Worker",
  "criterion": "Budget",
  "evidence_summary": "The lead mentioned a
      current budget of $3,203 and interest in
      discounts or financing options.",
  "supporting_records": [
    {"Body__c": "We have a current budget of $3
        ,203, but discounts or financing options
         would help."}
  ],
  "decision": "satisfied",
  "confidence": 0.7
}
```


Listing 41: Round 1: Worker-Authority output


```
{
  "role": "Authority Evidence Worker",
  "criterion": "Authority",
  "evidence_summary": "Raj Patel explicitly
      stated that he does not have full
      financial authority to approve large
      expenses and needs internal discussion
      before a decision.",
  "supporting_records": [
    {
      "Body__c": "Raj Patel: That would be very
          helpful since I don't have full
          financial authority to approve large
          expenses.",
      "Title": "VP of Technology Integration"
    }
  ],
  "decision": "failed",
  "confidence": 0.9
}
```


Listing 42: Round 1: Worker-Need output


```
{
  "role": "Need Evidence Worker",
  "criterion": "Need",
  "evidence_summary": "The lead expressed a
      concrete need to optimize energy
      consumption and reduce costs, with product
       interest aligned to that need.",
  "supporting_records": [
    {"Body__c": "We are looking for ways to
        optimize energy consumption and reduce
        costs."}
  ],
  "decision": "satisfied",
  "confidence": 0.8
}
```


Listing 43: Round 1: Worker-Timeline output


```
{
  "role": "Timeline Evidence Worker",
  "criterion": "Timeline",
  "evidence_summary": "The lead described
      urgency and a short installation schedule,
       indicating an explicit implementation
      timeline.",
  "supporting_records": [
    {"Body__c": "We are on a tight schedule and
        would need installation within three
        days."}
  ],
  "decision": "satisfied",
  "confidence": 0.7
}
```


Listing 44: Round 2: Context Allocator output


```
<OPS>
[
  {"op":"SCOPE","key":"worker_budget_summary_t0
      ","vis":[0]},
  {"op":"SCOPE","key":"
      worker_authority_summary_t0","vis":[1,2]},
  {"op":"SCOPE","key":"worker_need_summary_t0","
      vis":[2,3]},
  {"op":"SCOPE","key":"
      worker_timeline_summary_t0","vis":[3]},
  {"op":"UPDATE","key":"authority_focus",
   "value":"Verify whether the authority blocker
        is explicit negative evidence rather
       than merely missing information; decide
       Authority only from retrieved records."}
]
</OPS>
```


K Running Example: Context Board Evolution

We provide a step-by-step running example of DCA on the CoopHumanEval task `calculate_rectangle_area(dimensions)` from Appendix J. While Appendix J reports the raw model outputs, this section traces how those outputs are represented in the Context Board and how role-specific Worker views are constructed. For readability, long slot values in the listings are abbreviated with ellipses (. . .); the examples preserve the actual keys, visibility sets, and state transitions.

Following Section 4, let B_t denote the Context Board before the Context Allocator acts, and let $B_t^+ = \texttt{apply}(B_t, a_t^{(c)})$ denote the board after the Context Allocator edits it. Each slot is represented as `(key, value, visibility)`, where visibility is the set of Workers that may observe the slot.

Initial board. The episode begins with reserved environment slots visible to both Workers.

Listing 45: Initial Context Board B_0.

```
B_0 = [
  ("problem","...",[0, 1]),
  ("entry_point","...",[0, 1])
]
```

Round 1 allocator action. The Context Allocator writes a shared contract and two role-specific contexts.

Listing 46: Round 1 Context Allocator action $a_0^{(c)}$.

```
<OPS>
[
  {"op": "WRITE","key": "contract","value":
       "..""vis": [0, 1],},
  {"op": "WRITE","key": "aux_focus","value":
       "..""vis": [0],},
  {"op": "WRITE","key": "main_focus","value":
       "..""vis": [1],}
]
</OPS>
```

Board after allocator edits. After applying the allocator action, the board contains both reserved environment slots and allocator-written coordination slots.

Listing 47: Context Board after Round 1 allocator edits, B_0^+.

```
B_0^+ = [
  ("problem", "...", [0, 1]),
("entry_point", "...", [0, 1]),
  ("contract","...",[0, 1]),
  ("aux_focus","...",[0]),
  ("main_focus","...",[1])
]
```

Scoped Worker views. The Worker observations are induced by slot visibility. Worker 0 receives the helper-side plan, while Worker 1 receives the main-function plan. Both receive the shared task slots and contract.

Listing 48: Scoped Worker views at Round 1.

```
C_0^(0) = [
  ("problem", "..."),
  ("entry_point", "..."),
  ("contract", "..."),
  ("aux_focus", "...")
]

C_0^(1) = [
  ("problem", "..."),
  ("entry_point", "..."),
  ("contract", "..."),
  ("main_focus", "...")
]
```

Worker execution and deterministic append. The Workers act concurrently from their scoped views. Their outputs and evaluator feedback are then appended to the same Context Board using deterministic bookkeeping.

Listing 49: Round 1 Worker outputs and deterministic append.

```
append(B_0^+, a_0^(w), y_0) adds:
  ("worker_0_code_t1", "...", [0])
  ("worker_1_code_t1", "...", [1])
  ("feedback_t1","...",[0, 1])
```

Board before the next allocator round. Once appended, Worker outputs and feedback become ordinary Context Board slots. At the next round, the Context Allocator observes them and may retain, update, delete, or scope them.

Listing 50: Context Board before Round 2 allocator edits, B_1.

```
B_1 = [
  ("problem", "...", [0, 1]),
  ("entry_point", "...", [0, 1]),
  ("contract", "...", [0, 1]),
  ("aux_focus", "...", [0]),
  ("main_focus", "...", [1]),
  ("worker_0_code_t1", "...", [0]),
  ("worker_1_code_t1", "...", [1]),
  ("feedback_t1","...",[0, 1])
]
```

Round 2 allocator revision. The Context Allocator now edits the board. In this example, it deletes the stale helper implementation, tightens the shared contract, and revises the helper-side plan.

Listing 51: Round 2 Context Allocator action $a_1^{(c)}$.

```
<OPS>
[
  {"op": "DELETE","key": "worker_0_code_t1"},
  {"op": "UPDATE","key": "contract","value":
       "..."},
  {"op": "UPDATE","key": "aux_focus","value":
       "..."}
]
</OPS>
```

Revised board and Worker views. After the Round 2 edits, the stale helper code is removed, while the shared contract and helper-specific plan are updated. The main Worker still receives the shared contract, but only Worker 0 receives the revised helper plan.

Listing 52: Context Board after Round 2 allocator edits, B_1^+.

```
B_1^+ = [
  ("problem", "...", [0, 1]),
```

```
  ("entry_point", "...", [0, 1]),
  ("contract","...",[0, 1]),
  ("aux_focus","...",[0]),
  ("main_focus","...",[1]),
  ("worker_1_code_t1", "...", [1]),
  ("feedback_t1","...", [0, 1])
]
```

Listing 53: Scoped Worker views at Round 2.

```
C_1^(0) = [
  ("problem", "..."),
  ("entry_point", "..."),
  ("contract", "..."),
  ("aux_focus", "..."),
  ("feedback_t1", "...")
]

C_1^(1) = [
  ("problem", "..."),
  ("entry_point", "..."),
  ("contract", "..."),
  ("main_focus", "..."),
  ("worker_1_code_t1", "..."),
  ("feedback_t1", "Hidden tests fail because aux
       accepts floats ...")
]
```

This running example makes the state transition explicit. Allocator guidances, Worker actions, and environment feedback are all represented as Context Board slots. Worker actions and feedback are introduced through deterministic append operations, but after they are appended, they become ordinary slots that the Context Allocator can retain, update, delete, or scope in later rounds.

L Pseudocode

For compactness, Algorithm 1 writes $\pi^{(c)} = \pi^{(c)}_{\theta_c}$ and $\pi^{(i)} = \pi^{(i)}_{\theta_i}$.

M Implementation Details

All fine-tuned agents are adapted with LoRA while keeping the base model weights frozen. We use the same LoRA configuration across all methods and roles: rank $r = 32$, scaling factor $\alpha = 32$, dropout 0.0, `target_modules=all-linear`, and `bias=none`. The Context Allocator and Workers are parameterized by separate adapters. For efficient rollout collection and evaluation, we use vLLM as the generation backend. For evaluation, we use comparable test-set sizes across benchmarks, following the size of the CoopHumanEval test split. Unless otherwise stated, we sample with temperature 0.7 and top-$p = 0.7$. The decoding configuration is fixed across compared methods within each benchmark unless otherwise specified. We format the DCA Context Allocator outputs with `<OPS>` and `</OPS>` tags, and parse only the operations enclosed between these tags as valid context-allocation operations. Our initial codebase is based on CoMLRL[2]. We use a single NVIDIA A100 GPU for CoopHumanEval experiments, a single NVIDIA H200 GPU for MultiWebGen experiments, and two NVIDIA B200 GPUs for CRM lead qualification experiments.

For CoopHumanEval experiments, we use the CoopHumanEval dataset from OpenMLRL[3]. Following (Liu et al., 2026b), we use `test[16:]` as the training split and `test[:16]` as the test split. We use `Qwen/Qwen2.5-Coder-3B-Instruct` as the base model. The maximum generation length is set to 256 tokens for each Worker and 200 tokens

Algorithm 1 DCA Training

Input: Policies $\pi^{(c)}, \{\pi^{(i)}\}_{i=1}^{N}$, prompts $\rho^{(c)}, \{\rho^{(i)}\}_{i=1}^{N}$, group size G, horizon H, discount γ, board capacity K

for update $u = 1, \dots, U$ **do**
  Sample task prompt x
  for rollout $g = 1, \dots, G$ **do**
    Initialize $s_0^g \sim p_0$ and $B_0^g \leftarrow \emptyset$
    for $t = 0, \dots, H-1$ **do**
      $o_t^{(c),g} \leftarrow [\rho^{(c)}, x, B_t^g]$
      $a_t^{(c),g} \sim \pi^{(c)}(\cdot \mid o_t^{(c),g})$
      $B_t^{+,g} \leftarrow \mathrm{apply}(B_t^g, a_t^{(c),g})$
      for all $i = 1, \dots, N$ **in parallel do**
        $C_t^{(i),g} \leftarrow \{(\kappa, v_\kappa) \mid (\kappa, v_\kappa, \mathrm{vis}_\kappa) \in B_t^{+,g}, i \in \mathrm{vis}_\kappa\}$
        $o_t^{(i),g} \leftarrow [\rho^{(i)}, x, C_t^{(i),g}]$
        $a_t^{(i),g} \sim \pi^{(i)}(\cdot \mid o_t^{(i),g})$
      end for
      $\mathbf{a}_t^{(w),g} \leftarrow (a_t^{(1),g}, \dots, a_t^{(N),g})$
      $(s_{t+1}^g, y_t^g, r_t^g, \mathrm{done}_t^g) \leftarrow \mathrm{EnvStep}(s_t^g, \mathbf{a}_t^{(w),g})$
      $B_{t+1}^g \leftarrow \mathrm{append}(B_t^{+,g}, \mathbf{a}_t^{(w),g}, y_t^g)$
      $r_t^{(c),g} \leftarrow r_t^g + r_{t,\mathrm{fmt}}^g$
      $r_t^{(i),g} \leftarrow r_t^g$ for all $i = 1, \dots, N$
      if done_t^g **then**
        $T_g \leftarrow t + 1$
        break
      end if
    end for
  end for
  Compute discounted returns $R_t^{(c),g}$ and $R_t^{(i),g}$ for all valid steps $t < T_g$
  Normalize returns within the group of G rollouts to obtain $A_t^{(c),g}$ and $A_t^{(i),g}$
  Update θ_c using $\sum_{g=1}^{G} \sum_{t=0}^{T_g-1} A_t^{(c),g} \log \pi^{(c)}(a_t^{(c),g} \mid o_t^{(c),g})$
  for $i = 1, \dots, N$ **do**
    Update θ_i using $\sum_{g=1}^{G} \sum_{t=0}^{T_g-1} A_t^{(i),g} \log \pi^{(i)}(a_t^{(i),g} \mid o_t^{(i),g})$
  end for
end for

[2] https://github.com/OpenMLRL/CoMLRL

[3] https://huggingface.co/datasets/OpenMLRL/CoopHumanEval

for the Context Allocator. We train with learning rate 2.0×10^{-5}, per-device training batch size 1, rollout buffer size 8, and 4 generations per prompt. For rollout sampling, we use temperature 0.7 and top-$p = 0.7$ for Workers, and temperature 0.6 and top-$p = 0.6$ for the Context Allocator. The discount factor is set to 0.9. Following (Liu et al., 2026b), the task reward consists of four components: structural integrity, syntactical correctness, test pass rate, and cooperation quality. We use a format reward with maximum value 0.4, multiplied by the fraction of generated operations that are structurally valid. We set the maximum number of Context Board slots to 12, considering the 2-Worker, 3-round setting.

For MultiWebGen experiments, the dataset accompanies the paper and will be publicly released upon publication. We use `Qwen/Qwen2.5-Coder-3B-Instruct` as the base model. The maximum generation length is set to 1280 tokens for each Worker and 768 tokens for the Context Allocator. We train with learning rate 4.0×10^{-5}, per-device training batch size 1, rollout buffer size 8, and 4 generations per prompt. For rollout sampling, we use temperature 0.7 and top-$p = 0.7$ for Workers, and temperature 0.6 and top-$p = 0.6$ for the Context Allocator. The discount factor is set to 0.9. The task reward follows the structure of CoopHumanEval, but is adapted to full-stack web generation: it first checks whether all PRD-required endpoints are defined across the frontend and backend, and then evaluates backend tests, frontend tests, and E2E tests. We use the same format-reward design and slot size as in CoopHumanEval.

For CRM lead qualification, we use the CRMArena-Pro dataset[4]. We use the database and evaluation code from the official CRMArena codebase[5]. We use the first 70 examples as the training set, the next 10 examples as the validation set, and the last 20 examples as the test set from the CRMArena-Pro B2B lead qualification task. We use `Qwen/Qwen3-4B-Instruct-2507` and `Qwen/Qwen3-8B` as base models. The maximum generation length is set to 1024 tokens for each Worker, consisting of 256 tokens for SQL query generation and 768 tokens for evidence-summary generation, and 512 tokens for the Context Allocator. We train with learning rate 2.0×10^{-5}, per-device training batch size 1, rollout buffer size 8, and 8 generations per prompt. For rollout sampling, we use temperature 0.7 and top-$p = 0.7$ for Workers, and temperature 0.6 and top-$p = 0.6$ for the Context Allocator. The discount factor is set to 0.9. The task reward combines an exact-match reward, which checks whether the predicted lead-qualification answer matches the gold answer, and a label-F1 reward, which gives partial credit based on whether the individual qualification criteria are correctly predicted. We use a format reward with maximum value 0.2, multiplied by the fraction of generated operations that are structurally valid. Because this benchmark uses four Workers, we set the maximum number of Context Board slots to 20.

The code for this paper is available at `https://github.com/junho328/DCA`.

N Disclosure of the AI assistant

In accordance with the ACL AI writing assistance policy, we disclose the use of AI in this paper. We used ChatGPT, an AI language model developed by OpenAI, solely as a writing aid to enhance the clarity and readability of our manuscript.

[4] `https://huggingface.co/datasets/Salesforce/CRMArenaPro`

[5] `https://github.com/SalesforceAIResearch/CRMArena`