# OptiMAS: Automatically Optimize Multi-Agent System

**Anonymous ACL submission**

## Abstract

Automated evolution of Multi-Agent Systems (MAS) holds significant potential for reducing the manual effort required to design and optimize LLM-based agent architectures. However, extant search-based paradigms face a fundamental trade-off, where an expanded optimization scope exacerbates evolutionary instability, while discrete branch-and-discard search isolates insights across lineages. To address these limitations, we propose a continuous, data-driven optimization paradigm built upon a unified ReAct-based infrastructure that reconciles a broad optimization scope with operational stability. Under this paradigm, we present *OptiMAS*, a task-agnostic agentic optimizer that leverages textual interaction trajectories and task feedback as loss signals for end-to-end MAS evolution. Equipped with a novel dual-track memory mechanism, OptiMAS sustains performance improvement over extended optimization horizons. Evaluation on four heterogeneous agentic benchmarks with three varying scale and accessibility LLM backbones, demonstrates that OptiMAS consistently achieves competitive or superior accuracy relative to both domain-specialized hand-crafted systems and existing evolutionary methods. Our work establishes a practical milestone toward robust, automated MAS evolution.

## 1 Introduction

Collaborative multi-agent systems (MAS) have fundamentally expanded the operational envelope of large language models (LLMs), extending their reach from advanced cognitive reasoning to pragmatic real-world information foraging (Wei et al., 2025; Mialon et al., 2023; Liu et al., 2026) and autonomous software engineering (Openai, 2024; Yang et al., 2024a; Jimenez et al., 2024). However, manual MAS construction remains inherently labor-intensive, demanding meticulous prompt calibration, capability augmentation, and orchestration design through iterative empirical trials. While recent work (Hu et al., 2024a) advocates for automated agent evolution, this nascent field has yet to match the efficacy and reliability of carefully handcrafted systems in practical deployments.

Existing automated MAS evolution efforts center on two axes: *optimization scope* and *evolutionary paradigm* (Gao et al., 2025b). The optimization scope governs which facets of a MAS undergo evolution, ranging from prompt tuning to autonomous agent creation (Shinn et al., 2023; Sun et al., 2023; Khattab et al., 2023; Fernando et al., 2023; Yuan et al., 2024; Wang et al., 2023a; Qiu et al., 2025; Hu et al., 2024b). Recent full-stack approaches that adapt the agent infrastructure itself at test time (Zhang et al., 2025c) have further automated the design pipeline, progressively minimizing human intervention. On the paradigm axis, search-based methods remain dominant (Zhang et al., 2024, 2025a; Hu et al., 2026). As illustrated in Figure 1, evolved MAS candidates are compiled into a structured pool, with subsequent generations inheriting from and expanding upon diverging ancestral branches (Lee et al., 2026; Zhang et al., 2025b). Despite these advances, handcrafted MAS architectures continue to prevail in industrial production (Xia et al., 2025), as current automated approaches face a persistent dilemma: (1) broadening the optimization scope exacerbates evolutionary instability, impeding robust generalization across diverse tasks (Zhang et al., 2024, 2025a); and (2) search-oriented mechanisms encounter severe performance bottlenecks as accumulated evolutionary insights remain confined to isolated lineages and cannot transfer across branches.

To overcome these structural limitations, we propose **OptiMAS**, an optimization-based MAS evolution paradigm driven by a general-purpose agentic optimizer. We first establish a ReAct-centric (Yao et al., 2022) foundational infrastructure shared symmetrically by the evolving MAS and the optimizer.

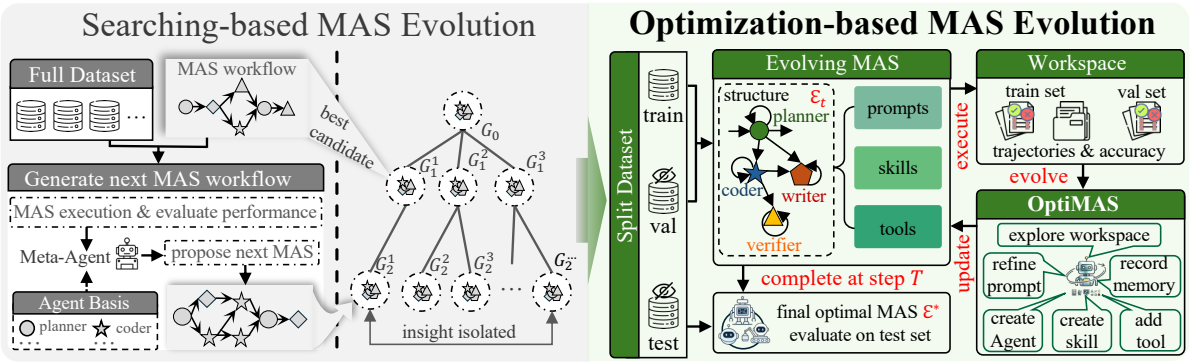


Figure 1: **Comparison of MAS evolution paradigms. (Left)** Search-based methods frame evolution as stochastic tree-branching ($G_0 \rightarrow G_t^i$) over full datasets, incurring low efficiency with insights isolated across branches. **(Right)** OptiMAS reformulates evolution as a data-driven continuous optimization over partitioned data (train/val/test), channeling action trajectories and accuracy as loss signals for OptiMAS to derive textual gradients and perform end-to-end MAS updates, yielding stable, sample-efficient, and overfit-resistant synthesis.

This unified substrate instantiates minimal configurations into robust, operational multi-agent systems, effectively bridging the gap between an expanded optimization scope and evolutionary stability (Zhang et al., 2025c). In contrast to search-based methods, our framework cultivates a MAS from inception through a data-driven optimization pipeline. As illustrated in Figure 1, the evolving MAS executes tasks on a partitioned *train set* at each step. Its complete action trajectories and performance metrics are then framed as a *loss* signal and routed to the optimizer for textual *gradient* induction and structural refinement. To preempt overfitting, generalization is continuously calibrated via execution accuracy on an optimizer-invisible *validation* set, with a held-out *test* set reserved solely for final evaluation. To sustain coherent optimization across an extended evolutionary horizon, OptiMAS incorporates a *dual-track memory mechanism* that continuously accumulates, validates, and retrieves evolutionary insights. Extensive experiments across heterogeneous benchmarks (Wei et al., 2025; Openai, 2024; Mialon et al., 2023; Styles et al., 2024) demonstrate the exceptional efficacy of our framework. Our principal contributions are:

- We introduce an optimization-based MAS evolution paradigm underpinned by a unified infrastructure that stabilizes the evolution while accommodating broad-scope MAS design.
- We present OptiMAS, a task-agnostic agentic optimizer featuring dual-track memory mechanism that enables steady performance compounding throughout prolonged optimization.
- We provide comprehensive evaluation on four challenging benchmarks, demonstrating the substantial potential of our framework for synthesizing practically deployable MAS.

## 2 Related Works

### 2.1 LLM-based Agent and MAS

Chain-of-thought prompting (Wei et al., 2022) first endowed LLMs with multi-step reasoning, and ReAct (Yao et al., 2022) closed the perception-action loop by interleaving reasoning traces with grounded tool calls, establishing a paradigm that underpins most contemporary agent architectures. Subsequent advances in verbal self-reflection (Shinn et al., 2023), autonomous tool acquisition (Schick et al., 2023), and task-level orchestration (Shen et al., 2023) further broadened the single-agent design space. Recognizing that complex objectives benefit from division of labor, researchers organized multiple agents into collaborative MAS. CAMEL (Li et al., 2023) and AutoGen (Wu et al., 2024) explored role-play and multi-turn coordination. MetaGPT (Hong et al., 2023) and ChatDev (Qian et al., 2024) imposed structured workflows for software development. AgentVerse (Chen et al., 2024) and multi-agent debate (Du et al., 2023) demonstrated that dynamic team assembly and adversarial interaction improve both performance and factual consistency. Despite this architectural diversity, existing MAS remain hand-designed, task-specific, and static once deployed (Zhou et al., 2025; Wang et al., 2023b), motivating the pursuit of automated, self-evolving MAS design.

### 2.2 Automated MAS Evolution

Automated MAS design spans a widening spectrum of design freedom (Gao et al., 2025b). Prompt-level methods (DSPy (Khattab et al., 2023), PromptBreeder (Fernando et al., 2023)) and agent-profiling approaches (EvoAgent (Yuan et al., 2024)) optimize within frozen topologies. Workflow-level search, represented by ADAS (Hu et al., 2024a), AFlow (Zhang et al., 2024), GPTSwarm (Zhuge et al., 2024), and AgentSquare (Shang et al., 2025), extends evolution to control flow, yet the resulting structures typically chain shallow, single-call agents, discard all non-selected search branches, and draw from a fixed role vocabulary. Query-adaptive extensions such as MaAS (Zhang et al., 2025b), EvoFlow (Zhang et al., 2025a), MAS-GPT (Ye et al., 2025b), and FlowReasoner (Gao et al., 2025a) relax the static-workflow assumption but inherit the shallow-agent limitation while introducing substantial training overhead. At the opposite extreme, fully self-referential systems (Gödel Agent (Yin et al., 2025), Darwin Gödel Machine (Zhang et al., 2025c)) evolve the agent codebase itself, yet presuppose strong code-generation backbones and frequently yield non-executable configurations. Our work navigates this tension by jointly evolving prompts, skills, orchestration topology, and tool configurations under a stable, protocol-guided optimization framework in which each agent operates as a recursive ReAct-based reasoner with proactive inter-agent invocation, achieving a broader optimization scope than constrained workflow search while maintaining greater reliability than unconstrained codebase mutation.

## 3 Methodology

### 3.1 Infrastructure

We formalize the multi-agent infrastructure and delineate the optimization scope of OptiMAS.

**ReAct Agent.** Our infrastructure adopts ReAct (Yao et al., 2022) as the atomic execution primitive for every agent. An agent $\mathcal{A}$ iteratively generates a thought $\tau$ followed by an action $a$, each conditioned on the accumulated context and the latest observation $o$ from environment $\Omega$. This interleaved history forms a trajectory $S_i = (\mathcal{P}, \tau_1, a_1, o_1, \ldots, \tau_i, a_i, o_i)$, where $\mathcal{P}$ denotes the initial prompt. At step $i$, the policy $\pi_{\mathcal{M}}$, parameterized by LLM backbone $\mathcal{M}$, produces:

$$\tau_{i+1}, a_{i+1} = \pi_{\mathcal{M}}(\cdot \mid S_i), \ \tau_1, a_1 = \pi_{\mathcal{M}}(\cdot \mid \mathcal{P}), \quad (1)$$

iterating until a capped horizon $n$ or an explicit termination signal.

**Proactive Assign-Deliver.** To compose ReAct agents into a collaborative MAS without hardcoded workflows, we adopt a *proactive assign-deliver* protocol. Task delegation from a parent agent $\mathcal{A}$ to its sub-agents is modeled as a tool-invocation action. The sub-agent resolves the assignment and returns its solution to the parent context as a structured observation. Agents are thus activated on demand rather than by a fixed schedule, providing both flexibility and scalability required for non-deterministic, complex workflows.

**Comprehensive Optimization Scope.** This unified infrastructure yields a configuration-driven MAS whose optimization scope $\mathcal{E}$ spans from prompt tuning to the macroscopic orchestration graph $\mathcal{G}(\mathbb{A})$ over an evolving agent population $\mathbb{A}$:

$$\mathcal{E} = \left\{\mathbb{A},\, \mathcal{G}(\mathbb{A}),\, (\mathcal{M}_{\mathcal{A}},\, \mathcal{P}_{\mathcal{A}},\, \mathcal{K}_{\mathcal{A}},\, \mathcal{T}_{\mathcal{A}})_{\mathcal{A}\in\mathbb{A}}\right\}, \quad (2)$$

where $\mathcal{M}_{\mathcal{A}}$, $\mathcal{P}_{\mathcal{A}}$, $\mathcal{K}_{\mathcal{A}}$, and $\mathcal{T}_{\mathcal{A}} \subseteq \mathbb{T}$ denote the LLM backbone, prompts, skill set, and accessible toolkit for each agent $\mathcal{A}$. To preserve operational stability, OptiMAS composes tools from a predefined macro-library $\mathbb{T}$ rather than generating code from scratch (Appendix E). Unlike prior methods that search a codified flow within a fixed population or static graph (Zhang et al., 2024; Hu et al., 2024a; Zhang et al., 2025b), OptiMAS freely instantiates heterogeneous agent roles via prompt compilation $\mathcal{P}_{\mathcal{A}}$ and coordinates $\mathcal{G}(\mathbb{A})$ through assign-deliver actions. The sole constraint on $\mathcal{G}(\mathbb{A})$ is the directed acyclic graph (DAG) property, guaranteeing deterministic termination.

### 3.2 Optimization Paradigm

Inspired by deep learning (LeCun et al., 2015; Michalski et al., 1983), we reformulate MAS evolution as an optimization-based paradigm. Instead of discretely selecting promising candidates and creating offspring along diverging branches, OptiMAS continuously refines the MAS configuration guided by textual gradients, thereby seeking monotonic performance improvement.

As illustrated in Figure 1, the task query set is first partitioned into $\mathcal{X}_{\text{train}}$, $\mathcal{X}_{\text{val}}$, and $\mathcal{X}_{\text{test}}$. At each training step $t$, a batch $\mathcal{B}_t$ of size $n$ is drawn from $\mathcal{X}_{\text{train}}$ via the adaptive sampling strategy described in Algorithm 1, which proactively revisits previous failures while preserving exploration dynamics.

**Algorithm 1** Adaptive Sampling and Priority Weighting

**Input:** $\mathcal{X}_{\text{train}}, \mathcal{F}_{\text{prev}}, \mathbf{p}, n, \phi$
**function** ADAPTIVESAMPLE($\mathcal{X}_{\text{train}}, \mathcal{F}_{\text{prev}}, \mathbf{p}, n, \phi$)
  $n_1 \leftarrow \lfloor \phi \cdot n \rfloor, \quad n_2 \leftarrow n - n_1$
  **if** $|\mathcal{F}_{\text{prev}}| \geq n_1$ **then**
    $\mathcal{B}_{\text{fail}} \sim \text{Categorical}(\mathcal{F}_{\text{prev}}, \mathbf{p}, n_1)$
    $\mathcal{B}_{\text{gen}} \sim \text{Categorical}(\mathcal{X}_{\text{train}} \setminus \mathcal{B}_{\text{fail}}, \mathbf{p}, n_2)$
  **else**
    $\mathcal{B}_{\text{fail}} \leftarrow \mathcal{F}_{\text{prev}}; \; n_2 \leftarrow n - |\mathcal{F}_{\text{prev}}|$
    $\mathcal{B}_{\text{gen}} \sim \text{Categorical}(\mathcal{X}_{\text{train}} \setminus \mathcal{B}_{\text{fail}}, \mathbf{p}, n_2)$
  **return** $\mathcal{B} \leftarrow \mathcal{B}_{\text{gen}} \cup \mathcal{B}_{\text{fail}}$
**Input:** update factors $\alpha, \beta$, bounds $p_{\min}, p_{\max}$, interval $\tau$
**function** UPDATEWEIGHTS($\mathbf{p}, \mathcal{B}, R_{\text{train}}, t$)
  **for** each sample $x_i \in \mathcal{B}$ **do**
    **if** is_correct($x_i \mid R_{\text{train}}$) **then**
      $p_i \leftarrow \max(p_{\min}, p_i \cdot \alpha)$
    **else**
      $p_i \leftarrow \min(p_{\max}, p_i \cdot \beta)$
  **if** $t \bmod \tau = 0$ **then** $\mathbf{p} \leftarrow \mathbf{1}_{|\mathcal{X}_{\text{train}}|}$
  **return** $\mathbf{p}$

**Algorithm 2** Continuous MAS Evolution Paradigm

**Input:** Initial MAS $\mathcal{E}_0$; Task queries $\mathcal{X}$; Epochs $T$; Batch size $n$; Retry ratio $\phi$
**Output:** Optimized Multi-Agent System $\mathcal{E}^*$
Partition query set $\mathcal{X} \rightarrow \{\mathcal{X}_{\text{train}}, \mathcal{X}_{\text{val}}\}$
Initialize sampling priority $\mathbf{p} \leftarrow \mathbf{1}_{|\mathcal{X}_{\text{train}}|}$ ▷ *Uniform init*
$\mathcal{F}_{\text{prev}} \leftarrow \varnothing$ ▷ *Failed queries from previous step*
**for** $t = 1$ **to** $T$ **do**
  $\mathcal{B}_t \leftarrow$ ADAPTIVESAMPLE($\mathcal{X}_{\text{train}}, \mathcal{F}_{\text{prev}}, \mathbf{p}, n, \phi$)
  $R_{\text{train}}, \mathcal{W}_{\text{train}} \leftarrow$ INFER&EVALUATE($\mathcal{E}_{t-1}, \mathcal{B}_t$)
  $R_{\text{val}}, _ \leftarrow$ INFER&EVALUATE($\mathcal{E}_{t-1}, \mathcal{X}_{\text{val}}$)
  $\mathcal{E}_t, \nabla_{\text{text}} \leftarrow$ OptiMAS($\mathcal{E}_{t-1}, \mathcal{W}_{\text{train}}, R_{\text{train}}, R_{\text{val}}$)
  $\mathcal{F}_{\text{prev}} \leftarrow \{x_i \in \mathcal{B}_t \mid \text{is_incorrect}(x_i)\}$
  $\mathbf{p} \leftarrow$ UPDATEWEIGHTS($\mathbf{p}, \mathcal{B}_t, R_{\text{train}}, t$)
$\mathcal{E}^* \leftarrow \mathcal{E}_T$
**return** $\mathcal{E}^*$

The current MAS $\mathcal{E}_{t-1}$ then executes on both $\mathcal{B}_t$ and $\mathcal{X}_{\text{val}}$, and the resulting outputs are evaluated against ground-truth references, yielding accuracy scores $R_{\text{train}}$ and $R_{\text{val}}$.

In the back-propagation stage, the agentic optimizer produces textual gradients and evolves $\mathcal{E}_{t-1}$ accordingly by deeply inspecting the workspace $\mathcal{W}_{\text{train}}$, which preserves the complete trajectory $S$ generated by $\mathcal{E}_{t-1}$ for each query along with all intermediate artifacts, e.g., code, crawled documents, working notes (Figure 2a). Equipped with versatile tools, the optimizer diagnoses root causes of failures and identifies recurring success patterns. Unlike structured backward passes that separate gradient computation from parameter update (LeCun et al., 2015; Yellamraju et al., 2024), our optimizer autonomously performs gradient analysis and MAS configuration improvement in an end-to-end manner, preserving coherence of reasoning throughout. Notably, the workspace of $\mathcal{E}_{t-1}$ on $\mathcal{X}_{\text{val}}$ is screened from the optimizer, while only the scalar accuracy $R_{\text{val}}$ provided as a generalization signal to guard against overfitting.

This inference-evolution cycle repeats until the maximum step $T$ is reached. The full algorithmic flow is given in Algorithm 2.

### 3.3 OptiMAS

To achieve effective continual evolution across diverse agentic tasks, we propose OptiMAS, a general-purpose agentic optimizer specialized for MAS evolution (Figure 2d), equipped with a comprehensive toolkit and dedicated skills $\mathcal{K}$ encoding knowledge of the MAS infrastructure and task-agnostic optimization scope (Appendix F).

Despite this self-contained design, long-horizon MAS evolution poses stability challenges. LLM amnesia and hallucination erode reasoning consistency as context grows, while textual gradients lack strict descent guarantees. We therefore introduce a dual-track memory mechanism, namely *plan-oriented short-term memory* and *hypothesis-driven long-term memory*, enabling OptiMAS to accumulate, verify, and propagate evolutionary insights across steps.

**Plan-oriented short-term memory.** MAS evolution on challenging tasks demands OptiMAS to process million-token trajectories for textual gradient extraction, making intra-step consistency critical. After obtaining performance metrics $R_{\text{train/val}}$ and preliminarily exploring $\mathcal{W}_{\text{train}}$, OptiMAS creates an instructional plan that serves as a structural spine throughout the back-propagation stage, mitigating forgetting and hallucination-induced drift. To balance flexibility with discipline, plans are managed via four statuses (`waiting`, `executing`, `completed`, `dropped`) and four operations (`add`, `modify`, `review`, `delete`). Crucially, OptiMAS is forbidden from erasing content, and inapplicable instructions are instead transitioned to `dropped` with mandatory justification, preserving a complete audit trail that prevents hallucination-affected inconsistencies. A programmatic completeness check at step termination enforces that all instructions reach a terminal status, redirecting OptiMAS to resume unfinished items otherwise (Figure 2c).

**Hypothesis-driven long-term memory.** While short-term memory governs intra-step coherence, MAS evolution demands *cross-step continuity*. Insights from one batch must be empirically veri-

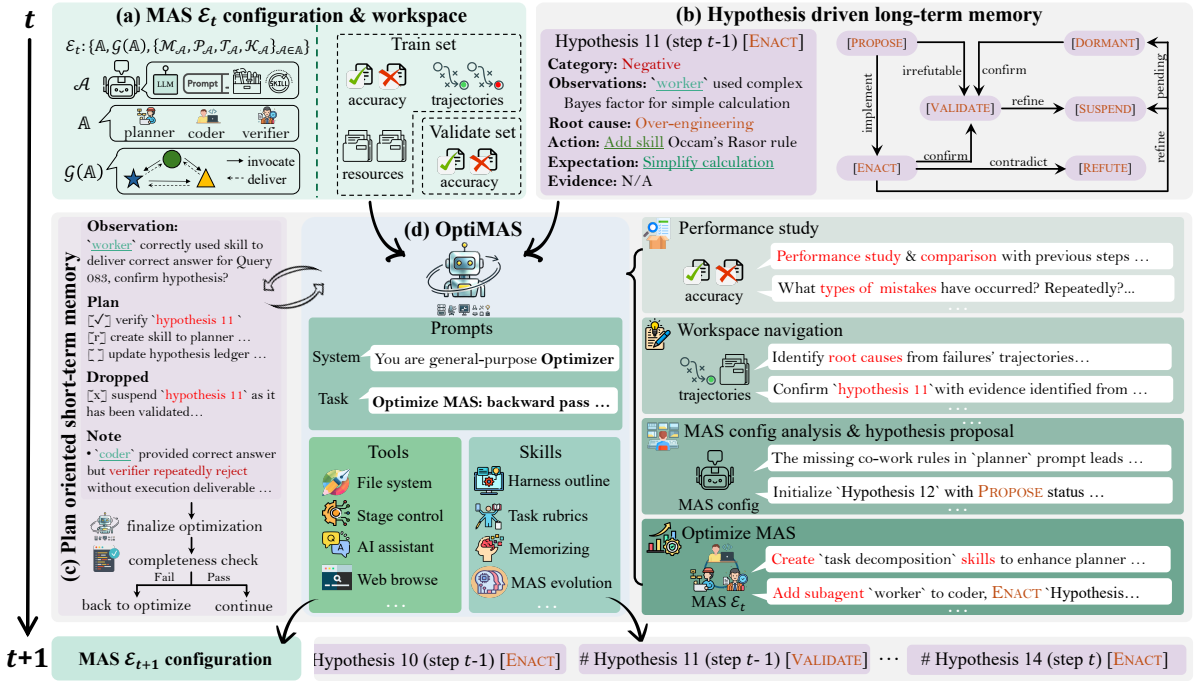


Figure 2: **OptiMAS optimizer architecture at step $t$. (a)** The MAS $\mathcal{E}_t$ executes on train/val queries, producing trajectories and accuracy. **(d)** The optimizer diagnoses root causes via workspace inspection and evolves $\mathcal{E}_t$, governed by **(c)** a plan-oriented short-term memory enforcing intra-step consistency and **(b)** a hypothesis-driven long-term memory maintaining a state-tracked ledger of falsifiable hypotheses across steps, yielding $\mathcal{E}_{t+1}$.

fied over subsequent iterations. Without persistent memory, the optimizer risks repeating failed interventions, accidentally reverting effective improvements, or conflating stochastic noise with systematic deficiencies.

We address this via hypothesis-driven long-term memory, whose core is a closed-loop lifecycle with verifiable rubrics governing how evolutionary insights accumulate across steps (Figure 2b). Each hypothesis is initialized as PROPOSE with OptiMAS's bootstrapped attributes, including classification, trajectory observations, identified textual gradient, prescribed MAS modifications, and falsifiable expectations. The hypothesis then transitions through a principled lifecycle:

- PROPOSE → ENACT: upon applying the modification to $\mathcal{E}_t$.
- ENACT → VALIDATE: when trajectory-level evidence, not accuracy alone, confirms the expected improvement.
- ENACT → REFUTE: when analysis reveals negative effects, triggering reversion.
- ENACT → SUSPEND/DORMANT: upon replacement by a successor or insufficient verification scenarios.

At each step, OptiMAS re-evaluates all active hypotheses against $\mathcal{W}_{\text{train}}$ evidence and maintains an accumulating evidence log. Every modification to evolving $\mathcal{E}_t$ must reference a hypothesis identifier, ensuring traceability.

By coupling each MAS modification to a falsifiable, evidence-grounded hypothesis, hypothesis-driven long-term memory transforms MAS evolution from undirected search into a principled, self-correcting optimization process, analogous to how a research log disciplines scientific experiments.

## 4 Experiments

### 4.1 Experiment Settings

**Benchmarks.** We evaluate on four benchmarks spanning heterogeneous agentic task settings to ensure comprehensive and rigorous assessment of evolutionary MAS approaches. *WorkBench* (Styles et al., 2024) tests multi-step planning, tool manipulation, and hierarchical execution on realistic workplace tasks. *GAIA-text* (Kazemi et al., 2025) poses complex reasoning questions requiring mathematical calculation, file parsing, and cross-tool orchestration. *BrowseComp* (Wei et al., 2025) targets long-horizon web navigation and programmatic multi-page information synthesis. *SWE-Bench-*

Table 1: Main results (task accuracy, %) across four benchmarks and three backbone LLMs: GPT5, Qwen3.6 and Gemini3 (abbreviated in column headers; full specifications in Section 4.1). Green and purple highlight the best and second-best results per backbone column. (–) denotes that such methodology is inapplicable or non-executable on given configurations. (*) marks evaluation subsets sampled via data curation(Appendix A).

| Method | WorkBench (%) ↑ | | | GAIA (%)↑ | | | BrowseComp* (%)↑ | | | SWE-Bench* (%)↑ | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | GPT5 | Qwen3.6 | Gemini3 | GPT5 | Qwen3.6 | Gemini3 | GPT5 | Qwen3.6 | Gemini3 | GPT5 | Qwen3.6 | Gemini3 |
| *Self-define Initial Agent* | | | | | | | | | | | | |
| Single $\mathcal{E}_0$ | 18.6 | 65.6 | 71.7 | 49.5 | 73.1 | 80.7 | 8.3 | 33.9 | 52.2 | 29.4 | 61.1 | 72.8 |
| *Hand-crafted MAS* | | | | | | | | | | | | |
| SMoA | 15.8 | 57.4 | 66.3 | 38.7 | 50.0 | 59.7 | 1.7 | 6.7 | 20.0 | – | – | – |
| MAS Debate | 15.0 | 39.2 | 44.2 | 32.3 | 38.7 | 50.0 | 5.0 | 11.7 | 20.0 | 13.3 | 23.3 | 26.7 |
| SWE-Agent | – | – | – | – | – | – | – | – | – | 16.7 | 67.2 | 75.0 |
| Tongyi-DR | – | – | – | 62.9 | 80.7 | 75.8 | 10.0 | 38.3 | 48.3 | – | – | – |
| *Evolutionary MAS* | | | | | | | | | | | | |
| ADAS | 41.6 | 50.8 | 62.1 | 35.5 | 27.4 | 45.2 | 3.3 | 8.3 | 13.3 | 18.3 | 21.7 | 18.3 |
| EvoAgent | 46.1 | 56.4 | 67.5 | 41.9 | 46.8 | 62.9 | 3.3 | 6.6 | 23.3 | 15.0 | 16.7 | 21.7 |
| DGM | – | – | – | – | – | – | – | – | – | 8.3 | 25.0 | 55.0 |
| OptiMAS (Ours) | 64.0$_{(+45.4)}$ | 84.5$_{(+18.9)}$ | 92.93$_{(+21.23)}$ | 56.9$_{(+7.4)}$ | 81.2$_{(+8.1)}$ | 87.1$_{(+6.4)}$ | 11.1$_{(+2.8)}$ | 43.8$_{(+9.9)}$ | 58.3$_{(+6.1)}$ | 41.7$_{(+12.3)}$ | 68.9$_{(+7.8)}$ | 77.2$_{(+4.4)}$ |

*Verified (SWE-Bench)* (Jimenez et al., 2023) tasks agents with resolving production-grade GitHub issues through multi-file code comprehension and repository modification.

To mitigate the prohibitive computational overhead while preserving evaluation integrity, we construct evaluation subsets of 100 queries each via reproducible seed-fixed random sampling for *SWE-Bench-Verified* and *BrowseComp*. For *GAIA* and *WorkBench*, we use the full set. Partition details of $\mathcal{X}_{\text{train}}/\mathcal{X}_{\text{val}}/\mathcal{X}_{\text{test}}$ are provided in Appendix A. All baselines are evaluated on identical subsets and splits to ensure a strictly fair comparison.

**Metrics.** We follow each benchmark's official metric protocol. *SWE-Bench-Verified* and *WorkBench* deliverables are verified via official sandboxed execution. *BrowseComp* and *GAIA-text* use a standardized LLM-as-a-Judge pipeline applied uniformly across all methods. All test results are averaged over three independent trials to account for LLM non-determinism.

**Baselines.** We compare OptiMAS against two categories of competitive methodologies (details in Appendix B):

- **Hand-crafted MAS**: domain-specialized systems including *Tongyi-DeepResearch* (Team et al., 2025) for *GAIA-text*/*BrowseComp* and *SWE-Agent* (Yang et al., 2024b) for *SWE-Bench-Verified*, alongside general-purpose *SMoA* (Li et al., 2025) and *Multi-Agent Debate* (Du et al., 2023).

- **Automated agent methods**: the state-of-the-art frameworks, including *ADAS* (Hu et al., 2024a), *EvoAgent* (Yuan et al., 2025), and *DGM* (Zhang et al., 2025c).

We additionally report the single-agent initialization $\mathcal{E}_0$, parameterized with minimalist prompts and basic toolkits, as a baseline isolating the intrinsic capacity of our infrastructure (Appendix C).

**Implementation Details.** *Gemini-3-Flash (Gemini3)* serves as the OptiMAS backbone and is mirrored as the meta-agent across all automated evolution baselines to ensure equitable comparison. Evolving MAS candidates are evaluated with three backbones: *GPT-5-Nano (GPT5)*, *Qwen-3.6-35B-A3B (Qwen3.6)*, and *Gemini-3-Flash*. Hyperparameters and environment configurations are detailed in Appendix D.

### 4.2 Main Results

Table 1 reports task accuracy across four benchmarks and three backbone LLMs. We analyze the results along three axes: evolution efficacy, cross-domain generalization, and backbone universality.

**Evolution efficacy.** OptiMAS achieves the highest accuracy in most benchmark–backbone configurations and improves over its single-agent initialization $\mathcal{E}_0$ in all conditions, with gains ranging from +2.8% (BrowseComp/GPT5) to +45.4% (WorkBench/GPT5). Notably, a single task-agnostic OptiMAS configuration is applied identically across all four benchmarks and three backbones without any task-specific adaptation, yet consistently yields positive evolution gains. We attribute this exceptional efficacy to the broad optimization scope and our hypothesis-driven memory mechanism, which enable OptiMAS to diagnose specific failure patterns

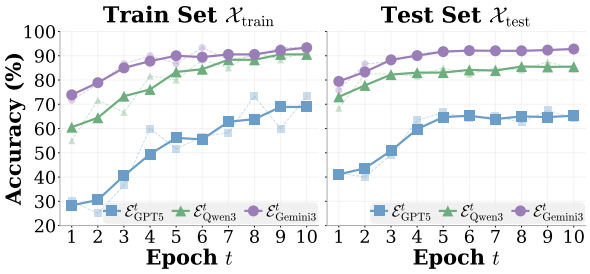


Figure 3: **Evolution dynamics on WorkBench.** Accuracy of $\mathcal{E}_t$ on $\mathcal{X}_{\text{train}}$ (left) and $\mathcal{X}_{\text{test}}$ (right) over 10 epochs. Dashed lines show mean over three trials while solid lines are moving averages to highlight trend.

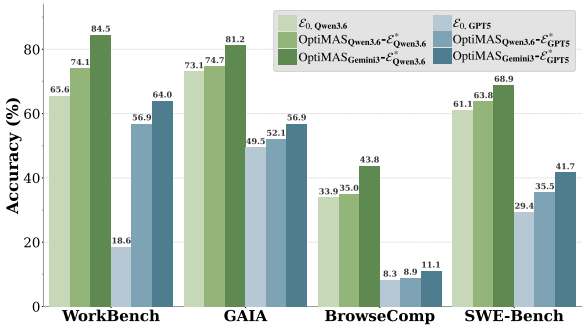


Figure 4: **Ablation analysis of optimizer backbones.** Performance is compared across combinations of two MAS backbones and two OptiMAS backbones.

and evolve targeted remedies regardless of the underlying domain. By contrast, the evolved systems produced by ADAS fall below our OptiMAS by up to 58.9% on SWE-Bench/Gemini3, suggesting that evolving MAS along a continuous, evidence-grounded paradigm is more effective than discrete search for long-horizon agentic tasks that demand multi-step reasoning and complex coordination.

**Cross-domain generalization.** OptiMAS matches or exceeds domain-specialized hand-crafted systems on their respective benchmarks while remaining fully transferable. On GAIA and BrowseComp, OptiMAS surpasses Tongyi-DR with both Qwen3.6 (81.2%/43.8% vs. 80.7%/38.3%) and Gemini3 (87.1%/58.3% vs. 75.8%/48.3%). On SWE-Bench, OptiMAS outperforms SWE-Agent across all three backbones. While these systems benefit from domain expertise, OptiMAS achieves competitive or superior performance on all four benchmarks from a general-purpose evolutionary optimizer, indicating that the breadth of the optimization scope enables task-adaptive capability.

**Backbone universality.** OptiMAS delivers consistent improvements across backbone LLMs of diverse scales and accessibility, including the closed-source GPT-5-Nano (128k context), the open-source Qwen-3.6-35B-A3B (262k context), and the commercial frontier Gemini-3-Flash (1M context). The evolution gain is inversely correlated with backbone capacity, e.g., +45.4%/+12.3% on WorkBench/SWE-Bench under GPT5 versus +15.5%/+4.4% under Gemini3, indicating that evolved orchestration and skills effectively compensate for limited backbone reasoning. This universality across both proprietary and open-source models suggests that OptiMAS is applicable to practical deployment scenarios without constraints on the underlying LLM.

### 4.3 Evolution Dynamics

Figure 3 traces the accuracy of each intermediate MAS $\mathcal{E}_t$ on the WorkBench train and test partitions over 10 optimization epochs. All three backbones exhibit progressive and near-monotonic improvement on both splits, with training accuracy gaining over +43% (GPT5), +36% (Qwen3.6), and +21% (Gemini3) absolute, and test accuracy closely tracking these advances. The smooth ascending curves empirically validate the continuous optimization paradigm proposed in Section 3. The hypothesis-driven long-term memory enables the optimizer to retain and compound insights across steps, producing sustained improvement that contrasts with the discrete, branch-and-discard dynamics of search-based methods. Equally notable is the narrow train-test gap maintained throughout evolution, indicating that the validation metric and adaptive sampling effectively prevent overfitting and ensure gains on $\mathcal{X}_{\text{train}}$ generalize to unseen queries. These results confirm that OptiMAS delivers stable, compounding MAS enhancement rather than volatile oscillation, a prerequisite for practical deployment.

### 4.4 Ablation Analysis

**Optimizer backbone.** Figure 4 compares the default Gemini-3-Flash OptiMAS against Qwen-3.6-35B-A3B across all four benchmarks and two MAS backbones. The Qwen-based OptiMAS yields positive gains over $\mathcal{E}_0$ in all eight configurations, confirming that the evolution framework is not contingent on a specific frontier model. The improvement is most pronounced on WorkBench (+8.5% for Qwen-MAS, +38.3% for GPT5-MAS) and SWE-Bench (+2.7%/+6.1%), where per-query agent working trajectories are relatively compact. However, the Qwen optimizer produces only marginal

Table 2: **Ablation analysis of framework components.** Bold indicates best results. "✕" signifies component removal, and "BS" represents batch size.

| Ablation Settings | WorkBench (%) ↑ | | SWE-Bench (%)↑ | |
|---|---|---|---|---|
| | GPT5 | Qwen3.6 | GPT5 | Qwen3.6 |
| Initial $\mathcal{E}_0$ (baseline) | 18.6 | 65.6 | 29.4 | 61.1 |
| **OptiMAS** w/ BS $n=10$ | **64.0**$_{(+45.4)}$ | **84.5**$_{(+18.9)}$ | **41.7**$_{(+12.3)}$ | **68.9**$_{(+7.8)}$ |
| ✕ w/ BS $n=4$ | 55.0$_{(+36.4)}$ | 81.1$_{(+15.5)}$ | 24.4$_{(-5.0)}$ | 62.8$_{(+1.7)}$ |
| ✕ Validation Metric | 58.2$_{(+39.6)}$ | 82.8$_{(+17.2)}$ | 28.9$_{(-0.5)}$ | 64.4$_{(+3.3)}$ |
| ✕ Hypothesis Memory | 53.9$_{(+35.3)}$ | 71.4$_{(+5.8)}$ | 22.2$_{(-7.2)}$ | 53.3$_{(-7.8)}$ |
| ✕ Adaptive Sampling | 60.3$_{(+41.7)}$ | 75.1$_{(+9.5)}$ | 24.4$_{(-5.0)}$ | 60.6$_{(-0.5)}$ |

gains (+1.6%/+1.1% for Qwen-MAS) on GAIA and BrowseComp, while the Gemini optimizer achieves substantially larger improvements on the same configurations (+8.1%/+9.9%). This disparity is consistent with the context-length bottleneck: complex tasks involving long-horizon web navigation generate extensive workspace artifacts, degrading the quality of gradient diagnosis. These results suggest that optimizer context capacity acts as a scaling factor for evolution quality on trajectory-intensive tasks, while the underlying framework remains effective across optimizer scales.

**Component effectiveness.** Table 2 isolates each framework component on WorkBench and SWE-Bench. Removing hypothesis memory causes the most severe degradation, with SWE-Bench accuracy regressing by over $-7\%$ on both backbones relative to $\mathcal{E}_0$. Without persistent inter-step memory, the optimizer cannot verify the effectiveness of prior interventions. Consequently, evolution reduces to isolated trial-and-error, which is particularly detrimental on complex tasks requiring sustained, coherent optimization. Removing adaptive sampling produces a similar pattern, with SWE-Bench performance falling below $\mathcal{E}_0$ on both backbones ($-5.0\%$/$-0.5\%$), as the optimizer loses the ability to examine previously failed queries and empirically validate its hypotheses. On the WorkBench, both ablations still yield gains over $\mathcal{E}_0$, indicating that the components become increasingly critical as task complexity grows. Reducing batch size degrades GPT5 on SWE-Bench by $-5.0\%$, because the weaker backbone frequently produces entirely incorrect batches that deprive the optimizer of positive signal for hypothesis validation. Removing the validation metric has a moderate effect overall but introduces slight overfitting on SWE-Bench/GPT5 ($-0.5\%$), consistent with its theoretical role for generalization.

**Cost and scalability.** Statistically, OptiMAS processes 1.4M to 3.2M tokens per optimization step across four benchmarks, equivalent to approximately 750K to 2.4M words of technical content, with input constituting over 98% of consumption. This volume, scaling with task-horizon complexity, reflects OptiMAS's ability to systematically explore large-scale workspaces containing extensive agent trajectories, intermediate code, retrieved documents, and working artifacts. Benefiting from the plan-oriented memory and advanced infrastructure, OptiMAS remains effective even when cumulative workspace content substantially exceeds the LLM's native context window, enabling thorough textual gradient analysis at a scale that meets industrial MAS evolution requirements. The evolved $\mathcal{E}^*$ is subsequently deployed at standard inference cost.

## 5 Conclusions

We presented OptiMAS, an optimization-based framework that achieves steady, continuous multi-agent system evolution on real-world, long-horizon agentic tasks. Our framework rests on three interconnected contributions. First, we established a unified ReAct-based infrastructure with a proactive assign-deliver protocol that reconciles a broad optimization scope with operational stability, enabling joint evolution of prompts, skills, orchestration, and tool configurations. Second, we reformulated MAS evolution as a continuous, data-driven optimization process with adaptive sampling and textual gradient analysis under strict train-validation-test separation to ensure generalization, replacing discrete branch-and-discard search with evidence-grounded progressive refinement. Third, we introduced the dual-track memory mechanism, comprising plan-oriented short-term memory for intra-step consistency and hypothesis-driven long-term memory for cross-step continuity, which transforms MAS evolution from undirected exploration into a principled, self-correcting process.

Extensive evaluation across four heterogeneous benchmarks and three backbone LLMs of varying scale and accessibility demonstrated that OptiMAS consistently improves over its initialization and achieves superior performance relative to both domain-specialized hand-crafted systems and existing evolutionary methods from a single task-agnostic configuration. We hope this work contributes to advancing the automated synthesis of deployable multi-agent systems.

## Limitations

**Backbone capability requirements.** The generality of our infrastructure, encompassing autonomous tool invocation, hierarchical sub-agent delegation, and proactive context management, places non-trivial demands on the backbone LLM's instruction-following and tool-calling proficiency. As a result, substantially weaker models may struggle to operate effectively within this framework. Extending compatibility to lighter-weight or edge-computation friendly models through simplified execution protocols or progressive capability scaffolding remains a valuable direction for future work.

**Optimization exploration breadth.** While OptiMAS yields consistent improvements across challenging agentic benchmarks, the current work represents an initial exploration of a vast design space. Drawing an analogy to the maturation of deep learning methodology, numerous dimensions remain to be investigated, including training curricula and scheduling strategies, dataset curation policies, multi-agent diversity and composition, and the inclusion of infrastructure components such as prompt templates and tools creation as first-class optimization targets. Each of these directions constitutes a promising avenue for extending the present framework.

## References


Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chi-Min Chan, Heyang Yu, Yaxi Lu, Yi-Hsin Hung, Chen Qian, and 1 others. 2024. Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors. In *ICLR*.

Yilun Du, Shuang Li, Antonio Torralba, Joshua B Tenenbaum, and Igor Mordatch. 2023. Improving factuality and reasoning in language models through multiagent debate. In *Forty-first International Conference on Machine Learning*.

Chrisantha Fernando, Dylan Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. 2023. Promptbreeder: Self-referential self-improvement via prompt evolution. *arXiv preprint arXiv:2309.16797*.

Hongcheng Gao, Yue Liu, Yufei He, Longxu Dou, Chao Du, Zhijie Deng, Bryan Hooi, Min Lin, and Tianyu Pang. 2025a. Flowreasoner: Reinforcing query-level meta-agents. *arXiv preprint arXiv:2504.15257*.

Huan-ang Gao, Jiayi Geng, Wenyue Hua, Mengkang Hu, Xinzhe Juan, Hongzhang Liu, Shilong Liu, Jiahao Qiu, Xuan Qi, Yiran Wu, and 1 others. 2025b. A survey of self-evolving agents: What, when, how, and where to evolve on the path to artificial super intelligence. *arXiv preprint arXiv:2507.21046*.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, and 1 others. 2023. Metagpt: Meta programming for a multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations*.

Shengran Hu, Cong Lu, and Jeff Clune. 2024a. Automated design of agentic systems. *arXiv preprint arXiv:2408.08435*.

Yue Hu, Yuzhu Cai, Yaxin Du, Xinyu Zhu, Xiangrui Liu, Zijie Yu, Yuchen Hou, Shuo Tang, and Siheng Chen. 2024b. Self-evolving multi-agent collaboration networks for software development. *arXiv preprint arXiv:2410.16946*.

Yuntong Hu, Matthew Trager, Yuting Zhang, Yi Zhang, Shuo Yang, Wei Xia, and Stefano Soatto. 2026. Evolutionary generation of multi-agent systems. *arXiv preprint arXiv:2602.06511*.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations*, volume 2024, pages 54107–54157.

Mehran Kazemi, Bahare Fatemi, Hritik Bansal, John Palowitch, Chrysovalantis Anastasiou, Sanket Vaibhav Mehta, Lalit K Jain, Virginia Aglietti, Disha Jindal, Yuanzhu Peter Chen, and 1 others. 2025. Bigbench extra hard. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 26473–26501.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T Joshi, Hanna Moazam, and 1 others. 2023. Dspy: Compiling declarative language model calls into self-improving pipelines. *arXiv preprint arXiv:2310.03714*.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *nature*, 521(7553):436–444.

Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. 2026. Meta-harness: End-to-end optimization of model harnesses. *arXiv preprint arXiv:2603.28052*.

Dawei Li, Zhen Tan, Peijia Qian, Yifan Li, Kumar Chaudhary, Lijie Hu, and Jiayi Shen. 2025. Smoa: Improving multi-agent large language models with s parse m ixture-o f-a gents. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 54–65. Springer.

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

Chang Liu, Chuqiao Kuang, Tianyi Zhuang, Yuxin Cheng, Huichi Zhou, Xiaoguang Li, and Lifeng Shang. 2026. Uis-digger: Towards comprehensive research agent systems for real-world unindexed information seeking. *arXiv preprint arXiv:2603.08117*.

Grégoire Mialon, Clémentine Fourrier, Thomas Wolf, Yann LeCun, and Thomas Scialom. 2023. Gaia: a benchmark for general ai assistants. In *The Twelfth International Conference on Learning Representations*.

R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, editors. 1983. *Machine Learning: An Artificial Intelligence Approach, Vol. I*. Tioga, Palo Alto, CA.

Openai. 2024. Introducing swe-bench verified. https://openai.com/index/introducing-swe-bench-verified.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, and 1 others. 2024. Chatdev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15174–15186.

Jiahao Qiu, Xuan Qi, Tongcheng Zhang, Xinzhe Juan, Jiacheng Guo, Yifu Lu, Yimin Wang, Zixin Yao, Qihan Ren, Xun Jiang, Xing Zhou, Dongrui Liu, Ling Yang, Yue Wu, Kaixuan Huang, Shilong Liu, Hongru Wang, and Mengdi Wang. 2025. Alita: Generalist agent enabling scalable agentic reasoning with minimal predefinition and maximal self-evolution. *CoRR*, abs/2505.20286.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. In *NeurIPS*.

Yu Shang, Yu Li, Keyu Zhao, Likai Ma, Jiahe Liu, Fengli Xu, and Yong Li. 2025. Agentsquare: Automatic LLM agent search in modular design space. In *ICLR*. OpenReview.net.

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. 2023. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face. *Advances in Neural Information Processing Systems*, 36:38154–38180.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*.

Olly Styles, Sam Miller, Patricio Cerda-Mardini, Tanaya Guha, Victor Sanchez, and Bertie Vidgen. 2024. Workbench: a benchmark dataset for agents in a realistic workplace setting. *arXiv preprint arXiv:2405.00823*.

Haotian Sun, Yuchen Zhuang, Lingkai Kong, Bo Dai, and Chao Zhang. 2023. AdaPlanner: Adaptive planning from feedback with language models. In *Advances in Neural Information Processing Systems*.

Tongyi DeepResearch Team, Baixuan Li, Bo Zhang, Dingchu Zhang, Fei Huang, Guangyu Li, Guoxin Chen, Huifeng Yin, Jialong Wu, Jingren Zhou, and 1 others. 2025. Tongyi deepresearch technical report. *arXiv preprint arXiv:2510.24701*.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2023a. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*.

Haiming Wang, Huajian Xin, Chuanyang Zheng, Lin Li, Zhengying Liu, Qingxing Cao, Yinya Huang, Jing Xiong, Han Shi, Enze Xie, and 1 others. 2023b. Lego-prover: Neural theorem proving with growing libraries. *arXiv preprint arXiv:2310.00656*.

Jason Wei, Zhiqing Sun, Spencer Papay, Scott McKinney, Jeffrey Han, Isa Fulford, Hyung Won Chung, Alex Tachard Passos, William Fedus, and Amelia Glaese. 2025. Browsecomp: A simple yet challenging benchmark for browsing agents. *arXiv preprint arXiv:2504.12516*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, and 1 others. 2024. Autogen: Enabling next-gen llm applications via multi-agent conversations. In *First Conference on Language Modeling*.

Chunqiu Steven Xia, Zhe Wang, Yan Yang, Yuxiang Wei, and Lingming Zhang. 2025. Live-swe-agent: Can software engineering agents self-evolve on the fly? *arXiv preprint arXiv:2511.13646*.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024a. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652.

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

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*.

Rui Ye, Zhongwang Zhang, Kuan Li, Huifeng Yin, Zhengwei Tao, Yida Zhao, Liangcai Su, Liwen Zhang, Zile Qiao, Xinyu Wang, and 1 others. 2025a. Agentfold: Long-horizon web agents with proactive context management. *arXiv preprint arXiv:2510.24699*.

Rui Ye and 1 others. 2025b. Mas-gpt: Training llms to build llm-based multi-agent systems. *arXiv preprint*.

S. Yellamraju, , and 1 others. 2024. TextGrad: Automatic "differentiation" via text. *arXiv preprint arXiv:2406.07496*.

Xunjian Yin, Xinyi Wang, Liangming Pan, Li Lin, Xiaojun Wan, and William Yang Wang. 2025. Gödel agent: A self-referential agent framework for recursive self-improvement. *Preprint*, arXiv:2410.04444.

Siyu Yuan, Kaitao Song, Jiangjie Chen, Xu Tan, Dongsheng Li, and Deqing Yang. 2024. Evoagent: Towards automatic multi-agent generation via evolutionary algorithms. *arXiv preprint arXiv:2406.14228*.

Siyu Yuan, Kaitao Song, Jiangjie Chen, Xu Tan, Dongsheng Li, and Deqing Yang. 2025. Evoagent: Towards automatic multi-agent generation via evolutionary algorithms. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 6192–6217.

Guanghui Zhang, Kang Chen, Guohao Wan, Hao Chang, Hao Cheng, and 1 others. 2025a. Evoflow: Evolving diverse agentic workflows on the fly. *arXiv preprint arXiv:2502.07373*.

Guanghui Zhang, Li Niu, Jianwei Fang, Kun Wang, Lu Bai, and 1 others. 2025b. Multi-agent architecture search via agentic supernet. *arXiv preprint arXiv:2502.04180*.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. 2025c. Darwin Godel Machine: Open-Ended Evolution of Self-Improving Agents. *arXiv preprint arXiv:2505.22954*.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, and 1 others. 2024. Aflow: Automating agentic workflow generation. *arXiv preprint arXiv:2410.10762*.

Han Zhou, Xingchen Wan, Ruoxi Sun, Hamid Palangi, Shariq Iqbal, Ivan Vulić, Anna Korhonen, and Sercan Ö Arık. 2025. Multi-agent design: Optimizing agents with better prompts and topologies. *arXiv preprint arXiv:2502.02533*.

Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. 2024. Gptswarm: Language agents as optimizable graphs. In *ICML*. OpenReview.net.

# A Benchmark $\mathcal{X}$ Curation and Partition

## A.1 Benchmarks $\mathcal{X}$ Curation

Due to prohibitive computational resource requirement on long-horizon SWE-Bench-Verified and tremendous search and crawl required BrowseComp benchmarks, we construct fixed-size subsets, each compressing 100 instances, from the two established benchmarks by fixed-seed ramdon sampling to ensure reproducibility.

**SWE-Bench-Verified.** SWE-bench Verified (Jimenez et al., 2024) is a human verified subset of the SWE-bench benchmark comprising real-world GitHub issues paired with ground-truth patches. We first obtain the full SWE-bench Verified dataset (totoally 500 instances) and sample a subset of $N = 100$ instances using Python's `random.Random` with seed $s = 42$. The resulting subset is stored as a static JSONL file and remains fixed throughout all experiments.

**BrowseComp.** BrowseComp (Wei et al., 2025) is a benchmark of challenging factual questions that require multi-step web browsing and cross-source verification, which totally compresses 1,266 queries. We also following the same random sampling rules with same random seed $s = 42$ in SWE-Bench to sample $N = 100$ queries from the BrowseComp validation set.

**GAIA-text and WorkBench.** GAIA-text (Mialon et al., 2023) comprises 103 complex reasoning questions requiring mathematical calculation, file parsing, and cross-tool orchestration. WorkBench (Styles et al., 2024) contains 690 tasks spanning six workplace domains that test multi-step planning, tool manipulation, and hierarchical execution. As both benchmarks impose substantially lower computational overhead than SWE-Bench-Verified and BrowseComp, we retain their complete established sets as $\mathcal{X}$ without subsampling.

## A.2 Deterministic $\mathcal{X}_{\text{train/val/test}}$ Partition

**SWE-Bench-Verified, BrowseComp, and GAIA.** As these three benchmarks share comparable scales (100, 100, and 103 instances respectively), we apply a unified deterministic partitioning procedure to each:

- **Index generation.** Construct an index array $I = [0, 1, \ldots, N-1]$.
- **Deterministic shuffle.** Shuffle $I$ in-place using Python's `random.Random` initialized with seed $s = 42$.
- **Contiguous partitioning.** Split the shuffled array into three contiguous segments with ratios $(r_{\text{train}}, r_{\text{val}}, r_{\text{test}}) = (0.36, 0.04, 0.60)$:

$$n_{\text{train}} = \lfloor N \cdot r_{\text{train}} \rfloor, \quad (3)$$
$$n_{\text{val}} = \lfloor N \cdot r_{\text{val}} \rfloor, \quad (4)$$
$$n_{\text{test}} = N - n_{\text{train}} - n_{\text{val}}. \quad (5)$$

- **Assignment.**

$$\mathcal{X}_{\text{train}} = \{x_{I[i]} \mid 0 \le i < n_{\text{train}}\},$$
$$\mathcal{X}_{\text{val}} = \{x_{I[i]} \mid n_{\text{train}} \le i < n_{\text{train}} + n_{\text{val}}\},$$
$$\mathcal{X}_{\text{test}} = \{x_{I[i]} \mid n_{\text{train}} + n_{\text{val}} \le i < N\}.$$

Table 3 summarizes the resulting partition sizes.

Table 3: Dataset partitioning for evolutionary experiments.

| Benchmark | Source size | $\lvert\mathcal{X}_{\text{train}}\rvert$ | $\lvert\mathcal{X}_{\text{val}}\rvert$ | $\lvert\mathcal{X}_{\text{test}}\rvert$ | Total $\lvert\mathcal{X}\rvert$ |
|---|---|---|---|---|---|
| SWE-Bench-Verified | 500 | 36 | 4 | 60 | 100 |
| BrowseComp | 1,266 | 36 | 4 | 60 | 100 |
| GAIA-text | 103 | 37 | 4 | 62 | 103 |

## A.3 WorkBench

WorkBench (Styles et al., 2024) comprises 690 task instances spanning six functional domains: *Analytics* (120), *Calendar* (110), *CRM* (80), *Email* (90), *Multi-Domain* (210), and *Project Management* (80). Unlike the three benchmarks above, WorkBench exhibits significant domain imbalance, so we apply stratified sampling to ensure uniform domain coverage in the training and validation splits. For each domain, instances are shuffled with seed $s = 42$, and the first 10 are assigned to $\mathcal{X}_{\text{train}}$, the next 2 to $\mathcal{X}_{\text{val}}$, with all remaining instances forming $\mathcal{X}_{\text{test}}$. This yields 60 training, 12 validation, and 618 test instances. Table 4 summarizes the per-domain partition.

## A.4 Role of Each Split in Evolutionary Training

**Training split $\mathcal{X}_{\text{train}}$.** At each optimization step, the adaptive sampler draws a batch of $n$ instances from $\mathcal{X}_{\text{train}}$ via probability-weighted sampling, prioritizing queries that failed in the previous step while filling remaining slots from the full training pool. The resulting trajectories and evaluation outcomes constitute the forward-pass output and are

Table 4: WorkBench dataset partition via stratified sampling (seed $s = 42$).

| Domain | Source | $\lvert\mathcal{X}_{\text{train}}\rvert$ | $\lvert\mathcal{X}_{\text{val}}\rvert$ | $\lvert\mathcal{X}_{\text{test}}\rvert$ |
|---|---|---|---|---|
| Analytics | 120 | 10 | 2 | 108 |
| Calendar | 110 | 10 | 2 | 98 |
| CRM | 80 | 10 | 2 | 68 |
| Email | 90 | 10 | 2 | 78 |
| Multi-Domain | 210 | 10 | 2 | 198 |
| Project Management | 80 | 10 | 2 | 68 |
| **Total** | **690** | **60** | **12** | **618** |

routed to the optimizer for backward-pass analysis. The sampled subset varies across steps, but the candidate pool remains fixed.

**Validation split $\mathcal{X}_{\text{val}}$.** The identical, small-scale $\mathcal{X}_{\text{val}}$ is evaluated at every step using the current MAS configuration, providing a consistent overfitting signal. The optimizer receives the validation accuracy and its step-over-step delta as backward-pass input but has no access to validation trajectories, preventing it from fitting to validation instances.

**Test split $\mathcal{X}_{\text{test}}$.** Held out entirely during evolution. Neither the optimizer nor any training component observes test instances or outcomes. The test set is used exclusively for final evaluation of the evolved MAS.

### A.5 Reproducibility Notes

- A fixed random seed $s = 42$ governs both subset sampling and train/val/test partitioning, ensuring deterministic reproduction given the same source dataset. All curated data and code will be publicly released.
- Partitioning is performed once before training and remains fixed across all epochs, steps, and benchmarks.
- Split ratios and seed are configurable via command-line arguments (e.g., `--split-ratios 0.36 0.04 0.60` and `--seed 42`).
- Python's `random.Random(42).shuffle()` is the sole source of randomness in partitioning, with no external library dependencies, ensuring cross-platform determinism.

## B Baseline Details

We evaluate OptiMAS against baselines spanning two paradigms: hand-crafted multi-agent systems designed by human engineers, and automated frameworks that algorithmically discover or optimize agent architectures.

### B.1 Hand-crafted Multi-Agent Systems

- **SMoA** (Li et al., 2025): A general-purpose Sparsified Mixture of Agents framework that harnesses collaborative capabilities of multiple heterogeneous language models. To reduce the computational overhead typical of dense multi-agent configurations, SMoA incorporates dynamic response selection and early stopping mechanisms to sparsify information flows, balancing multi-agent synergy with execution efficiency.

- **Multi-Agent Debate** (Du et al., 2023): A general-purpose cooperative framework in which multiple independent language model agents address a problem simultaneously and iteratively refine their reasoning through successive rounds of structured peer critique. This process enables cross-verification of facts and systematic reduction of individual agent hallucinations.

- **SWE-Agent** (Yang et al., 2024b): An open-source software engineering agent framework designed to resolve real-world code defects within complex repositories. It introduces a custom Agent-Computer Interface (ACI) that provides optimized terminal interaction, repository navigation, and file-editing tools, simplifying the action space for LLM-driven code repair. In our experiments, we re-implement SWE-Agent atop our unified infrastructure while retaining its official prompts and tool configurations, ensuring both ease of integration and fair comparison. We use SWE-Agent as a domain-specialized baseline for *SWE-Bench-Verified*.

- **Tongyi-DeepResearch** (Team et al., 2025): An open-source agentic framework with a task-specific finetuned LLM backbone (30B total, 3.3B activated) developed by Tongyi Lab for long-horizon, deep information-seeking tasks. It employs an end-to-end agentic training paradigm that unifies continual pre-training on large-scale agentic interaction data with multi-turn reinforcement learning via a customized GRPO framework. At inference, it supports both a standard ReAct mode and an IterResearch-based test-time scaling mode. We use Tongyi-DR framework with general-purposed LLM backbone as

a domain-specialized baseline for *GAIA-text* and *BrowseComp*.

### B.2 Automated Agent Evolution Methods

- **ADAS** (Hu et al., 2024a): The Automated Design of Agentic Systems framework formalizes open-ended discovery of agent architectures in a code-level search space. A meta-agent iteratively proposes novel programmatic agentic components, which are then evaluated in isolated sandbox environments, progressively building an archive of discovered designs.

- **EvoAgent** (Yuan et al., 2025): An automated framework that extends single-agent systems into multi-agent collaborative networks through evolutionary operators including semantic mutation, crossover, and quality-based selection. EvoAgent generates a diverse population of specialized agent personas from a single user-provided template, removing the need for manual orchestration design.

- **DGM** (Zhang et al., 2025c): The Darwin Gödel Machine is a self-evolving system that unifies functional execution logic and self-modification routines into editable programs. Through iterative cycles of programmatic mutation, sandboxed evaluation, and archive-based exploration, DGM enables agents to continuously refine their own code and capabilities beyond their initial architectural boundaries.

## C $\mathcal{E}_0$ Configurations

This section documents the initial (epoch-zero) Multi-Agent System configurations used in our experiments on the all four benchmarks. Both configurations adopt a *single-agent* architecture $\mathcal{E}_0$ as the starting point for evolutionary optimization, comprising one general agent (`worker`) with no sub-agents. The evolutionary process may subsequently introduce additional agents, create new skills, modify prompts, and adjust tool permissions as the OptiMAS identifies structural or behavioral improvements. The initial configurations therefore represent the *minimal viable MAS* from which the optimizer begins its search.

### C.1 General $\mathcal{E}_0$ Configuration

#### C.1.1 Topology and Iteration Budget

The MAS consists of a single entry agent (`worker`) with no sub-agents. The iteration budget is set to 40 reasoning steps per phase and 100 total steps across all phases, providing sufficient room for multi-phase exploration, patch construction, and verification.

#### C.1.2 LLM/VLM Backbone

The primary reasoning backbones span from GPT-5-Nano (128K), Qwen-3.6-35B-A3B(262K), Gemin-3-Flash(1M) under different experimental settings. The same model is used for internal AI tools (e.g., `text_qa` for document understanding and context compression). The vision-language model for `image_qa` is GLM-4.6V-Flash under Qwen-3.6-35B-A3B setting, and GPT-4o-mini under GPT-5-Nano and Gemin-3-Flash settings.

#### C.1.3 Initial System Prompt

The initial system prompt establishes the agent's role and environment:

```
You are an autonomous agent that solves tasks
by using tools.
Today's date: {today_date}
## System protocols (mandatory)
The following protocols are fully loaded
below. You must follow them.
{system_protocols}
## Optional skills
You may load additional skills with the
'load_skill' tool if needed. Only each
skill's name and description appear below;
the full instruction body is returned in the
tool response when you call 'load_skill'.
{optional_skills}
```

The value in '{}' placeholders will be resolved upon MAS instantiation. The initial system prompt is intentionally minimal. It defines the agent's role and execution environment but does not prescribe any specific workflow, debugging strategy, or patch construction methodology. This design leaves the behavioral search space maximally open for the optimizer to populate through skill creation and prompt refinement. The system prompt also contains two template regions, `${system_protocols}` and `${optional_skills}`, into which the framework injects the system skill bodies and optional skill descriptions at runtime.

#### C.1.4 Initial Message Template

```
{task}
```

The initial message template consists solely of the `${task}` placeholder, which is replaced at runtime with the specific queries description directly

given by benchmark. No additional scaffolding or structured instructions are provided, leaving the agent to rely entirely on its system prompt and skills for guidance.

#### C.1.5 System Protocols

Based on the execution mechanism and strong coupled work control tools of our infrastructure, the foundational system protocol is inlined in default into the system prompt to ensure the Agent knows how to work and deliver solution upon completion.

```
name: phase-transition
description: Decide when to call phase_done_and_summarize versus task_done.
- Call 'phase_done_and_summarize' when there is remaining work or the context is getting long. This compresses history and continues execution.
- Call 'task_done' only when the task is fully complete.
```

#### C.1.6 Available Tools

The general-purpose $\mathcal{E}_0$ is equipped with 10 tools:

- **File operations**: `read_file`, `write_file`, `file_edit`, `grep_file_content`, `list_file_tree` for navigating, reading, and modifying artifacts in its own workspace.
- **Content analysis**: `text_qa` for LLM-powered question answering over long files with automatic sliding-window chunking.
- **Workflow control**: `load_skill`, `task_done`, `phase_done_and_summarize` for optional skill loading, task completion, and phase transitions.

#### C.1.7 Optional Skills

The initial configuration includes no optional skills. The OptiMAS may create and register optional skills during evolution to address recurring failure patterns.

### C.2 Specific $\mathcal{E}_0$ Configurations

Based on the general-purpose configurations of $\mathcal{E}_0$, some specific tools are required to ensure the corresponding task can be completed, otherwise the worse initial $\mathcal{E}_0$ performance could be necessary tools absence.

#### C.2.1 SWE-Bench $\mathcal{E}_0$

Specific tools for $\mathcal{E}_0$ on SWE-Bench-Verified benchmarking:

- **Shell execution** (1 tool): `shell_exec` for running commands (e.g., `pytest`, `git diff`, `python`) inside a Docker container with the repository's environment pre-configured.

#### C.2.2 BrowseComp $\mathcal{E}_0$

Specific tools for $\mathcal{E}_0$ on BrowseComp benchmarking:

- **Web and network** (3 tools): `search` (web search via Serper API), `crawl` (page content retrieval via Jina Reader), `download` (file downloading to workspace), enabling multi-step web research, source retrieval, and cross-referencing.

#### C.2.3 GAIA-text $\mathcal{E}_0$

Specific tools for $\mathcal{E}_0$ on GAIA-text benchmarking:

- **Web and network** (3 tools): `search` (web search via Serper API), `crawl` (page content retrieval via Jina Reader), `download` (file downloading to workspace), enabling multi-step web research, source retrieval, and cross-referencing.

#### C.2.4 WorkBench $\mathcal{E}_0$

The initial environment $\mathcal{E}_0$ for WorkBench equips the agent with 29 tools spanning six functional categories, mirroring the complete API surface of the WorkBench sandbox (Styles et al., 2024):

- **Calendar management** (5 tools): `create_event`, `delete_event`, `update_event`, `get_event_information_by_id`, and `search_events`. These provide full CRUD access over a calendar database, enabling the agent to schedule, modify, query, and remove events by ID, name, participant, or time range.
- **Email operations** (6 tools): `send_email`, `delete_email`, `forward_email`, `reply_email`, `get_email_information_by_id`, and `search_emails`. Beyond basic send and delete, the inclusion of `forward` and `reply` as first-class operations reflects WorkBench tasks that require multi-hop communication chains (e.g., forwarding a searched email to a looked-up contact).
- **Web analytics** (6 tools): `create_plot`, `total_visits_count`, `engaged_users_count`, `traffic_source_count`, `get_average_session_duration`, and `get_visitor_information_by_id`. This toolkit combines aggregation queries (visits,

engagement, traffic sources, session duration over date ranges) with a visualization tool, covering the analytical reasoning tasks in the benchmark.

- **Project management** (5 tools): `create_task`, `delete_task`, `update_task`, `get_task_information_by_id`, and `search_tasks`. Tasks are organized by boards (*Back end*, *Front end*, *Design*), lists (*Backlog*, *In Progress*, *In Review*, *Completed*), and assignees, requiring the agent to navigate structured project hierarchies.

- **Customer relationship management** (4 tools): `add_customer`, `update_customer`, `delete_customer`, and `search_customers`. The CRM toolkit supports multi-field filtering (name, email, product interest, status, contact dates) and enforces domain constraints on status and product categories, testing the agent's ability to respect enumerated value restrictions.

- **Company directory** (1 tool): `find_email_address` resolves employee names to email addresses. This cross-cutting utility is essential for multi-domain tasks where the agent must first look up a contact before performing calendar, email, or CRM operations—a common pattern in WorkBench's *multi_domain* category.

- **Agent control** (2 tools): `task_done` signals task completion, and `phase_done_and_summarize` enables phased execution with intermediate summaries. These are system-level primitives for the MAS coordination protocol rather than domain-specific APIs.

**Minimalist Initialization Policy.** The initialization of the agentic toolkits is intentionally governed by a *minimalist design principle*. Rather than seeding the ecosystem with domain-specific heuristics, specialized sequential workflows, or optional high-level skills, we equip each agent exclusively with generic foundational primitives essential for baseline operations (i.e., fundamental file manipulation, context routing, and standardized execution termination). This strict structural constraint maximizes the behavioral exploration space accessible to the evolutionary optimizer. Consequently, it guarantees that any observed trajectory optimizations and performance increments during the evolutionary training phase are strictly attributable to the optimizer's structural interventions rather than pre-engineered initial configurations.

**Orthogonal Tool-Isolation and Data Leakage Prevention.** To preserve evaluation integrity and ensure strict alignment with the underlying benchmarks, we enforce an orthogonal tool-isolation protocol across distinct task modalities. Specifically, we prohibit web search capabilities for the multi-agent systems (MAS) evaluated on **SWE-Bench-Verified**, while symmetrically blocking shell execution and programmatic code-generation tools for the MAS tested on **BrowseComp** and **GAIA-text**.

The structural necessity of this constraint stems from a telemetry vulnerability discovered during early-stage exploration. When unconstrained, the search-enabled MAS could heuristically locate the remote ground-truth data repositories via open-web queries. It would then attempt to bypass real-time reasoning by downloading and programmatically parsing the dataset files (e.g., extracting serialized labels from cloud-hosted tables via synthesized Python scripts). By restricting shell execution and coding toolkits in web-centric benchmarks, the agents are structurally blocked from processing production-grade data formats such as Parquet (frequently deployed on Hugging Face repositories), as our basic file-reading utilities lack native handlers for compressed binary schemas. This restriction forces the evolved MAS to perform authentic, zero-leakage information foraging and verification. Symmetrically, stripping web-search primitives from the code-centric **SWE-Bench-Verified** pipeline compels the agents to rely entirely on static repository comprehension and local unit-test validation, preventing any empirical contamination from external web-hosted patches.

## D Experiment Hyperparameters

This section details the numerical hyperparameters governing our evolutionary optimization framework. We categorize these parameters into three dimensional layers: *training loop* configurations (defining the temporal optimization boundaries), *adaptive sampling* coefficients (modulating the density-weighted batch construction). All parameters are maintained uniformly across all benchmark evaluations unless explicitly stated otherwise.

### D.1 Training Horizon and Optimization Boundaries

To accommodate variations in sequence length and task density across the evaluated benchmarks, we adopt the maximum number of *epochs* rather than a fixed step budget as the optimization termination criterion. This design choice prevents empirical bias stemming from heterogeneous dataset scales while holding the candidate batch size $n$ constant. For a designated training split $\mathcal{X}_{\text{train}}$, the step cardinality per epoch is formalized as follows:

$$\text{step} = \left\lceil \frac{|\mathcal{X}_{\text{train}}|}{n} \right\rceil . \tag{6}$$

Given the heavy computational requirements of long-horizon evolution, the optimization horizon is tightly bounded at exactly 10 epochs across all four benchmarks, guaranteeing that the evolving MAS exhaustively explores the complete support dataset $\mathcal{X}_{\text{train}}$ ten times. To preserve structural equity during comparison, all search-based and evolutionary baseline counterparts (e.g., DGM (Zhang et al., 2025c), which mandates an offspring generation policy of two candidate MAS structures per cycle) are configured to execute for precisely ten generations, with all remaining parameters adhering strictly to their officially calibrated settings.

### D.2 Adaptive Sampling Parameters and Design Rationales

As formulated in Algorithm 1, the adaptive sampling subsystem introduces a localized, priority-weighted batch construction mechanism. At each gradientless optimization step, a designated fraction $\phi$ of the batch capacity is reserved exclusively to re-execute queries that failed during the immediate historical iteration. The remaining allocation slots are dynamically populated via probability-weighted sampling across the entire training split $\mathcal{X}_{\text{train}}$. Sampling weight updates are parametrically controlled by multiplicative scaling factors $\alpha$ and $\beta$, bounded within the closed interval $[p_{\min}, p_{\max}]$ to mitigate long-term distributional drift. To prevent catastrophic weight saturation, a periodic hard reset interval $\tau$ is enforced. The specific parameter calibrations are consolidated in Table 5.

The calibration of these hyperparameter values is dictated by structural design rationales aimed at stabilizing trajectory exploration. The clamping boundaries ($p_{\min} = 0.2, p_{\max} = 2.0$) restrict the maximum probability discrepancy between consistently erroneous tasks and successfully resolved tasks to a factor of $\times 10$, ensuring the sampling distribution remains non-degenerate and safe from saturation. Correspondingly, the step-wise modifiers $\alpha = 0.6$ and $\beta = 1.2$ are mathematically mirrored to ensure that the lower probability boundary is systematically reached after five consecutive successes, while the upper ceiling is attained after five consecutive execution failures.

Crucially, as established by the global framework ablations in Section 4, the macroscopic trajectory improvements are robust to marginal variations in these auxiliary local coefficients. The validation of the global adaptive mechanism, coupled with the hypothesis-driven long-term memory module, fully demonstrates structural stability without requiring hyperparameter grid-searches.

### D.3 Hyperparameter Synthesis

Table 6 summarizes the key global hyperparameter configurations implemented across our entire framework infrastructure.

## E Infrastructure

This section describes the infrastructure that supports the our configuration-based agent instantiate and agent's interaction with the environment.

Our framework adopts an industry-standard client–server architecture grounded in the *Model Context Protocol* (MCP), providing a modular, extensible, and deployment-ready foundation for autonomous software engineering agents.

### E.1 Infrastructure Overview

The system is organized as a decoupled two-tier architecture comprising an **Agent Client** and a extensible **Tools Server**, communicating via the Model Context Protocol over Streamable HTTP transport. This separation confers several advantages:

- **Deployment Flexibility.** The agent client (hosting the LLM reasoning loop) and the tool server (hosting environment interactions) can be deployed on the same machine or distributed across heterogeneous infrastructure. This enables scenarios such as running the agent on GPU-equipped nodes while the tool server operates on standard compute instances co-located with development environments.
- **Multi-Tenancy and Isolation.** The server manages concurrent sessions with per-session

Table 5: Parameter configurations for the adaptive sampling subsystem.

| Hyperparameter | Value | Mathematical Functional Description |
|---|---|---|
| $\phi$ (Retry Ratio) | 0.5 | Mass allocation ratio reserved for historic failed queries |
| $\alpha$ (Decay Factor) | 0.6 | Multiplicative discount multiplier for successful queries |
| $\beta$ (Boost Factor) | 1.2 | Multiplicative scaling multiplier for failed queries |
| $p_{\min}$ (Weight Floor) | 0.2 | Lower-bound boundary for sample probability clamping |
| $p_{\max}$ (Weight Ceiling) | 2.0 | Upper-bound boundary for sample probability clamping |
| $\tau$ (Reset Interval) | 2 epochs | Temporal stride between global uniform weight resets |
| $w_0$ (Initial Weight) | 1.0 | Uniform initial scalar assigned to all dataset instances |

Table 6: Consolidated global hyperparameter specifications.

| Operational Category | Hyperparameter Layer | Calibrated Value |
|---|---|---|
| Training Loop Boundaries | Max Optimization Horizon $E$ | 10 epochs |
| | Batch Capacity $B$ | 10 |
| Adaptive Sampling Subsystem | Retry Ratio $\phi$ | 0.5 |
| | Decay Factor $\alpha$ | 0.6 |
| | Boost Factor $\beta$ | 1.2 |
| | Weight Floor $p_{\min}$ | 0.2 |
| | Weight Ceiling $p_{\max}$ | 2.0 |
| | Global Reset Interval $\tau$ | $2 \times \lceil \lvert\mathcal{X}_{\text{train}}\rvert / n \rceil$ steps |

workspace isolation, supporting parallel agent executions with no cross-contamination. Each session maintains its own file system scope, process state, and lifecycle management.

- **Protocol Compliance.** By building on MCP, our infrastructure ensures interoperability with the broader ecosystem of MCP-compatible tools and clients, facilitating integration with third-party services and framework.
- **Security.** The server enforces API key authentication, session validation (via HMAC-based session identifiers), and workspace sandboxing, ensuring that agent actions are confined to authorized scopes.

The agent client implements a ReAct-style reasoning loop that generates tool calls dispatched to the MCP server. Tool schemas are dynamically discovered at session initialization via the protocol's `tools_accessible` capability, enabling the agent to adapt to varying server configurations without client-side modification.

### E.2 Tool Server Capabilities

The tool server provides a comprehensive suite of operations organized into five functional categories, collectively enabling the agent to perform long horizon, multi-step agentic tasks autonomously.

**File Operations.** The file operations module provides fine-grained control over the agent's workspace file system:

- *Reading*: full-file and line-range reading with line number annotations, chunked reading for large-file question answering, and image reading with base64 encoding for vision-language model integration.
- *Writing*: full-file creation and overwriting, append mode, and line-level insertion for surgical modifications.
- *Editing*: targeted insertion, replacement, and deletion operations with line-level precision, as well as block-level replacement for refactoring workflows.
- *Search and Navigation*: content-level regular expression search across the workspace, recursive file tree listing with configurable depth, and file move/rename/delete operations.
- *Rich Format Support*: transparent extraction of textual content from structured document formats (PDF, DOCX, PPTX, XLSX, CSV, HTML) via an integrated document reader, enabling the agent to process heterogeneous file types uniformly.

**System Execution.** The shell execution module supports both *local* and *containerized* execution modes. In local mode, commands execute within the session workspace directory under permission identification. In containerized mode, which is

used for reproducible evaluation on benchmarks such as SWE-bench, commands are transparently proxied into a Docker container with the appropriate environment pre-activated (e.g., conda environments with correct dependency versions). This dual-mode design allows the same agent logic to operate across development and evaluation contexts without modification.

**Network Operations.** The framework integrates two complementary information retrieval services:

- *Web Search*: powered by the Serper API, supporting multiple search modalities (web, images, video, academic literature). A server-side token-bucket rate limiter ensures compliance with API rate constraints under concurrent multi-agent workloads.
- *Web Crawling*: powered by the Jina Reader API, providing structured extraction of web page content in Markdown format. The crawler supports batch URL processing with automatic file persistence to the agent's workspace, and includes retry logic with configurable timeouts for robustness against transient network failures.
- *File Download*: asynchronous streaming download of remote resources (including images, datasets, and documentation) with content validation, size limits, and automatic MIME type detection.

These capabilities enable the agent to gather external information, such as API documentation, library changelogs, issue discussions, and code examples, as part of its problem-solving workflow, significantly expanding the scope of tasks it can address autonomously.

**Coding Operations.** Dedicated coding tools provide language-aware support for the software engineering workflow:

- *Syntax and Compilation Diagnostics*: static analysis of Python source code for syntax errors and compilation issues, returning structured feedback with line numbers, error types, and source excerpts.
- *Function-Level Testing*: in-process execution of test functions against agent-written code with configurable timeouts, enabling rapid feedback cycles without full test suite overhead.

### E.3 Containerized Execution for Reproducible Evaluation

For benchmark evaluation, the framework provides seamless Docker container integration. Given a benchmark instance identifier (e.g., a SWE-bench problem ID), the system automatically:

1. Resolves and pulls the corresponding Docker image containing the exact repository snapshot and dependency environment.
2. Extracts the working directory from the container image to the host file system for workspace initialization.
3. Starts a new container with the workspace bind-mounted, enabling the agent to read and modify files via the file operations module while executing commands inside the container's pre-configured environment.
4. Manages the container lifecycle (creation, monitoring, cleanup) to prevent resource leaks in batch evaluation scenarios.

This approach ensures that the agent operates in an environment identical to the one used by human developers for the target repository, eliminating environment mismatch as a source of evaluation noise.

### E.4 Middleware Architecture

The server employs a layered middleware pipeline that intercepts all tool invocations, providing cross-cutting concerns without polluting individual tool implementations:

- **Authentication Middleware**: validates API keys on every request, preventing unauthorized access.
- **Session Middleware**: resolves and validates session identifiers, scopes tool invocations to the correct workspace, and maintains session liveness tracking for automatic expiration and cleanup.
- **Logging Middleware**: records structured invocation logs (tool name, arguments, success/failure, duration) in JSONL format, enabling fine-grained performance analysis and debugging of agent behavior.

The middleware chain executes in a fixed, deterministic order (authentication $\rightarrow$ session resolution $\rightarrow$ logging $\rightarrow$ tool execution), ensuring consistent behavior across all tool types.

### E.5 Scalability and Concurrency

The infrastructure is designed for concurrent multi-agent execution, which is essential for training workflows where multiple agent instances solve different problems in parallel:

- **Asynchronous I/O**: the server is built on an ASGI framework with full asynchronous support, enabling efficient handling of concurrent tool invocations without thread-per-request overhead.
- **Session Isolation**: each agent instance operates in a unique session with isolated workspace, process context, and container binding, preventing interference between concurrent executions.
- **LLM Concurrency Control**: a process-wide semaphore limits concurrent LLM API calls, preventing GPU or API rate limit saturation when multiple agents share a backend model deployment.
- **Rate-Limited External APIs**: outbound calls to external services (web search, crawling) are governed by token-bucket rate limiters, ensuring compliance with provider rate limits even under high-concurrency workloads.

### E.6 Generality and Extensibility

The framework's modular design supports straightforward extension to new task domains and tool ecosystems:

- **Plugin-Style Tool Registration**: new tools are added by implementing a registration function and declaring parameter schemas via standard JSON Schema annotations. No modifications to the core framework or agent logic are required.
- **Multi-Domain Applicability**: the tool suite is not specific to any single benchmark or programming language. The same infrastructure supports software engineering tasks, mathematical reasoning, question answering, and web-based research, demonstrating the framework's generality.
- **Configuration-Driven Behavior**: agent capabilities, tool access, iteration budgets, and context limits are all governed by declarative YAML configuration, enabling rapid experimentation with different agent profiles without code changes.
- **Hierarchical Multi-Agent Support**: the architecture supports multi-agent delegation, where a parent agent can spawn sub-agents with independent LLM instances, tool sets, and conversation contexts, enabling sophisticated task decomposition strategies.

In summary, the infrastructure layer provides an industrial-grade, protocol-compliant foundation that abstracts the complexity of environment interaction behind a clean tool interface. This enables the agent's reasoning layer to focus on high-level problem solving while the infrastructure ensures reliable, secure, and scalable execution of its actions.

### E.7 Context Management Mechanism

A central challenge in deploying LLM-based agents for long-horizon agentic tasks, such as web searching and software engineering, is *context window management*. The finite context length of the underlying language model imposes an inherent upper bound on the amount of information available during each reasoning step. As the agent interacts with tools, accumulates observations, and formulates plans, the conversation history grows monotonically, and naive truncation risks discarding critical state information. We address this challenge through a principled, multi-tier context lifecycle management framework that orchestrates *proactive* (agent-initiated) and *reactive* (system-enforced) phase transitions alongside *semantic* and *structural* compression mechanisms, ensuring that the agent retains the maximal amount of task-relevant information within its capacity constraints.

#### E.7.1 Context Lifecycle Architecture

We model the evolution of the agent's conversational context as a managed lifecycle governed by a three-tier capacity envelope: a **soft limit** $C_{\text{soft}}$, a **hard limit** $C_{\text{hard}}$, and a **force limit** $C_{\text{force}}$ (with $C_{\text{soft}} < C_{\text{hard}} < C_{\text{force}}$). Each tier triggers progressively stronger interventions, forming a graceful degradation cascade rather than an abrupt cutoff.

Formally, let $\mathcal{H}_t = [m_1, m_2, \ldots, m_t]$ denote the conversation history at step $t$, and let $S(\mathcal{H}_t) = \sum_{i=1}^{t} |m_i|$ be the estimated payload size (measured in characters, with heuristic token approximation $\hat{T} \approx S/3 + 4|\mathcal{H}_t|$). At each iteration of the ReAct loop, the system evaluates $S(\mathcal{H}_t)$ against the three thresholds and applies the corresponding intervention from the pipeline described below.

#### E.7.2 Three-Tier Context Capacity Envelope

The context management pipeline is evaluated *sequentially* at the beginning of each agent iteration,

following a carefully designed ordering that ensures compression precedes restriction, and restriction precedes notification:

**Tier 1: Force-Limit Compression ($S > C_{\text{force}}$).** When the context payload exceeds $C_{\text{force}}$, the system initiates *forced LLM-based semantic compression*. This is the most aggressive intervention: an auxiliary LLM pass summarizes the middle segment of the conversation history (see §E.7.4), replacing verbose tool I/O traces with a structured, information-preserving summary. If LLM compression continually fails (e.g., due to API errors), the system falls back to *tail-drop compaction* (§E.7.5), which retains the most recent messages within a target budget.

**Tier 2: Hard-Limit Tool Restriction ($S > C_{\text{hard}}$).** If the context exceeds $C_{\text{hard}}$, the system restricts the agent's action space to a minimal set of *exit tools*: `phase_done_and_summarize` (proactive phase transition) and `task_done` (task completion). All other tools, including file operations, search, and code editing, are *dynamically masked* from the LLM's tool schema for that iteration. A structured guard prompt is injected to inform the agent of the restriction and instruct it to either summarize progress or submit a final answer. This prevents the agent from generating tool calls that would further inflate the context while providing a clear escape path.

**Tier 3: Soft-Limit Advisory Warning ($S > C_{\text{soft}}$).** When the context exceeds $C_{\text{soft}}$ but remains below $C_{\text{hard}}$, a non-restrictive advisory warning is injected into the conversation. The agent retains full tool access but is prompted to proactively call `phase_done_and_summarize` to preserve progress before harder limits are reached. This early-warning mechanism leverages the LLM's instruction-following capabilities to encourage self-regulated context management, reducing the frequency of forced compressions.

**Ordering Rationale.** The sequential evaluation order is critical: by applying compression before restriction, we maximize the probability that the agent can continue productive work after a force-limit trigger. Applying restriction after compression ensures that tool masking only activates when the context is genuinely intractable, avoiding premature capability reduction.

#### E.7.3 Proactive Phase Transition: Agent-Initiated Context Reset

The primary mechanism for *proactive* context management is the `phase_done_and_summarize` tool, which the agent can invoke at any point during execution. This tool implements a structured *phase transition protocol* that performs the following operations atomically:

1. **Auto-Compression of Current Phase.** Before discarding the conversation history, the system extracts all messages from the first assistant response onward and passes them through the LLM-based compression pipeline (§E.7.4) to generate a detailed, structured summary of the completed phase's work. This ensures that no critical information is lost during the transition.

2. **Phase Resume Message Construction.** The system constructs a comprehensive *phase resume* message that aggregates three information sources: (a) the agent's self-authored `progress_report` summarizing key findings and decisions; (b) the agent's `remaining_tasks` specifying concrete next steps; and (c) the auto-compressed work log from the previous phase. Additionally, the system re-injects any preloaded skill protocols to maintain behavioral consistency across phases.

3. **Context Reset and Reconstruction.** The conversation history is cleared entirely, and a fresh system prompt is rebuilt—including updated workspace state, followed by the phase resume message as the initial user input. The phase-local iteration counter is reset to zero while the *global* iteration counter is preserved, preventing infinite phase cycling.

4. **Trajectory Persistence.** A phase boundary event is emitted to the trajectory logging system, ensuring complete observability and enabling offline analysis of phase transition dynamics.

This design follows the principle of *structured forgetting*. Rather than retaining raw conversation traces indefinitely, the agent periodically distills its accumulated knowledge into a compact, high-fidelity summary and continues from a clean slate (Ye et al., 2025a). The dual-counter mechanism (phase-local and global) provides both flexibility (resetting within-phase budgets) and safety

(hard global budget), preventing degenerate behavior where the agent repeatedly transitions phases without making progress.

#### E.7.4 LLM-Based Semantic Compression

The semantic compression module implements a three-stage pipeline that preserves the structural and informational integrity of the conversation while achieving significant size reduction:

**Stage 1: Conversation Segmentation.** The conversation history $\mathcal{H}$ is partitioned into three semantically meaningful segments:

- **Preamble** $\mathcal{P}$: All messages preceding the first assistant response (typically the system prompt and initial user task). These are preserved verbatim as they define the task specification and agent identity.
- **Compressible Segment** $\mathcal{C}$: Messages from the first assistant response up to (but not including) the last assistant response. This segment contains the bulk of tool interactions, intermediate reasoning, and accumulated observations that are candidates for compression.
- **Last Round** $\mathcal{L}$: The most recent assistant response and any subsequent tool/user messages. This segment is preserved verbatim to maintain the agent's immediate working context and avoid disrupting in-progress reasoning chains.

This segmentation ensures that compression targets the information-dense middle of the conversation while preserving both the foundational task context and the agent's active working state.

**Stage 2: Structured Summarization.** The compressible segment $\mathcal{C}$ is flattened into a linearized text representation, with tool result bodies truncated to a configurable maximum length to prevent individual oversized observations from dominating the compression input. This linearized representation is then passed to a dedicated LLM call guided by a structured compression prompt that mandates the following sections in the output summary:

- *Task Understanding*: the agent's interpreted goal and any scope refinements;
- *Completed Work*: chronological enumeration of actions with results, including exact file paths, function names, and data samples;
- *Key Decisions & Reasoning*: design choices, trade-offs, and hypothesis testing outcomes;
- *Current Working State*: files modified, configurations set, and the agent's position in its workflow;
- *Pending Work & Plan*: remaining tasks with priority ordering;
- *Errors, Blockers & Workarounds*: failed approaches with root causes to prevent retry;
- *Critical Data & References*: exact identifiers, paths, and verbatim snippets needed for continuation.

This structured format is designed to maximize information density while providing the agent with clear organizational cues for retrieval during subsequent reasoning.

**Stage 3: History Reconstruction.** The compressed history is assembled as $\mathcal{H}' = \mathcal{P} \oplus [\text{compressed summary}] \oplus \mathcal{L}$, where the compressed summary is injected as a single user message annotated with a marker indicating the number of messages that were summarized. A subsequent system notification informs the agent that earlier raw tool traces have been absorbed into the summary, calibrating its expectations about available context.

#### E.7.5 Tail-Drop Compaction

As a complementary fallback mechanism, tail-drop compaction provides a deterministic, non-LLM-dependent compression strategy. When invoked, it retains the system message (if present) and greedily accumulates messages from the *most recent* backward, stopping when the cumulative size exceeds the target budget $B_{\text{target}} = 0.7 \times C_{\text{hard}}$. This ensures that the agent retains its most recent context—which is typically the most relevant for continued execution—while discarding older messages that are more likely to have been superseded by subsequent actions.

After compaction, a system notification is appended informing the agent that the context window was compacted and advising against re-reading large files unless necessary. This lightweight feedback loop helps the agent adapt its behavior to the reduced context availability.

#### E.7.6 Auxiliary Context Guards

Beyond the three-tier capacity envelope, the framework incorporates several auxiliary guards that interact with the context management system:

Table 7: Context Management hyperparameters on LLM Backbones.

| LLM Backbone | Context Limit | soft limit $C_{\text{soft}}$ | hard limit $C_{\text{hard}}$ | force limit $C_{\text{force}}$ | Tool Output Limit |
|---|---|---|---|---|---|
| GPT-5-Nano | 128K | 126K | 127K | 128K | 20K |
| Qwen3.6-35B-A3B | 262K | 260K | 261K | 262K | 20K |
| Gemini-3-Flash | 1M | 970K | 980K | 990K | 80K |

**No-Tool-Call Streak Detection.** When the agent produces consecutive text-only responses without invoking any tool (indicating potential "stuck" behavior), an escalating sequence of nudge prompts is injected. After a configurable number of consecutive text-only turns $k_{\max}$, the system restricts the agent's tool set to `phase_done_and_summarize` only, forcing a phase transition that resets the context and provides the agent with fresh instructions.

**Phase-Local Iteration Budget.** Near the end of each phase's iteration budget, the system proactively restricts the available tools to phase transition and task completion tools, preventing the agent from initiating new long-running operations that cannot be completed within the remaining budget.

**Global Iteration Tail Guard.** When the global iteration counter approaches the maximum, the system restricts tools to task completion only (`task_done` or `finalize_optimization`), ensuring that the agent submits a result rather than exhausting its entire compute budget without output.

**Tool Output Truncation.** Individual tool results exceeding a configurable character limit are truncated with tool-specific guidance messages (e.g., suggesting the use of line-range parameters for file reading or more selective search queries). This proactive truncation prevents single large tool observations from consuming a disproportionate share of the context budget and provides actionable feedback that helps the agent adopt more efficient information-gathering strategies in subsequent iterations.

#### E.7.7 Implementation

For the our experimental implementation, we keep fixed parameters for the context management mechanism across all four benchmarks as shown in Table 7

### E.8 Discussion

Our context management framework differs from prior work in several key respects. First, the three-tier capacity envelope provides *graduated* interventions rather than binary truncation, giving the agent maximum opportunity to self-manage its context before system-level overrides activate. Second, the proactive phase transition mechanism transforms context management from a purely system-side concern into a *collaborative* protocol between the agent and the framework: the agent is encouraged (via soft-limit warnings and skill protocols) to strategically partition its workflow into phases, each with a clean context slate and a distilled summary of prior work. Third, the structured compression prompt ensures that the compression process is guided by task-relevant organizational principles, producing summaries that are directly actionable by the agent rather than generic text condensations.

Together, these mechanisms enable the agent to tackle complex, multi-step software engineering tasks that far exceed the native context window of the underlying LLM, while maintaining coherent state tracking and avoiding the information loss that characterizes simpler truncation-based approaches.

## F OptiMAS Design

This section provides a comprehensive description of our proposed **OptiMAS** that drives the backward pass of our evolutionary training loop. The OptiMAS is itself an LLM-based agent that operates within the same ReAct framework as the evolutionary MAS it optimizes. It receives structured performance feedback from the forward pass (training batch diagnostics, validation signals, and full agent trajectories), analyzes failure patterns through a principled evidence-first methodology, and applies targeted interventions to the Multi-Agent System (MAS) configuration. A central design principle is *domain agnosticism*: the OptiMAS's reasoning machinery, skill library, and intervention taxonomy are independent of the downstream task domain, enabling the same OptiMAS to drive evolution across code engineering, web research, tool orchestration, and other long-horizon agentic benchmarks.

### F.1 OptiMAS Architecture Overview

The OptiMAS is essentially developed based on ReAct agent, inheriting all infrastructure capabilities, tool calling, context management, phase transitions, and skill loading, while introducing optimization-specific tools and a structured backward-pass workflow. At each training step, the OptiMAS receives:

- **Training batch diagnostics**: a structured report containing per-query outcomes (success/failure, scores), per-agent efficiency metrics (rounds, phases, tool call distributions, skill usage), and batch-level aggregates (accuracy, wall time).
- **Validation signal**: the current validation accuracy and its delta from the previous step, serving as an overfitting guard over the fixed validation set.
- **Previous optimization summary**: a textual record of the prior step's interventions, enabling continuity across optimization steps.
- **Full trajectory artifacts**: per-query directories containing structured audit logs (`tool_calls.jsonl`), full ReAct conversation traces (per-agent, per-phase JSONL files), evaluation diagnostics, and workspace artifacts.

The OptiMAS processes this information with four typical workflows, *Review & Hypothesize*, *Evidence Collection*, *Intervention*, and *Finalization*, guided by a hypothesis-driven long-term memory mechanism that facilitate cross-step experience condensation and ensures every modification is traceable and attributively grounded.

### F.2 Prompt Architecture

The OptiMAS's prompt architecture comprises three interconnected components that collectively define its reasoning behavior.

#### F.2.1 System Prompt

The system prompt establishes the OptiMAS's identity, workflow structure, and decision-making principles. It begins by defining the OptiMAS's domain-agnostic role:

```
You are OptiMAS, a general-purpose agent
that improves Multi-Agent Systems (MAS).
You are domain-agnostic:  the MAS you
optimize may target web research, code
engineering, mathematical reasoning, GUI
automation, multimodal analysis, or any
other long-horizon task...
```

The system prompt encodes the following critical components:

**Hypothesis-Driven Workflow.** The OptiMAS follows a structured six work modes: (1) review optimization history and the persistent hypothesis ledger; (2) gather evidence from trajectories using an audit-log-first discipline; (3) formulate or update hypotheses with explicit observation, root cause, action, expected outcome structure; (4) choose an intervention from the leveled action space; (5) apply changes to the MAS configuration; (6) validate and finalize.

#### F.2.2 Backward Pass Task Prompt

The backward pass template is instantiated at each optimization step with dynamically computed fields:

- `{progress}`: epoch and step counters indicating the current position in the training schedule.
- `{validation_metrics}`: a formatted table presenting the current validation accuracy, accuracy delta.
- `{prev_summary}`: the optimization summary from the previous step.
- `{training_batch_diagnostics}`: a comprehensive report including batch-level overview, per-query result tables (with scores, wall times, agent behavior summaries), and per-agent efficiency breakdowns (ReAct rounds, phases, tool call distributions, skill usage, delegation patterns).
- `{artifact_navigation}`: workspace-relative paths to trajectory artifacts, evaluation diagnostics, and MAS configuration files.

**Adaptive Sampling Awareness.** The backward pass prompt explicitly informs the OptiMAS about the adaptive sampling strategy: some queries in each training batch are *resampled* from the previous step's failures, enabling direct verification of whether the last optimization intervention was effective. The prompt instructs the OptiMAS to prioritize resampled failures as they represent "strong direct evidence that your previous intervention was insufficient."

#### F.2.3 Initial Message Prompt

The initial message template provides the entry point for the OptiMAS's ReAct loop, incorporating the task description (backward pass prompt), preloaded skill protocols. This template uses the standard `${task}` placeholder mechanism shared with worker agents, ensuring consistency across the agent hierarchy.

### F.3 Skill Library

The OptiMAS employs a rich library of modular skills organized into three categories, *system protocols*, *preloaded protocols*, and *optional skills*, following the same skill architecture used by worker agents but specialized for the optimization task.

#### F.3.1 System Protocols (Always Active)

These skills are inlined into the system prompt and define the OptiMAS's foundational operating contracts:

**Handle-Workspace.** Defines conventions for workspace navigation, file operations, and artifact management. Includes detailed guidance on using each available tool (file reading, writing, editing, searching, and document understanding) with OptiMAS-specific best practices.

#### F.3.2 Preloaded Protocols (Persistent Across Phases)

These skills are injected into the initial user message and re-injected after each phase transition, ensuring persistence across context resets:

**Hypothesis Memory.** The core knowledge management protocol. Defines the structure and lifecycle of optimization hypothesis long-term memory:

```
### H<N> [status] (Step <created>)
- **Category**: positive / negative
- **Observation**: <specific behavior with evidence>
- **Root cause**: <why this happens>
- **Action**: <MAS change made or planned>
- **Expected outcome**: <measurable prediction>
- **Evidence**:
- Step <X>: <observation>
```

The memory enforces a rigorous status lifecycle, with explicit transition rules that require trajectory-level evidence for status changes. This prevents the OptiMAS from making unfounded generalizations from aggregate accuracy metrics alone. The hypothesis file persists on disk across all optimization steps, serving as the OptiMAS's long-term memory.

**Phase Transition.** Guides the OptiMAS's decisions about when to use `phase_done_and_summarize` versus `task_done`, with specific instructions for writing informative progress reports and persisting analysis findings to disk before phase boundaries.

#### F.3.3 Optional Skills (Loaded On Demand)

These skills are available for the OptiMAS to load when their specific expertise is needed:

- **Trajectory Analysis.** The most comprehensive skill, providing a universal protocol for analyzing agent trajectories.
- **OptiMAS Skill Design.** Provides the theory and practice of the three-tier skill system (system / preload / optional).
- **OptiMAS MAS Evolve.** Encodes the universal theory of MAS architecture evolution.
- **OptiMAS Create Sub-Agent.** A comprehensive recipe for adding sub-agents to the MAS.
- **OptiMAS Toolkits.** Detailed guidance on the available tool suite $\mathbb{T}$.
- **OptiMAS Investigator.** Defines a read-only investigator sub-agent pattern for deep trajectory analysis on complex failing queries.

### F.4 Tool Suite

The OptiMAS has access to a curated set of tools that balance analytical capability with safety constraints.

#### F.4.1 MCP Server Tools

- `read_file`: Read MAS configuration files, trajectory logs, and evaluation artifacts with optional line-range selection.
- `write_file`: Create or modify MAS configuration files, prompts, skills, and working notes.
- `file_edit`: Targeted line-level editing for surgical modifications to existing files.
- `grep_file_content`: Regular expression search across the workspace for pattern detection in trajectories and configurations.
- `list_file_tree`: Navigate the MAS configuration structure and trajectory artifact directories.

#### F.4.2 Internal AI Tools

- `text_qa`: LLM-powered document understanding with sliding-window processing for large trajectory files. Used for grounded analysis of specific trajectory segments after the audit log has been reviewed directly.
- `image_qa`: Vision-language model integration for analyzing visual artifacts (e.g., MAS crawled artifacts during rollout).

#### F.4.3 Workflow Tools

- `load_skill`: Dynamically load optional skills on demand, enabling the OptiMAS to access specialized knowledge only when needed.
- `phase_done_and_summarize`: Trigger a phase transition with auto-compression, enabling the OptiMAS to work on complex optimization tasks that exceed a single context window.
- `task_done`: Mark the optimization step as complete.

#### F.4.4 Optimization-Specific Tool: `finalize_optimization`

The `finalize_optimization` tool is the OptiMAS's delivery mechanism. Upon invocation, it executes a multi-stage validation pipeline before accepting the optimization step. The programmatic completeness check of Plan oriented short-memory is implemented in this stage. If any stage fails, detailed error messages are returned to the OptiMAS, which can fix the issues and re-invoke `finalize_optimization`. This iterative validation loop ensures that every delivered optimization step produces a structurally valid, deployable MAS configuration and the OptiMAS working consistently without forgetting and hallucination as well.

### F.5 Hypothesis-Driven Long-Term Memory

A distinguishing feature of our OptiMAS is its *hypothesis-driven long-term memory* system, implemented through two persistent artifacts:

**Hypothesis.** The data-based structural hypothesis persists across all optimization steps and epochs, serving as the OptiMAS's cumulative knowledge base. Each hypothesis records an observation, root cause analysis, chosen action, concrete intervention, expected outcome, and evolving evidence trail. Hypotheses cannot be confirmed without trajectory-level evidence, and cannot be refuted based on aggregate accuracy metrics alone.

### F.6 Domain Adaptability

The OptiMAS's domain-agnostic design is achieved through a strict separation of concerns:

- The *universal optimization machinery*: Hypothesis management, trajectory analysis protocols, failure taxonomy, action space, skill design theory, MAS architecture evolution, are all encoded in domain-agnostic skills and prompts.
- The *backward pass template* dynamically injects evaluation guidance based on the benchmark type detected from the batch report, without modifying the universal template structure.

This architecture enables the same OptiMAS configuration to drive evolution across heterogeneous benchmarks, from code engineering to web research to long reasoning too orchestration, with the core optimization methodology remains invariant.

## G Case Study

**Evolutionary MAS on SWE-Bench-Verified.** To investigate the macro-level behavioral optimization of the evolving MAS, we trace the structural mutations captured in Figure 5 at the final evolution ($\mathcal{E}^*$). The results demonstrates that the OptiMAS transform single agent in favor of granular role specialization. Specifically, the OptiMAS instantiates an architectural core on the *Worker* agent, expanding its repository to eight highly specialized meta-skills, including *logic preservation audit* and *forced delegation gates*. Concurrently, resource-intensive sandbox tracking and post-patch validation are decoupled into localized execution loops managed by the *Refiner*, *Reproducer* and *Verifier* sub-agents, respectively. Rather than relying on fragile localized prompt engineering, these emergent architectural workflows, distinct sub-agent delegation cascades, and targeted skill discovery runs validate that our evolutionary framework systematically acquires the structural rigor required to navigate complex code repositories.

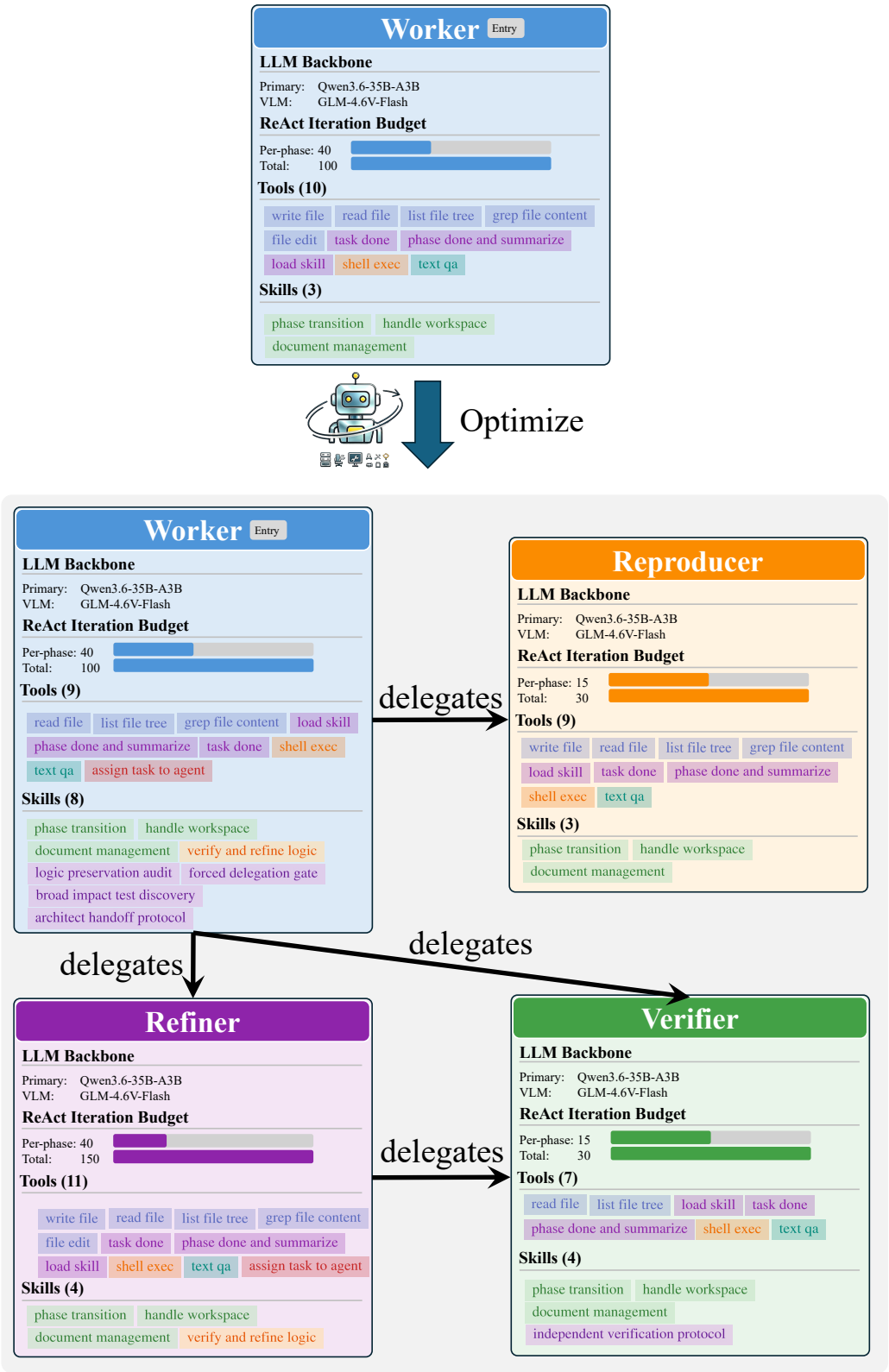


Figure 5: **Architectural Topography of the Evolved Multi-Agent System (MAS) on SWE-Bench-Verified.** This schematic contrasts the initial configuration $\mathcal{E}$, against the evolved architecture ($\mathcal{E}^*$). Through our OptiMAS optimization, the system undergoes severe structural transformation: transitioning from a standard flat-worker scheme into a modular, hierarchical coalition composed of specialized agents (*Worker*, *Reproducer*, *Refiner*, and *Verifier*). Each sub-agent customized ReAct iteration budgets, and differentiated, priority-driven tool and skill allocations.