AUTOGENESIS: A SELF-EVOLVING AGENT PROTOCOL

Anonymous authors
Paper under double-blind review

ABSTRACT

LLM-based agent systems can adapt by revising prompts, constructing tools, and changing other runtime resources. Managing these heterogeneous updates requires a shared lifecycle for identifying targets, assessing candidates, and retaining accepted versions, complementing communication protocols such as A2A and MCP. We introduce **AUTOGENESIS PROTOCOL (AGP)**, which decouples what evolves from how updates are proposed. Its Resource Substrate Protocol Layer (RSPL) represents prompts, agents, tools, environments, and memory as protocol-registered resources[1] with explicit state and versioned interfaces. Its Self-Evolution Protocol Layer (SEPL) specifies candidate proposal, assessment, and commit, with auditable histories and support for restoring retained resource states. We instantiate AGP in **AUTOGENESIS SYSTEM (AGS)**, a self-evolving multi-agent system that retrieves, instantiates, and refines resources during execution. Across reasoning, tool-use, and coding benchmarks, AGS improves over its non-evolving configuration, and repeated reasoning runs report the accompanying output-token cost. Execution cases show a shared update lifecycle across all five resource types.

1 INTRODUCTION

A long-running agent may repair a tool, revise an instruction, or change which memories survive compaction. Beyond generating a better answer, these adaptations require identifying the resource, evaluating the candidate, deciding when it becomes active, and retaining its history. A revised tool may change the evidence available to downstream reasoning even when its calling interface remains unchanged. We study a common contract for managing updates across resource types.

LLM-based systems combine reasoning, planning, and external tools (Yao et al., 2023; Wei et al., 2022; Wu et al., 2024). Communication standards address complementary concerns: MCP exposes tools, prompts, and contextual resources, while A2A supports agent discovery and task-oriented interaction (Model Context Protocol Contributors, 2025; A2A Protocol Contributors, 2026). We ask whether heterogeneous resource updates can share one lifecycle while each resource keeps its own execution contract. Our focus is a contract for proposing, evaluating, and retaining changes to heterogeneous agent-internal resources, complementing these communication mechanisms.

We propose **AUTOGENESIS PROTOCOL (AGP)**, with two complementary layers. The **Resource Substrate Protocol Layer (RSPL)** records resource identities, implementations, configurations, evolvability markers, and version histories. The **Self-Evolution Protocol Layer (SEPL)** composes proposal, evaluation, and commit operators over these records. An optimizer can thus propose changes through a shared lifecycle while resource-specific managers determine the permitted fields and invocation contracts. Together, the layers separate update proposals from the management of accepted resource versions and consumer bindings for subsequent task execution.

We instantiate AGP as **AUTOGENESIS SYSTEM (AGS)**, combining planner–specialist execution with reflection-driven resource updates. Our evaluation covers reasoning, tool use, and coding, complemented by repeated accuracy–cost comparisons and cases spanning all five resource types. These studies distinguish task-level refinement from protocol execution: benchmarks characterize system performance, while the cases illustrate concrete resource edits, acceptance decisions, and subsequent use by agents during task execution. We make three contributions:

[1]Resources instantiate the five RSPL types. Tools include local code-based tools, MCP tools, and skills. Outputs and solutions are task-local refinement targets, not persistent resource types.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

- A two-layer resource-update contract separating heterogeneous resource management from the strategy used to propose changes.
- An AGS reference instantiation and three cases illustrating resource selection, candidate evaluation, versioned commit, and subsequent use across all five supported resource types.
- An empirical study on reasoning, tool-use, and coding benchmarks, including repeated runs with output-token accounting.

2 Related Work

2.1 Agent Systems and Interoperability

ReAct interleaves reasoning and actions, while AutoGen coordinates customizable agents through conversations (Yao et al., 2023; Wu et al., 2024). MetaGPT encodes role-based operating procedures, and AgentVerse supports dynamically composed teams for collaborative problem solving (Hong et al., 2024; Chen et al., 2024). MCP standardizes contextual resources, prompts, tools, and client–server interaction, while A2A standardizes discovery, messages, tasks, and artifacts (Model Context Protocol Contributors, 2025; A2A Protocol Contributors, 2026). AGP complements these mechanisms by managing candidate changes, activation decisions, and retained versions of the underlying resources. For example, a tool exposed through MCP can retain its communication interface while AGP manages updates to its implementation. The communication contract specifies how the tool is invoked; the update contract specifies which implementation is available to subsequent consumers. Appendix E compares these objects and lifecycle boundaries.

2.2 Refinement and Resource Optimization

Self-Refine uses an LLM's feedback to revise its current response, whereas Reflexion stores verbal reflections to guide subsequent trials (Madaan et al., 2023; Shinn et al., 2023). TextGrad propagates textual feedback through computation graphs; MIPRO optimizes module instructions and demonstrations; GEPA uses trajectory reflection to guide prompt search (Yuksekgonul et al., 2025; Opsahl-Ong et al., 2024; Agrawal et al., 2026). AFlow and EvoAgentX extend optimization to workflow structure (Zhang et al., 2025b; Wang et al., 2025), while Alita constructs tools that can be reused in later execution (Qiu et al., 2025). These methods already retain feedback, candidates, or reusable artifacts. AGP specifies a shared contract for resource identity, candidate evaluation, and versioned commit across resource types. Appendix E.1 compares update targets, feedback, scope, and retained state. This comparison concerns mechanisms, not matched-budget performance.

3 Autogenesis Protocol

Figure 1 links task execution, candidate updates, and resource management. In the spreadsheet case, the Tool `xlsx_reader_tool` is registered at version `v1.0.0`: its implementation reads a workbook, and its invocation contract returns rows, path, and sheet. File-derived checks precede commit; the aggregation Agent then invokes the accepted Tool through its registered interface. Later Agent and Prompt changes follow the same lifecycle (Appendix A.1). RSPL records identities and versions; SEPL coordinates proposal, assessment, and commit across resource types.

3.1 Layer 1: Resource Substrate Protocol Layer

RSPL supports five resource types: instructions (Prompt), decision logic (Agent), callable capabilities (Tool), executable environment models (Environment), and storage and retrieval policies (Memory). Tools include local implementations, MCP-connected capabilities, and skills. SEPL acts on the exposed implementation or configuration fields through management interfaces; ordinary task-local state changes remain part of execution. This separation allows the same representation to support different update strategies without prescribing an exhaustive resource taxonomy.

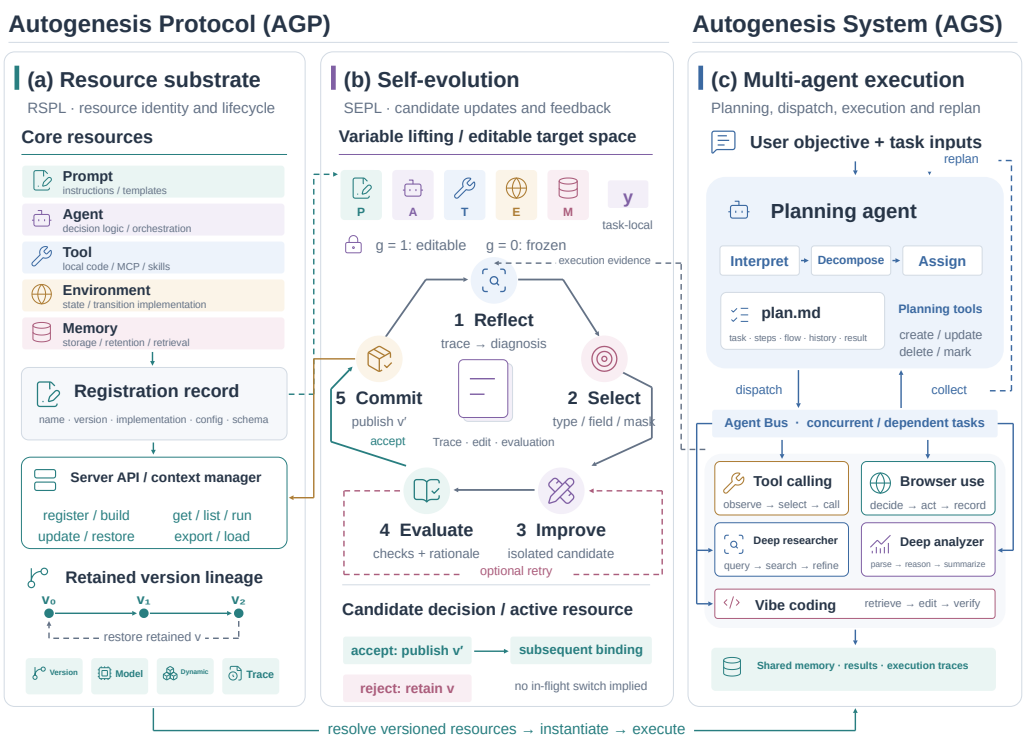


Figure 1: Autogenesis architecture. (a) RSPL manages five resource types and retained versions. (b) SEPL organizes candidate updates, evaluation, and commit; rejection differs from restoring a retained version. (c) AGS coordinates planning, specialists, and shared memory. Execution traces inform updates, and subsequent execution binds versioned resources. The version path is schematic.

Definition 3.1 (Resource Entity)**.** A resource entity and its type-level collection are defined as:

$$\begin{aligned} e_{\tau,i} &= (n_{\tau,i},\, d_{\tau,i},\, \phi_{\tau,i},\, g_{\tau,i},\, m_{\tau,i}), \\ \mathcal{E}_\tau &= \{\, e_{\tau,i} \mid i \in \mathcal{I}_\tau \,\}, \end{aligned} \tag{1}$$

where $\mathcal{T} = \{\textsc{Prompt}, \textsc{Agent}, \textsc{Tool}, \textsc{Env}, \textsc{Mem}\}$ indexes types and $i \in \mathcal{I}_\tau$ indexes registered resources of type τ. The name $n_{\tau,i}$ is unique within its type, giving the typed identity $(\tau, n_{\tau,i})$. Other fields are a description $d_{\tau,i}$, invocation mapping $\phi_{\tau,i} : \mathcal{X}_\tau \to \mathcal{Y}_\tau$, evolvability marker $g_{\tau,i} \in \{0, 1\}$, and metadata dictionary $m_{\tau,i}$. Invocation mappings include relevant runtime state.

Definition 3.2 (Resource Registration Record)**.** The record and its type-level collection are

$$\begin{aligned} c_{\tau,i} &= (e_{\tau,i},\, v_{\tau,i},\, \eta_{\tau,i},\, \theta_{\tau,i},\, \mathcal{F}_{\tau,i}), \\ \mathcal{C}_\tau &= \{\, c_{\tau,i} \mid i \in \mathcal{I}_\tau \,\}, \end{aligned} \tag{2}$$

where $e_{\tau,i}$ is the entity from Definition 3.1, $v_{\tau,i} \in \mathbb{V}$ identifies its version, $\eta_{\tau,i}$ specifies its implementation, and $\theta_{\tau,i}$ supplies instantiation parameters. The exported representations $\mathcal{F}_{\tau,i}$ describe how consumers interact with the resource, including function-calling schemas and text.

Definition 3.3 (Type-level resource interface)**.** For type τ, the registry $\mathcal{R}_\tau$ maps typed identities to active records in $\mathcal{C}_\tau$; retained versions are managed separately. The global registry $\mathcal{R} = \bigcup_\tau \mathcal{R}_\tau$ uses these type-qualified keys. RSPL exposes a type-level management interface, rather than a single resource instance, through the triple

$$r_\tau = (\mathcal{C}_\tau,\ \mathcal{M}_\tau,\ \mathcal{A}_\tau), \tag{3}$$

where $\mathcal{C}_\tau$ stores registration records, $\mathcal{M}_\tau$ manages registration, construction, execution, update, and restoration, and $\mathcal{A}_\tau$ exposes these operations to consumers. Exported contracts specify capabilities and arguments. Retained histories cover resource configurations and associated artifacts rather than arbitrary external state; Appendix G.2.2 details the APIs and their restoration boundaries.

3.1.1 INFRASTRUCTURE SERVICES

Four infrastructure services support resource management. The **model manager** routes requests across configured providers. The **version manager** tracks identifiers and lineage; resource managers retain configurations and artifacts used for restoration. The **dynamic manager** loads implementations and converts schemas, while the **trace manager** links execution observations to proposed changes and evaluation outcomes. Together, these services connect an update's motivating evidence to its recorded version and subsequent use, with activation and restoration boundaries specified below.

3.2 LAYER 2: SELF-EVOLUTION PROTOCOL LAYER (SEPL)

SEPL organizes updates as a closed loop over candidate states and evaluation messages. Here, self-evolution refers to feedback-driven edits to resources and task-local solutions rather than model-parameter training. In the reflection instantiation used by our experiments, execution evidence motivates an edit, evaluation assesses the candidate, and commit records an accepted version. The interface separates this proposal strategy from RSPL's management operations. Appendix G.3 provides the reflection algorithm and extension sketches.

3.2.1 EVOLVABLE VARIABLES

Variable lifting resolves a resource entity to editable fields of its registration record. The marker restricts targets; the manager specifies permitted fields, such as implementation η or parameters θ. Task-local outputs are also eligible for discrete, non-differentiable candidate edits.

Definition 3.4 (Evolvable Variable Set)**.** We write $\mathcal{V}_{\text{evo}} = (\bigcup_{\tau \in \mathcal{T}} \mathcal{E}_\tau) \cup \{y\}$, where y is a task-local output/solution, not a sixth persistent resource type. A marker $g_v \in \{0, 1\}$ selects $\Theta = \{v \in \mathcal{V}_{\text{evo}} \mid g_v = 1\}$. The state space $\mathfrak{V}$ includes configurations with new resource instances. Traces remain observational evidence and cannot be rewritten as candidate resource edits.

3.2.2 OPERATOR COMPOSITION

SEPL operators carry resource/output states alongside auxiliary messages containing candidates. The protocol requires proposal and assessment to leave active resource configurations and bindings unchanged; only accepted commits may publish updates. Composition requires compatible message types and lifecycle preconditions.

Definition 3.5 (SEPL Operator)**.** Let $\mathfrak{V}$ be the state space and $\mathcal{P}$ the message space carrying traces, hypotheses, edits, candidates, and assessments. A *SEPL operator* has the interface

$$f : \mathfrak{V} \times \mathcal{P}_{\text{in}} \ \rightarrow \ \mathfrak{V} \times \mathcal{P}_{\text{out}}, \tag{4}$$

where $\mathcal{P}_{\text{in}}, \mathcal{P}_{\text{out}} \subseteq \mathcal{P}$ specify message types. Applied to a state and message, f returns $(\mathcal{V}'_{\text{evo}}, p')$ with $\mathcal{V}'_{\text{evo}} \in \mathfrak{V}$ and $p' \in \mathcal{P}_{\text{out}}$. Compatible operators compose as $f_n \circ \cdots \circ f_1$. This is a dataflow interface; sampled proposals may be stochastic, while recovery requires consistent retained state.

3.2.3 EVOLUTIONARY LOOP

Initial execution under $\mathcal{V}_{\text{evo}}^{(0)} \in \mathfrak{V}$ produces the trace carried by $p^{(0)} \in \mathcal{P}$. At iteration t, compatible operators transform the current state and its associated message as follows:

$$\big(\mathcal{V}_{\text{evo}}^{(t+1)}, p^{(t+1)}\big) = (f_n \circ \cdots \circ f_1)\big(\mathcal{V}_{\text{evo}}^{(t)}, p^{(t)}\big), \tag{5}$$

where the reflection cycle comprises REFLECT, SELECT, IMPROVE, EVALUATE, and COMMIT, with task continuation only when required. Rejection must preserve the active resource binding while its assessment informs later proposals. Accepted solution edits retain task-local y without creating a persistent resource or triggering an unconditional new answer. Appendix G.3 specifies these rules.

Candidate, commit, and restoration boundaries. Commit records an accepted version for subsequent use; publication and consumer binding are distinct events. Restoration must realign retained configurations, version records, and consumer bindings. It assumes mediated mutations and a trusted registry/version store, covers only captured internal state, and cannot undo completed external actions. The cases in Appendix A exercise accepted updates, not recovery after faulty activation.

Feedback boundaries. For GPQA-Diamond/AIME, evaluation is a separate call to the same backbone, receiving the question and candidate reasoning/answer and returning a satisfaction flag and rationale. Reference answers are used only for final scoring. Executable tasks additionally provide execution feedback. Appendix B.1 lists the optimizer-visible fields; a positive self-assessment expresses the evaluator's judgment, not correctness against the withheld reference answer.

4 AUTOGENESIS SYSTEM

AGS instantiates AGP with a planner and five specialist roles. This organization builds on prior multi-agent and planner–specialist frameworks (Wu et al., 2024; Zhang et al., 2025c); an Agent Bus handles dispatch and result collection. The bus is an implementation choice, not a protocol requirement. Resource identities remain separate from instantiated implementations, allowing subsequent dispatches to resolve an accepted version. The resulting workflow proceeds as follows.

Task planning. The planner decomposes each incoming task without executing the subtasks itself. It maintains a structured `plan.md` containing the original task, assigned subtask steps, an execution flowchart, execution history, and a final result summary. Subtasks are assigned to deep researcher, browser-use, deep analyzer, tool-calling, or coding agents. The planner dispatches independent subtasks concurrently and respects dependencies between sequential steps, collecting specialist outputs through the bus before revising the plan and determining dependencies for the next round.

Execution and replanning. Each specialist retrieves the relevant prompt and tool resources from RSPL, executes its subtask, and writes results and traces to shared memory. After collecting the outputs, the planner updates `plan.md` and either concludes the task or initiates another round of decomposition. AGS also supports *agent-as-tool* composition: a specialist is exposed through an RSPL tool schema and invoked directly by another agent. This alternative uses the same resource interfaces without requiring bus-level orchestration, using the registered contracts for tool invocation.

Resource updates. Execution traces can trigger the reflection loop in Algorithm 1. The optimizer selects editable resources, constructs candidates, and evaluates them before commit. Subsequent execution can then use an accepted version through RSPL. The cases distinguish this post-commit execution from candidate replay; they also record when a frozen implementation requires creating a new resource rather than modifying the existing one. Publication does not automatically switch every in-flight consumer, so the binding scope remains part of the execution record.

The evaluated AGS update strategies use reflection. Appendix G.3 presents unevaluated textual-feedback and reward-informed sketches as partial mappings. Full integration requires candidate assessment and acceptance-gated publication. The reward-informed sketches use text similarity, not token likelihood ratios, and are not standard Reinforce++ or GRPO training.

5 EMPIRICAL STUDIES

We ask three questions: (i) How do different edit targets affect reasoning accuracy, and what output-token costs accompany the repeated evaluations (Table 1; Figure 3)? (ii) What task-level performance do the configured pipelines achieve in tool use and coding (Table 2, Figure 2, and Table 3)? (iii) How do different resource types follow the same update lifecycle (Appendix A)? The benchmarks characterize task outcomes, while the cases illustrate resource-management behavior.

Benchmarks. GPQA-Diamond has 198 graduate-level multiple-choice questions; AIME24 and AIME25 each have 30 integer-answer problems. Reasoning uses no external retrieval. GAIA (Mialon et al., 2024) tests planning and tool use across three difficulty levels; Humanity's Last Exam (HLE) (Phan et al., 2025) covers expert questions. Coding evaluation uses 100 selected LeetCode problems in Python3, C++, Java, Go, and Kotlin, reporting acceptance, execution statistics, and submission-relative metrics. Appendix F details the setup, metrics, and reproducibility constraints.

5.1 EXPERIMENTS ON SCIENTIFIC AND MATHEMATICAL BENCHMARKS

Experimental setup. We evaluate **AGS** on GPQA-Diamond, AIME24, and AIME25. *Prompt-Evo* edits prompts, *Solution-Evo* performs within-task solution refinement, and *PS-Joint-Evo* jointly edits both. The multi-model comparison permits up to three reflection rounds without external tools and

Table 1: Reasoning accuracy (%) with counts.

Agent	GPQA-Diamond	AIME24	AIME25
gpt-4o			
Vanilla	47.98 (95/198)	13.33 (4/30)	6.67 (2/30)
Prompt-Evo	53.54 (106/198)	13.33 (4/30)	13.33 (4/30)
Solution-Evo	53.54 (106/198)	16.67 (5/30)	13.33 (4/30)
PS-Joint-Evo	58.08 (115/198)	16.67 (5/30)	13.33 (4/30)
Gain (%)	**21.05↑**	**25.00↑**	**100.00↑**
gpt-4.1			
Vanilla	65.15 (129/198)	23.33 (7/30)	20.00 (6/30)
Prompt-Evo	68.69 (136/198)	33.33 (10/30)	23.33 (7/30)
Solution-Evo	68.69 (136/198)	36.67 (11/30)	30.00 (9/30)
PS-Joint-Evo	67.68 (134/198)	40.00 (12/30)	33.33 (10/30)
Gain (%)	**3.88↑**	**71.43↑**	**66.67↑**
grok-4.1-fast			
Vanilla	83.33 (165/198)	96.67 (29/30)	90.00 (27/30)
Prompt-Evo	83.84 (166/198)	96.67 (29/30)	93.33 (28/30)
Solution-Evo	87.88 (174/198)	96.67 (29/30)	90.00 (27/30)
PS-Joint-Evo	89.39 (177/198)	96.67 (29/30)	96.67 (29/30)
Gain (%)	**7.27↑**	0.00	**7.41↑**
claude-sonnet-4.5			
Vanilla	78.28 (155/198)	76.67 (23/30)	73.33 (22/30)
Prompt-Evo	79.80 (158/198)	86.67 (26/30)	90.00 (27/30)
Solution-Evo	80.30 (159/198)	80.00 (24/30)	90.00 (27/30)
PS-Joint-Evo	81.31 (161/198)	86.67 (26/30)	90.00 (27/30)
Gain (%)	**3.87↑**	**13.04↑**	**22.73↑**
gemini-3-flash-preview			
Vanilla	88.38 (175/198)	83.33 (25/30)	83.33 (25/30)
Prompt-Evo	88.89 (176/198)	93.33 (28/30)	86.67 (26/30)
Solution-Evo	87.88 (174/198)	93.33 (28/30)	90.00 (27/30)
PS-Joint-Evo	90.40 (179/198)	93.33 (28/30)	93.33 (28/30)
Gain (%)	**2.29↑**	**12.00↑**	**12.00↑**

Table 2: GAIA scores (%).

Agent	L1	L2	L3	Avg.
Validation				
HF ODR (Roucher et al., 2025)	67.92	53.49	34.62	55.15
o3-DR (OpenAI, 2025)	74.29	69.06	47.60	67.36
DeSearch (Desearch, 2025)	90.57	72.01	38.46	72.73
Co-Sight (Zhang et al., 2025a)	86.79	73.26	42.31	72.73
Manus (Shen & Yang, 2025)	86.50	70.10	57.69	73.90
AWorld (Yu et al., 2025)	88.68	77.91	53.85	77.58
Langfun (Peng, 2023)	88.68	80.23	57.69	79.39
Skywork (Zhang et al., 2025c)	92.45	83.72	57.69	82.42
agent-2030 (GAIA, 2025)	96.23	90.70	57.69	87.27
Alita† (Qiu et al., 2025)	88.68	89.53	76.92	87.27
Vanilla	92.45	88.37	88.46	89.70
Agent-Evo	**96.23**	**93.02**	**88.46**	**93.33**
Gain (%)	**4.09↑**	**5.26↑**	0.00	**4.05↑**
Test				
o4-mini-DR (OpenAI, 2025)	67.59	59.10	44.28	59.30
JoyAgent (Liu et al., 2025)	77.42	67.30	46.94	67.11
o3-DR (OpenAI, 2025)	79.42	68.97	47.48	68.70
Langfun (Peng, 2023)	84.95	73.58	48.98	73.09
Alita (Qiu et al., 2025)	92.47	71.70	55.10	75.42
DeSearch (Desearch, 2025)	91.40	75.47	61.22	78.07
h2o (H2O.ai, 2025)	89.25	79.87	61.22	79.73
Su-Zero-Ultra (GAIA, 2025)	93.55	77.36	65.31	80.40
AWorld (Yu et al., 2025)	95.70	81.13	57.14	81.73
HALO (Hou et al., 2025)	94.62	84.91	69.39	85.38
ToolOrchestra (Su et al., 2025)	95.70	82.39	87.76	87.38
openJiuwen (openJiuwen, 2026)	**98.92**	**88.68**	**87.76**	**91.69**
Vanilla	91.40	77.36	61.22	79.07
Agent-Evo	98.92	85.53	81.63	89.04
Gain (%)	**8.23↑**	**10.56↑**	**33.34↑**	**12.61↑**

Table 1: single complete evaluations; accuracy and Gain use integer counts before rounding. Repeated controls are separate (Appendix B). Table 2: external systems use their own backbones and budgets; †Alita validation uses pass@3. Gain compares the AGS variants only (Appendix B.2).

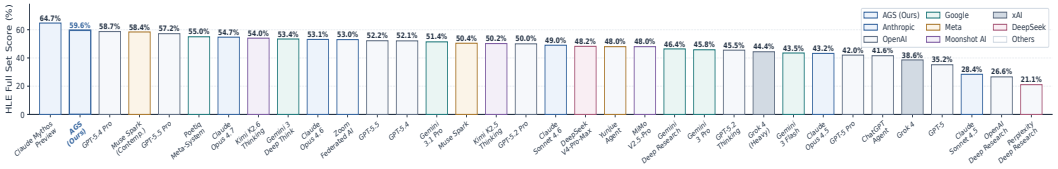


Figure 2: HLE full-set scores of agent systems (Zoom AI, 2026).

scores the final output after termination. Accuracy requires the correct GPQA-Diamond option or exact AIME integer. Table entries are single runs; repeated controls are analyzed separately. Reference answers are used only for scoring (Appendix B.1).

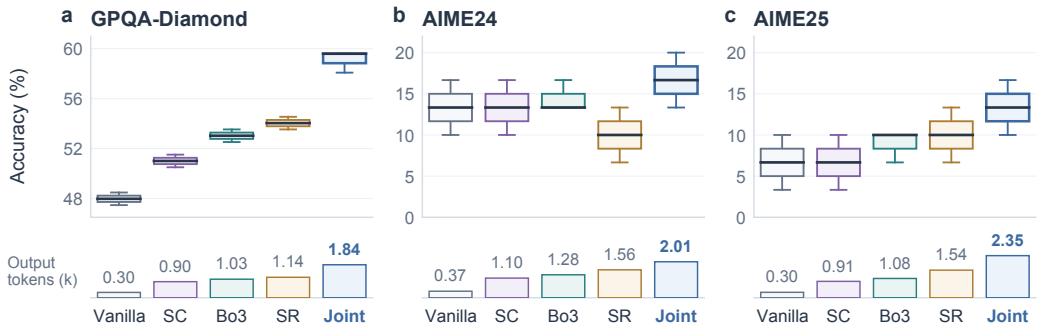


Figure 3: GPT-4o accuracy and cost (three runs). Boxes show the IQR and median; whiskers show the range. Bars show output tokens per problem (k). Appendix B gives exact results and settings.

Results. Joint evolution improves on Vanilla in most reported configurations, although the relative benefit of editing both targets depends on the backbone and task. On GPT-4.1 GPQA-Diamond, joint evolution scores 67.68 compared with 68.69 for each single-target variant; on Grok-4.1-fast AIME24, all evolution variants score 96.67. Net GPQA-Diamond gains over Vanilla span 4–20 correct answers across backbones. On AIME, each additional correct answer changes accuracy by 3.33 percentage points, making counts essential to interpreting relative gains. Joint editing is therefore not uniformly preferable, and its benefit should be weighed against inference cost. We supplement these single-run comparisons with repeated GPT-4o evaluations that report per-run counts, accuracy variation, and output-token costs under controlled primary generation opportunities.

Repeated, opportunity-controlled comparison. We additionally compare GPT-4o under at most three primary generation/refinement opportunities. Vanilla uses one unrefined answer; Self-Consistency (SC) samples three answers and votes; Best-of-3 (Bo3) uses a model judge; Simple Reflection (SR) produces an initial solution and two refinements without persistent resource updates; PS-Joint-Evo (Joint) edits the prompt and solution. All use the same reported route and decoding configuration, with a 32,768 output-token ceiling per call. This ceiling is not an equal realized-compute budget. Auxiliary reflection, editing, and evaluation calls must be distinguished from primary opportunities when interpreting the measured accuracy–cost trade-off.

Across three evaluations, joint editing reaches 59.09% mean accuracy on GPQA-Diamond versus 54.04% for Simple Reflection (351/594 versus 321/594 correct responses). On AIME24, it yields 15/90 versus 13/90 for Best-of-3; on AIME25, 12/90 versus 9/90 for Simple Reflection. Gains are 5.05, 2.22, and 3.33 percentage points, with approximately 61%, 57%, and 52% more output tokens than these respective controls. Denominators count repeated responses to the same 198 or 30 questions, not additional independent problems. These results characterize a higher-accuracy operating point at increased output-token cost, not an equal-compute advantage.

5.2 Experiments on General Agent Benchmarks

Experimental setup. AGS uses a planning agent with step bound $m = 30$ and five specialist roles, each with $m = 20$ and `gemini-3.1-pro-preview`. These are per-agent limits rather than a cap on total system calls. The *Agent-Evo* variant permits edits to prompts, tools, and code, whereas *Vanilla* retains the initial resources. We report GAIA task-completion accuracy by difficulty and split; HLE uses `o3-mini` for final judging. Sources are cited in Table 2 (see also Appendix B.2). The comparison evaluates the complete configured system, including the additional work required to propose and assess edits and execute the updated resources within the stated per-agent step limits.

Results. Agent-Evo increases reported GAIA validation accuracy from 89.70 to 93.33 and test accuracy from 79.07 to 89.04 within the AGS configuration, and AGS scores 59.6 on HLE. Level 3 shows the largest test increase, from 61.22 to 81.63, but no validation increase, indicating that improvements differ across splits and difficulty levels. The within-AGS comparisons share the reported backbone and planner–specialist organization. External leaderboard entries contextualize these scores, but differences in backbones, tools, and inference budgets prevent a controlled comparison.

Table 3: Coding results on 100 LeetCode problems. Gain compares Solution-Evo with the preceding Gemini baseline (↑: favorable; ↓: unfavorable). Appendix F defines metrics and aggregation.

Model	Capability metrics								Execution metrics			Relative metrics	
	PR	TLE	MLE	CE	RE	WA	TO	RpE	AR (ms)	AM (MB)	APC	ARB (%)	AMB (%)
Python3													
deepseek-v3.2	34	1	1	0	23	8	1	36	1806.79	55.91	640.38	63.04	25.81
grok-4.1-fast	73	9	0	0	3	13	0	3	1860.90	56.02	741.60	49.92	30.15
claude-sonnet-4.5	42	37	2	0	0	10	8	1	880.98	45.16	702.64	61.06	22.12
claude-opus-4.5	82	9	0	0	0	5	3	1	1559.87	70.77	749.45	64.77	32.70
gemini-3-flash-preview	79	4	0	0	2	14	1	0	1376.19	56.59	750.89	73.28	36.62
+ Solution-Evo	87	3	0	0	1	9	0	0	1269.39	59.08	750.98	70.29	42.15
Gain (%)	**10.1↑**	**25.0↑**	0	0	**50↑**	**35.7↑**	**100↑**	0	**7.8↑**	4.4↓	0.0	4.1↓	15.1↑
C++													
deepseek-v3.2	11	1	0	30	0	6	4	43	158.73	163.59	605.82	73.11	74.05
grok-4.1-fast	79	9	0	1	0	5	2	4	428.32	223.68	748.61	58.57	46.67
claude-sonnet-4.5	41	42	2	0	1	9	2	3	379.68	179.86	710.59	56.17	50.84
claude-opus-4.5	85	6	0	0	0	6	1	2	382.45	184.22	758.21	64.06	55.58
gemini-3-flash-preview	84	2	0	2	1	10	0	1	266.04	168.93	743.31	68.02	59.24
+ Solution-Evo	99	0	0	0	0	1	0	0	142.60	148.43	749.86	88.99	73.14
Gain (%)	**17.9↑**	**100↑**	0	**100↑**	**100↑**	**90↑**	0	**100↑**	**46.4↑**	**12.1↑**	**0.9↑**	**30.8↑**	**23.5↑**
Java													
deepseek-v3.2	11	0	0	47	1	1	5	35	72.91	143.63	481.45	57.37	32.71
grok-4.1-fast	73	5	0	5	0	12	1	4	227.45	136.80	746.23	52.98	41.97
claude-sonnet-4.5	41	40	1	1	1	15	0	1	161.49	130.54	679.41	58.04	46.22
claude-opus-4.5	87	4	1	0	0	6	1	1	188.63	134.27	748.63	59.54	55.61
gemini-3-flash-preview	84	0	0	2	2	9	1	2	125.04	126.09	752.86	71.03	59.18
+ Solution-Evo	98	1	0	0	0	1	0	0	96.30	120.00	751.09	88.33	72.38
Gain (%)	**16.7↑**	—	0	**100↑**	**100↑**	**88.9↑**	**100↑**	**100↑**	**23.0↑**	**4.8↑**	0.2↓	**24.4↑**	**22.3↑**
Go													
deepseek-v3.2	7	0	0	39	0	0	1	53	112.71	12.59	709.57	62.73	54.36
grok-4.1-fast	69	3	0	16	0	4	3	5	194.90	23.26	755.43	66.83	62.44
claude-sonnet-4.5	44	41	0	0	0	13	0	2	222.64	19.71	712.55	57.09	53.32
claude-opus-4.5	84	5	0	0	0	9	0	2	162.50	19.95	744.45	72.91	63.00
gemini-3-flash-preview	82	1	0	9	0	7	0	1	139.22	22.01	739.46	76.22	63.48
+ Solution-Evo	95	0	0	0	0	5	0	0	111.64	18.35	754.17	81.52	67.94
Gain (%)	**15.9↑**	**100↑**	0	**100↑**	0	**28.6↑**	0	**100↑**	**19.8↑**	**16.6↑**	**2.0↑**	**7.0↑**	**7.0↑**
Kotlin													
deepseek-v3.2	7	2	0	0	0	0	1	48	59.29	62.27	793.57	58.33	74.56
grok-4.1-fast	62	2	0	22	0	8	2	4	307.45	75.45	759.55	78.12	72.83
claude-sonnet-4.5	42	36	1	8	1	10	1	1	192.62	78.49	757.64	81.59	77.79
claude-opus-4.5	83	4	0	5	0	5	0	3	210.47	76.60	750.98	83.18	76.53
gemini-3-flash-preview	75	2	0	8	1	10	2	2	171.99	72.80	760.43	83.49	79.07
+ Solution-Evo	95	1	0	0	0	4	0	0	122.83	77.88	749.38	83.58	67.21
Gain (%)	**26.7↑**	**50↑**	0	**100↑**	**100↑**	**60↑**	**100↑**	**100↑**	**28.6↑**	7.0↓	1.5↓	**0.1↑**	15.0↓

5.3 Experiments on Self-Evolving Code Agent Benchmark

Experimental setup. We compare Vanilla and Solution-Evo using `gemini-3-flash-preview` on 100 recently released LeetCode problems across five languages. Solution-Evo permits an initial submission and up to two revisions using online-judge feedback. Recent release dates do not establish freedom from training contamination, and other backbones are contextual rather than matched-budget controls. Appendices F and D define the correctness, runtime, memory, and submission-percentile metrics and the associated platform data-use and release policies.

Results. Solution-Evo increases solved counts from 79 to 87 in Python3, 84 to 99 in C++, 84 to 98 in Java, 82 to 95 in Go, and 75 to 95 in Kotlin. These gains characterize execution-guided revision within a task. Mean runtime decreases over each method's accepted solutions, whereas memory and submission-relative percentiles show mixed changes. For example, Kotlin's higher solved count accompanies lower mean runtime but higher mean memory, illustrating why acceptance and execution metrics should be reported separately. Different accepted-problem sets prevent a per-problem efficiency conclusion: this requires paired measurements on commonly solved problems and accounting for extra submissions and reflection calls.

Error profile. Solution-Evo has no remaining compilation errors in any language. Wrong answers remain the largest failure category in Python3, Go, and Kotlin, so successful compilation alone does not ensure correct solutions. These aggregate categories do not trace individual repairs. Figure 4 indexes evaluated tasks, not optimization rounds; its curves aggregate outcomes as the evaluated task set grows, rather than tracking successive revisions of a single program.

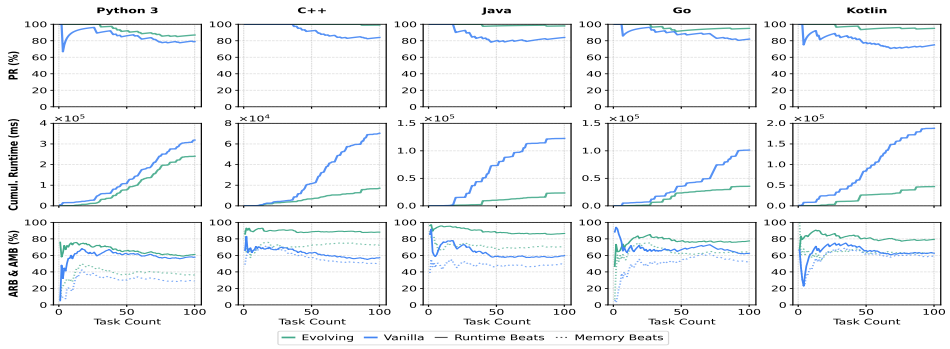


Figure 4: Coding metrics over task count for Vanilla and Solution-Evo.

5.4 Resource-Update Cases

The cases in Appendix A connect resource interfaces to execution decisions. Spreadsheet processing requires a new Tool for reading, a new Agent for auditable aggregation because the built-in implementation is frozen, and a Prompt update separating verification from the answer. A map dead end instead motivates changing the executable Environment. Resource selection thus addresses the source of failure while keeping task inputs and scoring fixed.

The Memory case changes retention and retrieval after compaction. On the same 45-record workload, returned constraints increase from zero to eight, with 25 ordinary recent records preserved. Because the updated policy also adds storage, this is a retrieval result rather than a task-accuracy gain. Across the three cases, different failure signals lead to resource-specific edits under the same sequence of candidate checks, commit, and subsequent invocation. The edited object determines the check: spreadsheet processing checks returned data, the Environment case checks transition behavior, and the Memory case checks retrieved records. Each check concerns the edited resource's behavior; an accepted version becomes available for subsequent task execution through its registered interface.

6 Limitations and Impact Statement

Empirical scope. The quantitative evaluations assess configured AGS pipelines, including the work required to propose and evaluate edits. GPT-4o controls primary generation opportunities rather than total computation; each AIME set has only 30 questions. The cases illustrate selected accepted updates, not population-level gains. We do not isolate resource management from the update strategy or establish equal-cost superiority, cross-task transfer, or recovery after faulty activation. Conclusions concern the interface and its evaluated uses, not general deployment reliability.

Safety and data governance. Version histories provide inspection records and restoration targets for retained internal state; they do not guarantee behavioral safety or monotonic improvement. Self-evaluation may accept incorrect changes, and external actions may be irreversible. Executable edits require deployment-specific isolation and permissions; traces and memory require access, redaction, and retention controls, separate from the protocol interface. The public release will exclude LeetCode-derived content, task-level traces, credentials, and platform-specific collection/submission tools. Platform use remains subject to the applicable terms; Appendix D specifies these release exclusions and the resulting limits on reproducing evaluations on the original platform tasks.

7 Conclusion

We introduced AGP, a two-layer protocol separating heterogeneous resource management from update strategies. AGS instantiates it through planner–specialist execution; three cases illustrate evaluation, versioned commit, and subsequent use across five resource types. The experiments show task-level gains at additional computational cost, and the cases expose concrete resource changes and activation decisions. AGP thus provides a common lifecycle for heterogeneous resource updates while keeping proposal strategies and resource-specific execution contracts separate.

REFERENCES

A2A Protocol Contributors. Agent2Agent Protocol Specification. https://a2a-protocol.org/latest/specification/, 2026. Accessed 2026-09-22; specification snapshot used for scope comparison.

Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. GEPA: Reflective Prompt Evolution Can Outperform Reinforcement Learning. In *International Conference on Learning Representations*, 2026. URL https://arxiv.org/abs/2507.19457v2.

Anthropic. Equipping Agents for the Real World with Agent Skills. https://www.anthropic.com/engineering/equipping-agents-for-the-real-world-with-agent-skills, October 2025. Published 2025-10-16; accessed 2026-09-23.

Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chi-Min Chan, Heyang Yu, Yaxi Lu, Yi-Hsin Hung, Chen Qian, et al. AgentVerse: Facilitating Multi-Agent Collaboration and Exploring Emergent Behaviors. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/578e65cdee35d00c708d4c64bce32971-Abstract-Conference.html.

Jasper Dekoninck, Nikola Jovanović, Tim Gehrunger, Kári Rögnvaldsson, Ivo Petrov, Chenhao Sun, and Martin Vechev. Beyond Benchmarks: MathArena as an Evaluation Platform for Mathematics with LLMs. *arXiv preprint arXiv:2605.00674*, 2026. URL https://arxiv.org/abs/2605.00674. Official citation supplied by the AIME 2025 dataset card: https://huggingface.co/datasets/MathArena/aime_2025. Accessed 2026-09-23.

Desearch. Desearch API in Python. https://pypi.org/project/desearch-py/1.0.0/, 2025. Version 1.0.0, released 2025-04-23. Accessed 2026-09-23.

GAIA. GAIA Leaderboard Public Results. https://huggingface.co/datasets/gaia-benchmark/results_public/tree/c4c701655c4f44c33074ca537bda54e713a26836, 2025. Entries agent-2030-v2.0 (validation, 2025-05-27) and Su Zero Ultra (test, 2025-06-26). Accessed 2026-09-26.

GAIA Benchmark Contributors. GAIA Leaderboard Evaluation Implementation. https://huggingface.co/spaces/gaia-benchmark/leaderboard/blob/010bcb1d14ff54230bd1c23e2e30aae2055b55b5/app.py, 2026. Version inspected 2026-09-22; official split and aggregation definitions, not an AGS run manifest.

H2O.ai. Enterprise h2oGPTe v1.6.0 Release Notes. https://docs.h2o.ai/enterprise-h2ogpte/v1.6.28/release-notes/v1.6.0, 2025. Released 2025-01-31. Accessed 2026-09-23.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, et al. MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework. In *International Conference on Learning Representations*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/6507b115562bb0a305f1958ccc87355a-Abstract-Conference.html.

Zhipeng Hou, Junyi Tang, and Yipeng Wang. HALO: Hierarchical Autonomous Logic-Oriented Orchestration for Multi-Agent LLM Systems. *arXiv preprint arXiv:2505.13516*, 2025. URL https://arxiv.org/abs/2505.13516.

Jian Hu. REINFORCE++: A Simple and Efficient Approach for Aligning Large Language Models. *arXiv preprint arXiv:2501.03262v1*, 2025. URL https://arxiv.org/abs/2501.03262v1.

Hugging Face H4. AIME 2024 dataset. https://huggingface.co/datasets/HuggingFaceH4/aime_2024, 2025. Dataset released 2025-01-23. Accessed 2026-09-23.

LeetCode. Terms of Service. https://leetcode.com/terms/, 2024. Page marked last modified 04/05/2024; accessed 2026-09-22.

Jiarun Liu, Shiyue Xu, Shangkun Liu, Yang Li, Wen Liu, Min Liu, Xiaoqing Zhou, Hanmin Wang, Shilin Jia, Zhen Wang, et al. JoyAgent-JDGenie: Technical Report on the GAIA. *arXiv preprint arXiv:2510.00510*, 2025. URL https://arxiv.org/abs/2510.00510v1.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-Refine: Iterative Refinement with Self-Feedback. *Advances in neural information processing systems*, 36:46534–46594, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/91edff07232fb1b55a505a9e9f6c0ff3-Abstract-Conference.html.

Grégoire Mialon, Clémentine Fourrier, Thomas Wolf, Yann LeCun, and Thomas Scialom. GAIA: A Benchmark for General AI Assistants. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/25ae35b5b1738d80f1f03a8713e405ec-Abstract-Conference.html.

Model Context Protocol Contributors. Model Context Protocol Specification, 2025-11-25. https://modelcontextprotocol.io/specification/2025-11-25, 2025. Accessed 2026-09-22.

OpenAI. Introducing Deep Research. https://openai.com/index/introducing-deep-research/, February 2025.

openJiuwen. openJiuwen-deepagent: GAIA Benchmark Submission. https://gaia-benchmark-leaderboard.hf.space/, 2026. Official leaderboard entry dated 2026-02-09, submitted by openJiuwen with GPT5 and Gemini-3-Pro. Accessed 2026-09-23.

Krista Opsahl-Ong, Michael J Ryan, Josh Purtell, David Broman, Christopher Potts, Matei Zaharia, and Omar Khattab. Optimizing Instructions and Demonstrations for Multi-Stage Language Model Programs. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 9340–9366, 2024. doi: 10.18653/v1/2024.emnlp-main.525. URL https://aclanthology.org/2024.emnlp-main.525/.

Daiyi Peng. Langfun. https://github.com/google/langfun, 2023. Version 0.0.1, released 2023-09-14. Accessed 2026-09-23.

Long Phan, Alice Gatti, Ziwen Han, Nathaniel Li, Josephina Hu, Hugh Zhang, Chen Bo Calvin Zhang, Mohamed Shaaban, John Ling, Sean Shi, et al. Humanity's last exam. *arXiv preprint arXiv:2501.14249*, 2025. URL https://arxiv.org/abs/2501.14249.

Jiahao Qiu, Xuan Qi, Tongcheng Zhang, Xinzhe Juan, Jiacheng Guo, Yifu Lu, Yimin Wang, Zixin Yao, Qihan Ren, Xun Jiang, et al. Alita: Generalist agent enabling scalable agentic reasoning with minimal predefinition and maximal self-evolution. *arXiv preprint arXiv:2505.20286*, 2025. URL https://arxiv.org/abs/2505.20286v1.

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R. Bowman. GPQA: A Graduate-Level Google-Proof Q&A Benchmark. In *First Conference on Language Modeling*, 2024. URL https://arxiv.org/abs/2311.12022.

Aymeric Roucher, Albert Villanova del Moral, Merve Noyan, Thomas Wolf, and Clémentine Fourrier. Open-source DeepResearch - Freeing Our Search Agents. https://huggingface.co/blog/open-deep-research, 2025. Published 2025-02-04.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models. *arXiv preprint arXiv:2402.03300*, 2024. URL https://arxiv.org/abs/2402.03300.

Minjie Shen and Qikai Yang. From Mind to Machine: The Rise of Manus AI as a Fully Autonomous Digital Agent, 2025. URL https://arxiv.org/abs/2505.02024v1.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/1b44b878bb782e6954cd888628510e90-Abstract-Conference.html.

Hongjin Su, Shizhe Diao, Ximing Lu, Mingjie Liu, Jiacheng Xu, Xin Dong, Yonggan Fu, Peter Belcak, Hanrong Ye, Hongxu Yin, et al. ToolOrchestra: Elevating Intelligence via Efficient Model and Tool Orchestration. *arXiv preprint arXiv:2511.21689*, 2025. URL https://arxiv.org/abs/2511.21689.

Yingxu Wang, Siwei Liu, Jinyuan Fang, and Zaiqiao Meng. EvoAgentX: An Automated Framework for Evolving Agentic Workflows. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 643–655, 2025. doi: 10.18653/v1/2025.emnlp-demos.47. URL https://aclanthology.org/2025.emnlp-demos.47/.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022. URL https://proceedings.neurips.cc/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, et al. AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversations. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/attachment?id=BAakY1hNKS&name=pdf.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. ReAct: Synergizing Reasoning and Acting in Language Models. In *International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

Chengyue Yu, Siyuan Lu, Chenyi Zhuang, Dong Wang, Qintong Wu, Zongyue Li, Runsheng Gan, Chunfeng Wang, Siqi Hou, Gaochi Huang, et al. AWorld: Orchestrating the Training Recipe for Agentic AI. *arXiv preprint arXiv:2508.20404*, 2025. URL https://arxiv.org/abs/2508.20404.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Pan Lu, Zhi Huang, Carlos Guestrin, and James Zou. Optimizing generative AI by backpropagating language model feedback. *Nature*, 639(8055):609–616, 2025. doi: 10.1038/s41586-025-08661-4. URL https://www.nature.com/articles/s41586-025-08661-4.

Hongwei Zhang, Ji Lu, Shiqing Jiang, Chenxiang Zhu, Li Xie, Chen Zhong, Haoran Chen, Yurui Zhu, Yongsheng Du, Yanqin Gao, Lingjun Huang, Baoli Wang, Fang Tan, and Peng Zou. Co-Sight: Enhancing LLM-Based Agents via Conflict-Aware Meta-Verification and Trustworthy Reasoning with Structured Facts. *arXiv preprint arXiv:2510.21557*, 2025a. URL https://arxiv.org/abs/2510.21557.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. AFlow: Automating Agentic Workflow Generation. In *International Conference on Learning Representations*, 2025b. URL https://arxiv.org/abs/2410.10762v4.

Wentao Zhang, Ce Cui, Yilei Zhao, Yang Liu, and Bo An. AgentOrchestra: A Hierarchical Multi-Agent Framework for General-Purpose Task Solving. *arXiv preprint arXiv:2506.12508v1*, 2025c. URL https://arxiv.org/abs/2506.12508v1.

Zoom AI. HLE Leaderboard for Agents with Tools. https://huggingface.co/spaces/zoom-ai/hle-leaderboard/tree/0cbe9ae3448216898de9c0016ea06b2f257e4fb3, 2026. Leaderboard revision dated 2026-07-23. Accessed 2026-09-23.

A Qualitative Resource-Evolution Cases

These selected cases use `google/gemini-3.1-pro-preview` and cover all five resource types. Figures paraphrase task requests and condense execution traces; the Memory policy is shown as a simplified sketch. All shown updates are accepted, so the cases do not exercise version rollback.

A.1 Tool, Agent, and Prompt: auditable spreadsheet aggregation

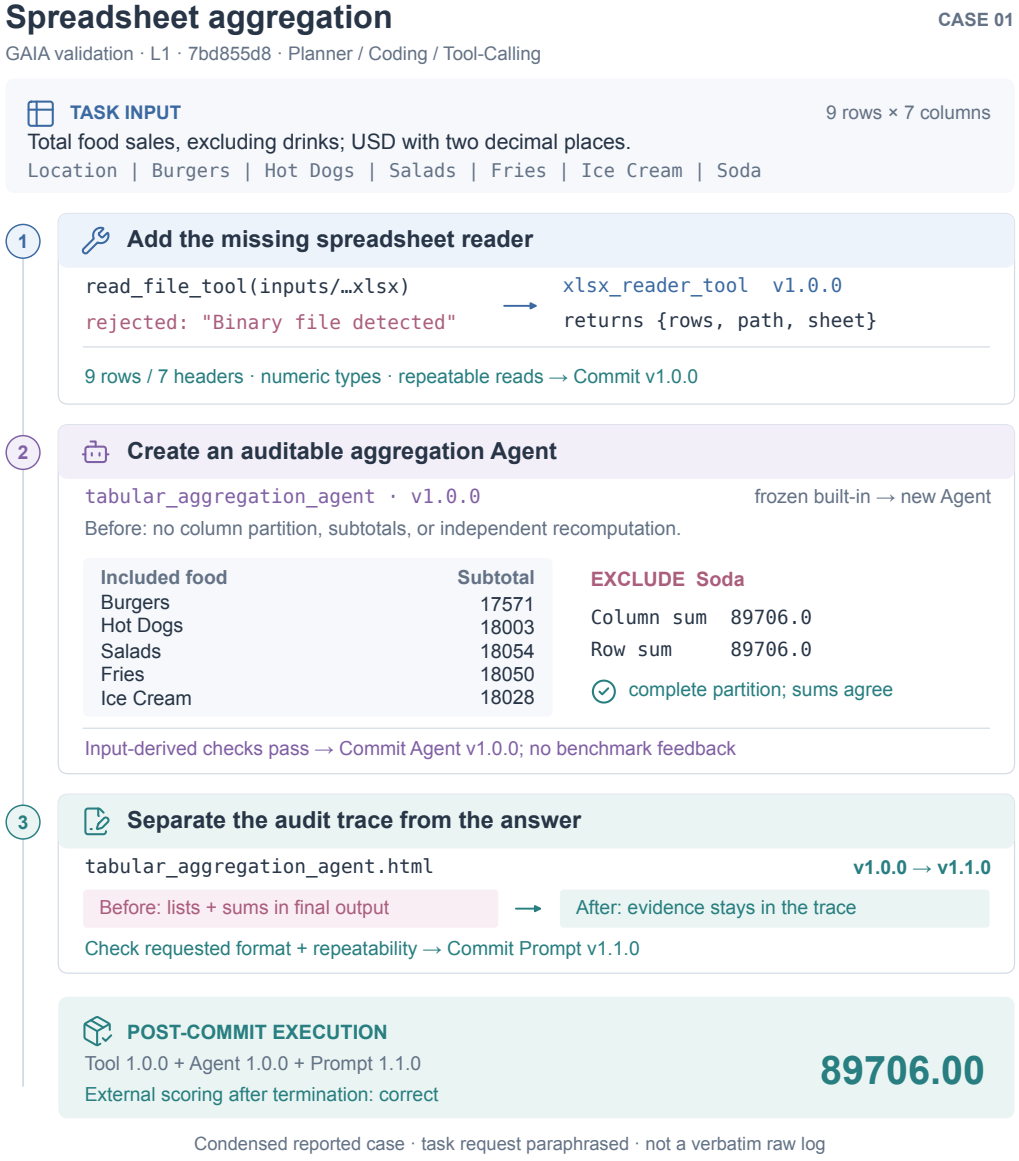


Figure 5: Tool, Agent, and Prompt evolution on a GAIA spreadsheet task. Input-derived checks precede each commit; the final benchmark comparison is external to the update loop.

Interpretation. The frozen built-in Agent is not edited: a new Agent supplies the missing aggregation contract. The later Prompt update is task-independent and contains neither an expected value nor specific columns; formatting follows the user's request.

A.2 Environment: correcting an executable transition model

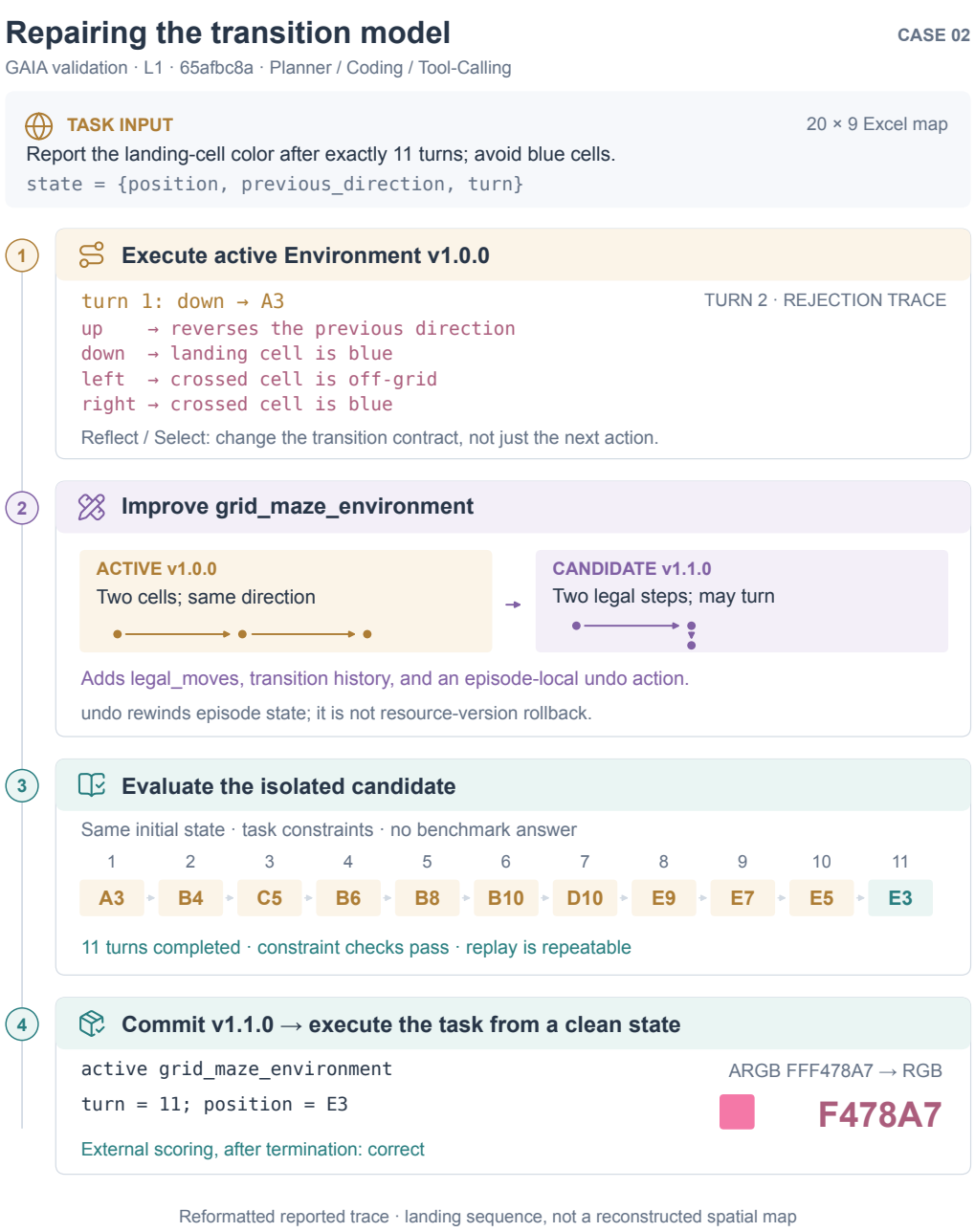


Figure 6: Environment evolution on a GAIA map task. The candidate changes executable transition semantics; task rules and map remain fixed inputs. The timeline lists landing cells from the trace, not a reconstructed spatial map.

Interpretation. The planner first registers the generated Environment, whose callable interface passes initial checks but whose transition semantics fail during execution. Candidate evaluation uses the task specification, not an expected terminal cell or benchmark answer. Episode-local `undo` is distinct from resource-version restoration.

A.3 MEMORY: CONSTRAINT RETENTION THROUGH THE RUNTIME INTERFACE



Figure 7: Memory-policy replay on a GAIA-derived workload. The log-style panels summarize retrieved records before and after the update; the policy sketch is simplified from the implementation.

Interpretation. The case measures what the active Memory resource returns to the prompt, not final task accuracy. The updated policy adds a persistent collection, so part of the gain may come from the extra storage.

B Repeated Evaluations and Feedback Boundaries

Configuration and interpretation. For the repeated evaluations, GPT-4o is accessed through OpenRouter: `openrouter/gpt-4o` resolves to `openai/gpt-4o`. Decoding uses temperature 0.7 and frequency penalty 0.3, with no specified top-p or seed. The reported ceiling is 32,768 output tokens per call. No browser, search, or retrieval tools are exposed. Three complete evaluations use the same benchmark snapshot, source prompts, parser, and method-specific evaluator. Repetition captures pipeline sampling and judge variation, not three disjoint datasets.

Table 4: Correct/total in each complete GPT-4o evaluation. Each run evaluates 198 GPQA-Diamond questions or 30 AIME questions. SD is the sample standard deviation of three run accuracies, not a confidence interval; each run repeats the same questions.

Dataset	Method	Run 1	Run 2	Run 3	Mean $\pm$ SD (%)
GPQA-Diamond ($n = 198$)	Vanilla	95/198	94/198	96/198	47.98 ± 0.51
	Self-Consistency	101/198	100/198	102/198	51.01 ± 0.51
	Best-of-3	104/198	105/198	106/198	53.03 ± 0.51
	Simple Reflection	107/198	106/198	108/198	54.04 ± 0.51
	PS-Joint-Evo	115/198	118/198	118/198	59.09 ± 0.87
AIME24 ($n = 30$)	Vanilla	4/30	3/30	5/30	13.33 ± 3.33
	Self-Consistency	4/30	3/30	5/30	13.33 ± 3.33
	Best-of-3	4/30	4/30	5/30	14.44 ± 1.92
	Simple Reflection	3/30	2/30	4/30	10.00 ± 3.33
	PS-Joint-Evo	5/30	4/30	6/30	16.67 ± 3.33
AIME25 ($n = 30$)	Vanilla	2/30	1/30	3/30	6.67 ± 3.33
	Self-Consistency	2/30	1/30	3/30	6.67 ± 3.33
	Best-of-3	2/30	3/30	3/30	8.89 ± 1.92
	Simple Reflection	3/30	2/30	4/30	10.00 ± 3.33
	PS-Joint-Evo	4/30	3/30	5/30	13.33 ± 3.33

Table 5: Three complete GPT-4o evaluations. Each cell reports mean accuracy $\pm$ sample SD (%), pooled correct/total responses, and approximate output tokens per problem. Pooled denominators are $3 \times 198 = 594$ for GPQA-Diamond and $3 \times 30 = 90$ for each AIME set, not distinct-question counts. Methods match primary opportunities, not total tokens or calls; Best-of-3 costs include judge output.

Method	GPQA-Diamond	AIME24	AIME25
Vanilla	47.98 ± 0.51 285/594 299 tokens	13.33 ± 3.33 12/90 365 tokens	6.67 ± 3.33 6/90 304 tokens
Self-Consistency	51.01 ± 0.51 303/594 896 tokens	13.33 ± 3.33 12/90 1,095 tokens	6.67 ± 3.33 6/90 913 tokens
Best-of-3	53.03 ± 0.51 315/594 1,033 tokens	14.44 ± 1.92 13/90 1,279 tokens	8.89 ± 1.92 8/90 1,075 tokens
Simple Reflection	54.04 ± 0.51 321/594 1,141 tokens	10.00 ± 3.33 9/90 1,556 tokens	10.00 ± 3.33 9/90 1,538 tokens
PS-Joint-Evo	$\mathbf{59.09 \pm 0.87}$ 351/594 1,838 tokens	$\mathbf{16.67 \pm 3.33}$ 15/90 2,005 tokens	$\mathbf{13.33 \pm 3.33}$ 12/90 2,345 tokens

Boxplot conventions. Quartiles use linear interpolation across the three run accuracies; whiskers span the observed minimum and maximum. Coincident quartiles or medians reflect repeated outcomes, not missing observations. The AIME panels share a vertical scale, while GPQA-Diamond uses its own scale. Lower bars show output-token costs on a shared zero-based scale and align with the boxes

by method. Individual points are omitted for clarity; Table 4 preserves all 45 run counts. With only three runs per method, these summaries do not characterize population distributions.

Question counts and repeated responses. The GPQA-Diamond specification gives 198 questions (Rein et al., 2024); public AIME2024 and AIME2025 releases each contain 30 questions (Hugging Face H4, 2025; Dekoninck et al., 2026). These sizes agree with the denominators in the reported repeated evaluations. Table 4 reports each run as k/n, while Table 5 sums correct and total responses across three runs. Thus, 594 GPQA-Diamond responses correspond to three evaluations of 198 questions, and 90 AIME responses to three evaluations of 30 questions. These totals do not imply additional unique questions or independent observations across runs.

B.1 Optimizer-visible information

Table 6: Feedback policy for GPQA-Diamond/AIME reflection.

Stage	Available information	Output / restriction
Reflect / Select / Improve	Question, system instruction, current editable prompt/solution, self-generated reasoning and prior label-free reflection	Proposed change; no reference answer, correctness label, or benchmark score
Evaluate	Question and candidate reasoning/answer; a separate call to the same backbone	`is_satisfied` (Boolean), `reasoning` (text); consistency, completeness, and format checks, not exact-match correctness
Commit / stop	Candidate identity, edited target, and evaluation result	Record an accepted resource version or retain an accepted task-local solution; stop according to the task configuration
Final scoring	Final answer and held-out benchmark answer	Accuracy computed after termination; not returned to the optimizer or stored in an evolving resource

The three-primary-opportunity control means one initial answer and at most two primary revisions for the sequential methods. It is distinct from the generic algorithm's update-iteration budget and from the total number of model calls. A negative self-evaluation can inform the next revision, but neither assessment exposes the reference answer.

Prompt-Evo enables prompt edits while retaining the other resource implementations. Solution-Evo enables edits to the current answer or code without creating a persistent solution resource. PS-Joint-Evo enables both targets within the same task. Here, "joint" describes the editable targets, not a separate model-training procedure or evidence of cross-task transfer. Self-evaluation selects or terminates candidates; final benchmark scoring is a separate operation after termination, with no label feedback to the optimizer.

System-level cost. AGS uses approximately 49 model-inference steps and 23 minutes per GAIA task, 67 steps and 11 minutes per HLE task (including judging), and 12 steps and 4 minutes per coding task. Total API expenditures are approximately USD 2,000, 6,000, and 200, respectively; the GAIA/HLE experiments ran in April 2026 and the coding experiments in February 2026. These are aggregate figures, not matched-cost comparisons between variants.

Table generation. Figure 3 and Tables 4–5 are generated from one machine-readable summary of the GPT-4o counts and output-token costs; means and sample SDs are computed from integer counts before rounding.

Table 1 reports one complete evaluation per model and method as accuracy (correct/total), with $n = 198$ for GPQA-Diamond and $n = 30$ for AIME. Accuracy is $100k/n$ and relative Gain is $100(k_{\text{joint}} - k_{\text{vanilla}})/k_{\text{vanilla}}$, both computed from counts and rounded only for display. These counts belong to the multi-model evaluation and are separate from the repeated GPT-4o controls.

Table 3 is likewise generated from a machine-readable coding summary, with Gain computed only for the Gemini Vanilla–Solution-Evo pair in each language; Appendix F explains the aggregation scope.

B.2 GAIA COMPARISON AND REPORTING CONVENTIONS

Table 2 cites a source beside each external system: a paper or report where available, otherwise a project page, SDK, or the official GAIA leaderboard results (agent-2030 and Su-Zero-Ultra (GAIA, 2025)). The Skywork entry cites the first AgentOrchestra report. External systems use their own backbones, tools, budgets, and evaluation snapshots, so the table provides context rather than a controlled comparison.

Attempt conventions also differ: Alita reports its 87.27 validation result as pass@3 (Qiu et al., 2025). We list the published value without converting it to pass@1.

The official GAIA evaluator checks split membership and per-level counts, and computes the overall score across questions rather than as an unweighted average of level scores (GAIA Benchmark Contributors, 2026). Our within-AGS scores follow these full-split definitions, and Gain entries are computed from the displayed percentages.

C NOTATION

We summarize the main mathematical symbols and their meanings in Table 7. Symbols follow the two-layer structure of **AGP**: resource identities and registration records in RSPL, followed by editable variables, operator messages, and iteration states in SEPL.

Table 7: Notation used in the paper. Grey rows indicate categories.

Symbol	Description
	Indexing and Sets
$\mathcal{T}$	Set of RSPL entity types, $\{$PROMPT, AGENT, TOOL, ENV, MEM$\}$.
τ	Entity type index, $\tau \in \mathcal{T}$.
$\mathcal{I}_\tau$	Index set of registered resources of type τ, not temporary runtime objects.
i	Registered-resource index, $i \in \mathcal{I}_\tau$.
$\mathbb{V}$	Space of version strings.
$\wp(\cdot)$	Power set operator.
	RSPL Resource Entity (Def. G.1)
$e_{\tau,i}$	Resource entity tuple $(n_{\tau,i}, d_{\tau,i}, \phi_{\tau,i}, g_{\tau,i}, m_{\tau,i})$.
$n_{\tau,i}$	Name unique within type τ; $(\tau, n_{\tau,i})$ is the resource identity.
$d_{\tau,i}$	Short description.
$\phi_{\tau,i} : \mathcal{X}_\tau \to \mathcal{Y}_\tau$	Invocation mapping, with runtime state included for stateful resources.
$g_{\tau,i}$	Evolvability marker, $g_{\tau,i} \in \{0, 1\}$, indicating whether the resource is evolvable.
$m_{\tau,i}$	Auxiliary metadata dictionary.
$\mathcal{E}_\tau$	Set of resource entities of type τ.
	RSPL Registration Record (Def. G.2)
$c_{\tau,i}$	Registration record $(e_{\tau,i}, v_{\tau,i}, \eta_{\tau,i}, \theta_{\tau,i}, \mathcal{F}_{\tau,i})$.
$\mathcal{C}_\tau$	Current registration records for type τ; historical records are retained separately.
$v_{\tau,i}$	Version string; $(\tau, n_{\tau,i}, v_{\tau,i})$ identifies a version-qualified record.
$\eta_{\tau,i}$	Implementation descriptor (e.g., import path, class, or source).
$\theta_{\tau,i}$	Instantiation parameters (e.g., constructor arguments).
$\mathcal{F}_{\tau,i}$	Exported representations for LLM interaction (schemas/text/structured args).
	Type-level Resource Interface (Def. G.3)
$\mathcal{R}_\tau$	Type-specific registry mapping identities to active registration records.
$\mathcal{R}$	Global registry, $\bigcup_\tau \mathcal{R}_\tau$, with type-qualified keys.
$\mathcal{M}_\tau$	Context manager for type τ (maintains registry and version lineage).
$\mathcal{A}_\tau$	Server-exposed interface for type τ (delegates to $\mathcal{M}_\tau$).
r_τ	Type-level management interface $(\mathcal{C}_\tau, \mathcal{M}_\tau, \mathcal{A}_\tau)$, not a single instance.
	SEPL Variables, Spaces, and Operators
$\mathcal{V}_{\text{evo}}$	Managed resource entities together with the current task-local output y.
$\mathfrak{V}$	Space of admissible resource/output states, including newly registered instances.
v	A variable in $\mathcal{V}_{\text{evo}}$.
g_v	Evolvability marker for variable v (binary).
Θ	Editable subset, $\{v \in \mathcal{V}_{\text{evo}} \mid g_v = 1\}$.
y	Current task-local output or solution; not an editable execution trace.
$\mathcal{P}$	Message space carrying evidence, provisional candidates, and assessments between operators.
$\mathcal{P}_{\text{in}}, \mathcal{P}_{\text{out}}$	Incoming and outgoing message types of a SEPL operator.

Continued on the next page

Table 7 (continued)

Symbol	Description
f	A SEPL operator, $f : \mathfrak{V} \times \mathcal{P}_{\text{in}} \to \mathfrak{V} \times \mathcal{P}_{\text{out}}$.
$\mathcal{Z}$	Trace space (execution observations).
$\mathcal{H}$	Hypothesis space (proposed explanations of observed failures).
$\mathcal{D}$	Modification space (proposed resource changes).
$\mathcal{G}$	Objective specification (task goals and declared evaluation constraints).
$\mathcal{S}$	Evaluation space (reported metrics and constraint-check outcomes).
$\rho, \sigma, \iota, \varepsilon, \kappa$	Reflect, Select, Improve, Evaluate, and Commit operators (reflection instantiation).
	Optimization Loop (Alg. 1)
A	Agentic system.
T	Optimization budget (number of iterations).
t	Iteration index.
$\mathcal{V}_{\text{evo}}^{(t)}$	Evolvable state at iteration t.
$p^{(t)} \in \mathcal{P}$	Message instance at iteration t; the initial message carries an execution trace.
$Z^{(t)} \in \mathcal{Z}$	Observational trace at iteration t.
$H^{(t)} \subseteq \mathcal{H}$	Hypotheses at iteration t.
$D^{(t)} \subseteq \mathcal{D}$	Proposed modifications at iteration t.
$\widetilde{\mathcal{V}}_{\text{evo}}^{(t+1)}$	Candidate state after applying modifications.
$S^{(t+1)} \in \mathcal{S}$	Evaluation result for the candidate state.
$F \in \mathcal{S}^*$	Ordered candidate-assessment history supplied to reflection; distinct from execution trace Z.

D Code, Prompts, and Resources

Reproducibility materials. Appendix B records the repeated-run counts and configuration; Appendix B.1 specifies optimizer-visible fields. Machine-readable summaries reproduce the reasoning and coding tables and the repeated-run figure. The case figures condense selected execution traces.

LeetCode-derived materials. The public release will contain the generic framework, aggregate experimental settings and results, and methodological descriptions, but no LeetCode-derived problem statements, examples, constraints, starter code, test cases, reference solutions, problem identifiers, fixed splits, or per-problem derived metadata. We will also exclude task-level prompts and traces, platform-specific collection/submission tools, and account credentials. These exclusions limit independent reproduction on the original problem set.

Platform use and data handling. Platform-facing access and evaluation must comply with the applicable LeetCode terms and access conditions (LeetCode, 2024). The experiments use only judge verdicts and execution metadata; hidden tests and official reference solutions are never accessed. Model inference runs through third-party APIs, so traces and requests are subject to the providers' data-handling policies.

D.1 Implementation details

Resource management. Prompt resources expose names, versions, evolvability flags, templates, and rendering variables; the prompt server delegates lifecycle operations to its context manager. The version manager stores version metadata, while retained prompt configurations reside in the prompt context manager. Unregister removes the active prompt binding but retains its local history; cleanup clears that manager's configurations and history. Restoration acts on retained prompt state and is not a system-wide transaction.

Datasets. The reasoning benchmarks contain 198 GPQA-Diamond questions, 30 AIME24 problems, and 30 AIME25 problems (released in two 15-problem parts). The GPQA-Diamond loader shuffles answer options with a fixed preprocessing seed of 42; this is not an LLM decoding seed.

State and feedback boundaries. Benchmark `reset` methods reset the item index and collected results, not every resource registry, memory store, or cache. Task records store the reference answer separately from the question, and label-free callers pass only the fields listed in Table 6 to the optimizer.

E Comparison with Other Protocols

MCP and A2A address communication and interoperability concerns complementary to AGP (Model Context Protocol Contributors, 2025; A2A Protocol Contributors, 2026). Table 8 compares their primary management objects and lifecycle boundaries. A client–server connection, an inter-agent task, and a versioned resource implementation have distinct state transitions, allowing the protocols to be combined within a larger application.

Table 8: Scope comparison. Application-layer extensions can combine these mechanisms; an item outside a standard's primary scope is not a claim that implementations cannot support it.

Concern	AGP	MCP	A2A
Primary object	Editable agent resources and candidate versions	Tools, prompts, and contextual resources	Agents, messages, tasks, and artifacts
Lifecycle focus	Resource registration, evaluation, commit, and retained versions	Client–server sessions and capability interaction	Task states and agent interaction
Evolution focus	Common interface for proposing and recording resource changes	Resource-edit history is an application concern	Internal implementation updates are an application concern
Composition	Can manage a tool or agent exposed through another interface	Can expose capabilities used by AGS	Can connect participants in a larger agent system
Evidence limit	Cases show accepted updates; recovery needs testing	Comparison concerns the cited specification	Comparison concerns the cited specification

AGP focuses on associating resource changes with execution evidence and retained versions; this does not imply that other standards or their implementations lack logging, security, or state management. The comparison concerns the cited interfaces rather than implementation-level performance.

E.1 Comparison with Resource and Workflow Optimizers

Table 9 compares the mechanisms described in the cited papers. It separates what changes, the scope and feedback of adaptation, and the state retained by the method. Several of these methods already retain search history, episodic memory, or reusable tools.

Table 9: Optimization mechanisms and retained state. This is a methodological comparison, not a matched-budget ranking.

Method	**Update target**	**Scope and feedback**	**Retained state**
Self-Refine (Madaan et al., 2023)	Current response	Within-instance revisions using feedback from the same LLM.	Response and feedback history in context.
Reflexion (Shinn et al., 2023)	Reflections guiding later actions	Subsequent trials use verbal reflections on scalar or textual task feedback.	Episodic reflection memory.
TextGrad (Yuksekgonul et al., 2025)	Textual variables in a computation graph	Instance-level solution edits or task-level prompt optimization using backpropagated LLM feedback.	Variable values, graph dependencies, and textual feedback.
MIPRO (Opsahl-Ong et al., 2024)	Module instructions and demonstrations	Program optimization over training examples using downstream metrics and Bayesian search.	Candidate combinations and a learned surrogate objective.
GEPA (Agrawal et al., 2026)	Module prompts; also code-search examples	Trajectory reflection and task evaluation guide prompt search; the paper also studies inference-time search.	Candidate pool, ancestry, and per-instance scores for Pareto selection.

Continued on the next page

Table 9 (continued)

Method	Update target	Scope and feedback	Retained state
AFlow (Zhang et al., 2025b)	Code-represented workflows	Workflow search using validation executions, scores, and feedback on code edits.	Search-tree experience, parent edits, and evaluation records.
EvoAgentX (Wang et al., 2025)	Prompts, tool configurations, and workflow topology	Workflow evaluation and optimization through integrated TextGrad, MIPRO, and AFlow components.	Workflow representations and optimizer-specific state.
Alita (Qiu et al., 2025)	Tools and execution environments	Runtime tool construction and self-correction from execution feedback; generated tools can be reused.	An MCP tool registry and execution logs.
AGP / AGS (this work)	Registered resources and task-local solutions	Within-task reflection and selected resource updates under declared feedback policies.	Resource identities, accepted versions, and trace-linked decisions.

AGP specifies a common resource-update interface, whereas the cited optimizers define proposal or search procedures and the state needed to support them. The distinction is not absolute: for example, GEPA records candidate ancestry and AFlow records parent edits, while Alita explicitly stores generated tools for reuse. An AGP adapter would need to map such state to resource identities, candidate evaluation, and activation decisions; a shared interface alone does not verify that mapping. Our main experiments use reflection, and the alternative adapters remain unevaluated sketches.

AGP's management contract requires rejection to preserve the active binding, while restoration targets a retained internal version. The cases document accepted updates, not recovery after a bad version becomes active. The reported target ablations compare what is revised within AGS; they do not isolate resource management while holding the editor, feedback, and realized budget fixed. Persistent storage alone likewise does not establish transfer to unseen tasks.

F Details of the Self-Evolving Code Agent Benchmark

F.1 Benchmark Design Rationale

The coding evaluation studies feedback-driven revision within a single problem-solving episode. It combines three perspectives: whether the final submission is accepted, how accepted solutions use runtime and memory, and how outcomes vary across programming languages. Online-judge feedback supports iterative revision without exposing hidden test cases or reference solutions to the agent.

Compilation errors, runtime failures, wrong answers, and resource-limit violations provide different signals for revision. The agent can use these signals to propose changes to syntax, interfaces, logic, or algorithmic structure. We evaluate final submissions under a declared reflection budget. Figure 4 presents metrics over task count, rather than successive revisions of one solution.

LeetCode also reports runtime and memory statistics for accepted submissions, including platform-relative percentiles. We use these statistics to describe solution efficiency alongside acceptance counts. They are relative to the platform's submission distributions, which may change over time, and should not be interpreted as measurements against a controlled sample of individual programmers.

The same selected problems are instantiated in Python3, C++, Java, Go, and Kotlin using the corresponding starter interfaces. This setup exposes language-specific compilation, typing, library, and runtime failures while holding the task specifications fixed. Cross-language results therefore characterize the interaction between code generation, execution feedback, and language-specific constraints, rather than treating success in one language as evidence of universal robustness.

F.2 Benchmark Construction

Data Collection. We collected a pool of 3,822 LeetCode problems from publicly visible pages, with statements, public examples, constraints, difficulty labels, topical tags, and starter interfaces. Figure 9

summarizes the topics of a 3,704-record metadata snapshot covering 72 topics. Problems may have multiple tags; the chart counts tag assignments and pools the 52 less frequent topics.

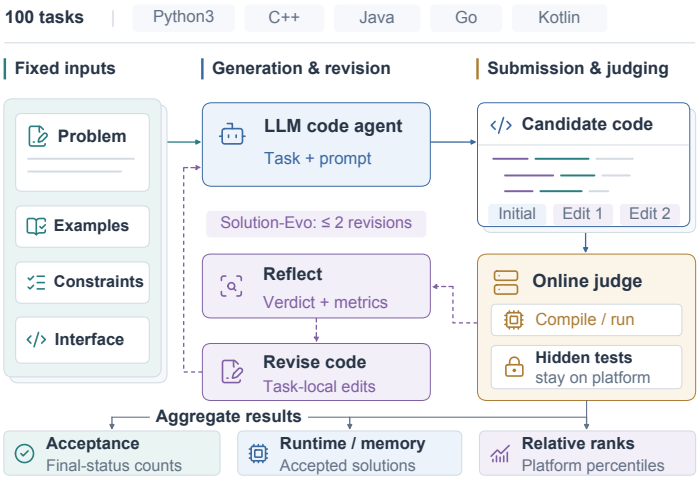


Figure 8: Coding evaluation with feedback-driven revision.

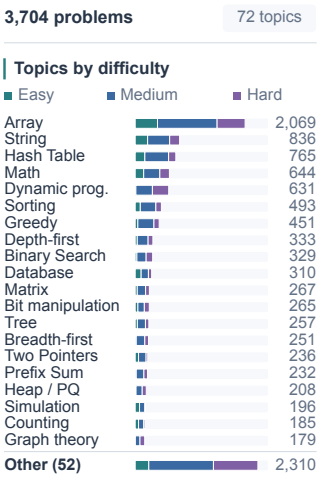


Figure 9: Topic coverage.

Each collected record is normalized into a task specification, official examples, a language-specific starter template, and difficulty and topic metadata. The preprocessing preserves judge-required signatures and class definitions, filters malformed and duplicate records, and checks template availability for the target languages. We select 100 recently released problems from the collected pool and instantiate them across all five languages. This selection aims to reduce contamination risk, but release recency alone cannot establish whether a task was absent from a model's training data.

Problem Characteristics. Each task specifies input and output behavior, constraints, and a required submission interface. The selected categories span different algorithmic and data-structure requirements, while hidden tests can expose boundary cases and implementation errors. The common task specifications support comparisons across languages; they do not eliminate language-specific compiler or runtime effects.

Problem Evaluation. For each problem, the agent receives the task representation and submits code to the platform judge, which checks hidden tests and reports the available execution statistics. The reported Solution-Evo budget permits at most three submissions: the initial solution and up to two feedback-driven revisions, with the task specification and submission interface held fixed. Vanilla uses the initial solution without these revisions. This comparison measures the combined effect of feedback-driven revision and additional attempts, rather than an equal-computation advantage.

F.3 Evaluation Metrics

We report three groups of metrics that capture complementary aspects of code-agent performance. Capability metrics evaluate functional correctness and diagnose execution-blocking failure modes. PR measures the number of fully accepted problems, while TLE, MLE, CE, RE, WA, TO, and RpE identify whether failures arise from algorithmic inefficiency, excessive memory use, compilation errors, runtime exceptions, incorrect logic, response timeout, or invalid output formatting. These metrics are particularly important for self-evolving agents because different failure modes correspond to different refinement opportunities.

AR and AM summarize runtime and memory over each method's accepted solutions. Because those sets can differ, these aggregates do not isolate per-problem efficiency; a common-solved-set paired analysis is required. APC measures passed-test coverage rather than resource efficiency, with larger values denoting more passed cases in the reported aggregate. Its interpretation also depends on which submissions contribute and how many tests each task contains.

Table 10: Evaluation metrics for the algorithmic coding benchmark.

Metric	Description
	Capability metrics
PR	Number of problems passing all hidden test cases within time and memory limits.
TLE	Number of problems exceeding the allowed execution time limit.
MLE	Number of problems exceeding the allowed memory usage.
CE	Number of problems where generated code failed to compile.
RE	Number of problems encountering a runtime error during execution.
WA	Number of problems producing incorrect output.
TO	Number of problems where the model failed to respond within the timeout.
RpE	Number of problems where the model returned an invalid or unparseable response.
	Execution metrics
AR	Mean runtime in milliseconds over accepted solutions.
AM	Mean memory usage in megabytes over accepted solutions.
APC	Mean number of test cases passed before failure.
	Submission-relative metrics
ARB	Mean platform-reported runtime-beats percentile over accepted solutions.
AMB	Mean platform-reported memory-beats percentile over accepted solutions.

Platform-relative metrics aggregate the reported runtime and memory percentiles of accepted submissions. These percentiles describe a changing submission distribution rather than a controlled human baseline. Their interpretation depends on the accepted set and aggregation procedure.

Table 3 reports aggregate coding metrics. In Figure 4, the horizontal axis is task count; the panels show acceptance rate, cumulative runtime, and submission-relative percentiles. These are not per-round learning curves or evidence of cross-task transfer, and their shape depends on task order.

Gain conventions and aggregation scope. For a nonzero Vanilla value b and Solution-Evo value e, Table 3 reports $100|e - b|/b$. Arrows denote the direction of benefit: increases are favorable for PR, APC, ARB, and AMB, whereas decreases are favorable for error counts, AR, and AM. Zero-to-zero entries are shown as no change; a zero-to-nonzero transition has no defined relative percentage. In particular, Java TLE increases from zero to one and is marked with a dash, not a zero gain. Values displayed as 0.0 indicate changes that round to zero at the stated precision.

Status categories are reported as recorded and need not sum to the number of evaluated problems, so we do not normalize them into error distributions. Gain compares corresponding metrics within each same-backbone Vanilla–Solution-Evo pair, not necessarily identical submissions or accepted sets.

G Details of Self-Evolution Protocol

G.1 Design Motivation

The design separates update proposals from resource management. An optimizer needs an identified target, a description of permitted edits, candidate evaluation, and a recorded activation decision. RSPL represents the target and retained versions; SEPL organizes those operations into an update loop. This separation permits common bookkeeping across resource types, while useful recovery additionally depends on which state is captured, which consumers bind to a version, and whether external effects occur.

G.2 Layer 1: Resource Substrate Protocol Layer

RSPL represents instructions (Prompt), decision logic (Agent), callable capabilities (Tool), executable environment models (Environment), and retention/retrieval implementations (Memory). Tool includes local code, MCP-connected capabilities (Model Context Protocol Contributors, 2025), and skills (Anthropic, 2025). The resource representation does not prescribe its own update strategy. Execution changes task-local state, whereas resource updates revise stored implementations or configurations.

G.2.1 CORE ENTITIES

We focus on these five entity types as the supported resource substrate for agentic systems. This choice is not intended to be exhaustive, but rather to identify a common denominator across modern agent stacks and provide a uniform target space on which SEPL can operate.

Definition G.1 (Resource Entity)**.** For type τ, the entity and its collection are

$$\begin{aligned} e_{\tau,i} &= (n_{\tau,i},\, d_{\tau,i},\, \phi_{\tau,i},\, g_{\tau,i},\, m_{\tau,i}), \\ \mathcal{E}_\tau &= \{\, e_{\tau,i} \mid i \in \mathcal{I}_\tau \,\}, \end{aligned} \tag{6}$$

where $\tau \in \mathcal{T} = \{\text{PROMPT}, \text{AGENT}, \text{TOOL}, \text{ENV}, \text{MEM}\}$ indexes resource types and $i \in \mathcal{I}_\tau$ indexes registered resources. Each entity has a name $n_{\tau,i}$ unique within its type, description $d_{\tau,i}$, input-to-output mapping $\phi_{\tau,i} : \mathcal{X}_\tau \to \mathcal{Y}_\tau$, evolvability marker $g_{\tau,i} \in \{0, 1\}$, and auxiliary metadata $m_{\tau,i}$. For stateful resources, invocation inputs and outputs include relevant runtime state; the mapping does not assume pure or stateless execution.

Representing prompts, tools, and memory as explicit resources separates agent control logic from the instructions and capabilities it consumes. A tool-calling agent can then retain its implementation while selecting different prompt or tool resources. The intended benefit is independent management and reuse of these components; whether an accepted resource improves performance on a new task remains an empirical question.

To support resource registration, unified management, and instantiation, RSPL stores a serializable registration record for each resource instance.

Definition G.2 (Resource Registration Record)**.** A resource registration record and its type-level collection can be represented as:

$$\begin{aligned} c_{\tau,i} &= (e_{\tau,i},\, v_{\tau,i},\, \eta_{\tau,i},\, \theta_{\tau,i},\, \mathcal{F}_{\tau,i}), \\ \mathcal{C}_\tau &= \{\, c_{\tau,i} \mid i \in \mathcal{I}_\tau \,\}, \end{aligned} \tag{7}$$

where $\tau \in \mathcal{T}$ indexes the entity type and $i \in \mathcal{I}_\tau$ indexes an individual instance. Here $e_{\tau,i}$ is the resource entity tuple defined in Definition G.1, $v_{\tau,i} \in \mathbb{V}$ is a version string, $\eta_{\tau,i}$ is an implementation descriptor (e.g., import path, class definition, or source-code string), $\theta_{\tau,i}$ are instantiation parameters (e.g., constructor arguments), and $\mathcal{F}_{\tau,i}$ is a set of exported representations used by LLMs to interact with the resource (e.g., function-calling schema, plain text, and structured argument schema).

Definition G.3 (Type-level resource interface)**.** For each entity type τ, the registry $\mathcal{R}_\tau$ maps typed identities $(\tau, n_{\tau,i})$ to active records in $\mathcal{C}_\tau$, while historical records remain separate restoration targets. The global registry $\mathcal{R} = \bigcup_\tau \mathcal{R}_\tau$ uses type-qualified keys. RSPL binds each type to a context manager $\mathcal{M}_\tau$ and server interface $\mathcal{A}_\tau$. The type-level management interface, not an individual instance, is

$$r_\tau = (\mathcal{C}_\tau,\ \mathcal{M}_\tau,\ \mathcal{A}_\tau), \tag{8}$$

where each $c_{\tau,i} \in \mathcal{C}_\tau$ is a registration record in Definition G.2. The context manager $\mathcal{M}_\tau$ maintains the collection $\mathcal{C}_\tau$, the version lineage for type τ, and implements lifecycle and update operations over these records; the server-exposed interface $\mathcal{A}_\tau$ encapsulates $\mathcal{M}_\tau$ and exposes a unified external interface by delegating requests to the corresponding context-manager routines.

Management interface versus invocation contract. The triple r_τ describes management at the resource-type level; $\phi_{\tau,i}$ describes invocation of an individual resource. A shared lifecycle does not require all tools or agents to accept the same arguments. The spaces $\mathcal{X}_\tau$ and $\mathcal{Y}_\tau$ collect the admissible type-level inputs and outputs, while each resource's exported representations $\mathcal{F}_{\tau,i}$ specify its own accepted schema. Consumers must resolve that contract before invocation. Updating an implementation without changing its callable schema preserves this part of the consumer interface; changing the schema may require corresponding consumer changes.

Identity, version, and runtime instance. The pair $(\tau, n_{\tau,i})$ identifies a registered resource; $(\tau, n_{\tau,i}, v_{\tau,i})$ identifies a version-qualified record. Here $c_{\tau,i}$ denotes the current record, while retained historical records belong to the manager's history rather than additional entries in the active registry. A version update preserves the resource identity. Constructing another runtime object from a record also does not create a new registered resource; registration under a different typed name does. Typed names distinguish identically named resources across separate registries.

G.2.2 CONTEXT MANAGER

The context manager maintains the active registry and retained version history. Table 11 defines the protocol's operation vocabulary, not a list of identically named Python methods implemented by every resource type. An adapter maps supported operations to a type's native interface and declares unavailable capabilities. Exported schemas and text describe resource interaction, including local tools, MCP-connected capabilities (Model Context Protocol Contributors, 2025), and skills (Anthropic, 2025). Such descriptions inform consumers about arguments and intended use; they do not certify correctness or enforce the stated restrictions.

Table 11: Conceptual RSPL management operations. Native names and availability vary by type; optional operations require explicit adapter support.

Operator	Description
	Lifecycle & Registration
`init`	Discover and initialize configured resources; exposed as `initialize` in AGS.
`build`	Construct a runtime object from its record; not a separate method on every server.
`register`	Add a resource under a typed name with an associated version.
`unregister`	Remove the active registration. History retention is manager-specific; deletion requires a separate policy, and deleted history cannot be restored.
	Retrieval & Inspection
`get`	Retrieve a resource instance by name from the active registry.
`get_info`	Retrieve a resource configuration by name from the active registry.
`list`	List all registered resource names.
`retrieve`	Optional similarity-based resource lookup; not natively implemented in AGS.
`get_state`	Get the current state of a resource instance when supported.
	Versioning
`update`	Request an implementation/configuration change under the manager's version policy.
`copy`	Duplicate a resource with an optional new name and version.
`restore`	Restore a specific historical version by name and version string.
`get_variables`	Adapter-level projection of permitted fields from a registration record.
`set_variables`	Adapter-level application of a candidate projection, subject to evaluation and commit.
	Execution
`run`	Invoke the type-specific execution surface; not a shared native method signature.
	Serialization
`save_to_json`	Optional export of configurations/history to a declared persistence format.
`load_from_json`	Optional import with type, version, and artifact checks supplied by the adapter.
`save_contract`	Optional persistence of the exported interaction description.
`load_contract`	Optional loading of that description, not activation of its implementation.

G.2.3 SERVER INTERFACE

In AGS, the server classes are local Python facades over context managers rather than a network transport or wire protocol. All five directly declare `initialize`, `register`, `get`, `get_info`, `list`, `update`, `unregister`, and `cleanup`. Their argument and return types vary: this is not a uniform server API. Table 12 maps representative execution and version operations to these declarations.

The conceptual names `retrieve`, `get_variables`, `set_variables`, and the four serialization operations in Table 11 are not natively declared by these five server/context pairs; an integration must supply the corresponding adapters before advertising those capabilities. In particular, Memory restoration goes through its context manager rather than a server method.

G.2.4 INFRASTRUCTURE SERVICES

RSPL further includes cross-cutting services that support resource access, serialization, traceability, and retained-version recovery:

Table 12: Native resource interfaces in AGS.

Resource	Execution / interaction examples	Copy and restore access
Prompt	`__call__`, `get_system_message`, `get_agent_message`	Declared by server and context manager
Agent	`__call__`, `function_callings`, `get_schema`	Declared by server and context manager
Tool	`__call__`, `function_callings`, `get_schema`	Declared by server and context manager
Environment	`__call__`, `get_state`, `get_schema`	Declared by server and context manager
Memory	`start_session`, `add_event`, `get_state`	Declared by context manager, not directly by server

Model manager. Routes calls by an explicit model name or configured role, with retry handling, a configured fallback, and usage logging.

Version manager. Tracks resource version identifiers and lineage metadata. Recovery additionally requires retained configurations and implementation artifacts in the relevant resource manager or artifact store. A version string alone does not ensure an immutable snapshot or a failure-atomic update; these properties depend on the storage and activation implementation.

Dynamic manager. Loads implementation classes from source strings or files into Python modules and supports schema/argument conversion. Resource managers use these facilities when constructing objects from records. Loading a new class does not by itself migrate existing instance state or rebind every consumer; activation and persistence remain separate management responsibilities.

Trace manager. Records execution observations and links them to update decisions. Raw traces are not used for training or dataset synthesis by default; deployments must decide which fields to capture, redact, retain, and delete.

Recovery consistency requirements. A complete restoration must realign the active configuration, the version metadata, and subsequent consumer resolution; restoring only one does not establish a consistent recovered state. Similarly, an interrupted update must not expose a candidate merely because its configuration was assigned before version registration completed. Version identities must preserve their associated retained content if they are to serve as restoration targets.

G.3 Layer 2: Self-Evolution Protocol Layer

SEPL composes proposal, evaluation, and commit operations over the resource interface. The reflection instantiation below is used in the main experiments. The subsequent textual-feedback and reward-informed adapters illustrate alternative mappings; they remain unevaluated extensions rather than evidence of optimizer-independent performance or standard RL training.

G.3.1 Evolvable Variables

Variable lifting resolves each resource entity to editable fields of its registration record, such as an implementation descriptor η, instantiation parameters θ, or a prompt template. The entity in $\mathcal{V}_{\text{evo}}$ identifies the target; it does not limit editing to the five fields of the entity tuple. The evolvability marker selects eligible resources, and the manager specifies the permitted field projection. Identity, version bookkeeping, and edit permissions are not unrestricted candidate fields. This representation supports heterogeneous discrete edits without assuming a differentiable parameter space.

Definition G.4 (Evolvable Variable Set)**.** We define the universal set of evolvable variables as

$$\mathcal{V}_{\text{evo}} = \Big(\bigcup_{\tau \in \mathcal{T}} \mathcal{E}_\tau \Big) \cup \{y\}, \tag{9}$$

where $\mathcal{E}_\tau$ denotes managed resources and y denotes the current task-local output/solution. Traces remain observations, not mutable evidence. A marker $g_v \in \{0, 1\}$ identifies the editable subset

$$\Theta = \{v \in \mathcal{V}_{\text{evo}} \mid g_v = 1\}. \tag{10}$$

Let $\mathfrak{V}$ denote the space of admissible resource/output states, including states with newly registered instances. An iteration state $\mathcal{V}_{\text{evo}}^{(t)} \in \mathfrak{V}$ records the current values; it is not the operator's domain by itself. Evolvability markers restrict edit targets through the manager, not operating-system permissions. Frozen implementations may still execute, and ordinary episode-state changes are distinct from implementation updates.

For example, adding an event to Memory changes stored episode data, whereas revising its retention policy changes a managed resource configuration. Likewise, advancing an Environment changes its current state, whereas revising transition rules changes its implementation. The former operations do not automatically create SEPL versions. Restoring a policy or rule implementation also does not rewind episode data or external effects unless those states were separately captured and restored. Creating a new resource requires a permitted registration operation; an editable target flag is not blanket authorization to execute generated code, access files, or transmit data.

G.3.2 Operator Composition

Definition G.5 (SEPL Operator)**.** Let $\mathfrak{V}$ be the state space and $\mathcal{P}$ the message space carrying execution evidence, proposed edits, provisional candidates, and evaluation outcomes. A *SEPL operator* has the interface

$$f : \mathfrak{V} \times \mathcal{P}_{\text{in}} \;\to\; \mathfrak{V} \times \mathcal{P}_{\text{out}}, \tag{11}$$

where $\mathcal{P}_{\text{in}}, \mathcal{P}_{\text{out}} \subseteq \mathcal{P}$ denote message types. An invocation returns a state $\mathcal{V}'_{\text{evo}} \in \mathfrak{V}$ and a message $p' \in \mathcal{P}_{\text{out}}$, not the spaces themselves. Composition requires compatible types and preconditions. The notation describes dataflow, not deterministic proposals; traceability and restoration additionally require recorded transitions and retained artifacts.

Operators use five auxiliary spaces: traces $\mathcal{Z}$, diagnostic hypotheses $\mathcal{H}$, proposed modifications $\mathcal{D}$, objective specifications $\mathcal{G}$, and evaluation outcomes $\mathcal{S}$. The reflection instantiation composes $\{\rho, \sigma, \iota, \varepsilon, \kappa\}$ for Reflect, Select, Improve, Evaluate, and Commit. Hypotheses are proposed explanations of observed failures; evaluation outcomes record the results of the declared checks.

Retained state and candidate payload. For the reflection instantiation, write V for the retained resource/output state and $\widetilde{V}$ for a provisional candidate. A message may carry the candidate, its base resource/version references, and its assessment without replacing V. The proposal and evaluation wrappers therefore pass V through unchanged. Using p_D for a message containing selected edits and $p_{\widetilde{V},S}$ for a candidate-assessment message, representative wrappers are

$$\begin{aligned} f_\iota(V, p_D) &= (V, p_{D,\widetilde{V}}), \qquad \widetilde{V} = \iota(V, D), \\ f_\kappa(V, p_{\widetilde{V},S}) &= \begin{cases} (\kappa(\widetilde{V}, S), p_S), & \text{Accept}(S), \\ (V, p_S), & \text{otherwise.} \end{cases} \end{aligned} \tag{12}$$

Here message subscripts identify relevant contents, not an exhaustive wire format; p_S retains the outcome for subsequent assessment history. Primitive signatures below abbreviate candidate computations, whereas the wrappers specify how those computations compose with the retained state. Sandboxing evaluation code is left to the deployment.

G.3.3 Evolutionary Loop

Initial execution under $\mathcal{V}_{\text{evo}}^{(0)} \in \mathfrak{V}$ produces a trace $Z^{(0)} \in \mathcal{Z}$, carried by $p^{(0)} \in \mathcal{P}$. Each iteration composes compatible operators:

$$\big(\mathcal{V}_{\text{evo}}^{(t+1)},\, p^{(t+1)}\big) = (f_n \circ \cdots \circ f_1)\big(\mathcal{V}_{\text{evo}}^{(t)},\, p^{(t)}\big), \tag{13}$$

where the reflection cycle includes proposal, evaluation, commit, and task-dependent continuation under the retained state. Algorithm 1 abbreviates the state as V and counts update iterations after initial execution, not total model calls. It returns the retained state for early stopping or zero iterations. In contrast, the opportunity-controlled experiment permits at most three primary attempts; the generic management loop is not its exact call schedule.

Reflection produces hypotheses from traces, selection chooses permitted targets, and improvement constructs a candidate. Evaluation applies the task-specific feedback policy; commit records an

accepted resource version or updates the task-local solution. Rejection must preserve the active state. Restoration targets retained resources; solution editing only revises task-local y.

G.3.4 REFLECTION OPTIMIZER

Evolvable Variables. The reflection optimizer edits RSPL-managed resources and, when enabled, the current task-local solution y. The evolvability mask restricts the selected targets while frozen implementations remain available for execution. Execution traces supply diagnostic evidence and are not retroactively modified as optimization variables.

Operator interfaces. We instantiate SEPL with the canonical reflection-driven operator suite. The signatures below abbreviate primitive computations; their outputs are carried by the wrappers above rather than unconditionally replacing the retained state. Their intended roles are as follows.

- **Reflect (ρ).** $\rho : \mathcal{Z} \times \mathfrak{V} \times \mathcal{S}^* \to \wp(\mathcal{H})$ maps execution observations, the retained state, and prior candidate assessments to hypotheses. Here $\mathcal{S}^*$ denotes finite sequences of assessment records, including rejected candidates. These hypotheses are proposed explanations, not established causal attributions or numerical gradients.
- **Select (σ).** $\sigma : \mathfrak{V} \times \wp(\mathcal{H}) \to \wp(\mathcal{D})$ selects targets implicated by the hypotheses and proposes modifications $D \subseteq \mathcal{D}$, restricted to the editable subset Θ. Target selection does not itself activate a resource change.
- **Improve (ι).** $\iota : \mathfrak{V} \times \wp(\mathcal{D}) \to \mathfrak{V}$ applies proposed edits D to construct a provisional candidate. The candidate remains distinct from the active state until the declared acceptance condition is checked.
- **Evaluate (ε).** $\varepsilon : \mathfrak{V} \times \mathcal{G} \to \mathcal{S}$ evaluates a candidate under a declared task-specific feedback policy. In the label-free reasoning setting, $\mathcal{S}$ contains self-assessed satisfaction and a rationale, not benchmark correctness or a universal safety certificate.
- **Commit (κ).** $\kappa : \mathfrak{V} \times \mathcal{S} \to \mathfrak{V}$ records an accepted resource version or updates task-local y. It is invoked only after acceptance; rejection must preserve the active state. A satisfaction gate can be wrong and does not enforce true-performance monotonicity. Restoring a previously active version is a separate operation over retained state.

The Evolutionary Loop. Algorithm 1 composes the five operators. Initial execution produces a trace $Z \in \mathcal{Z}$ of tool outputs, decisions, failures, and progress. Reflection also receives the assessment history F, so a rejected candidate can inform a later proposal without replacing the active resource or rewriting the execution trace. Improvement constructs a candidate and evaluation supplies the declared checks; commit retains an accepted edit. Stopping is checked before any further task execution. The retained version history identifies potential restoration targets but does not itself demonstrate successful recovery.

Algorithm 1 Reflection-driven resource update (management specification)

Require: System A, objective/feedback policy $\mathcal{G}$, maximum update iterations T
Ensure: Retained resource/output state V

$V \leftarrow \mathrm{ExposeResourceState}(A)$ {include y and edit markers}
$(V, Z) \leftarrow \mathrm{Execute}(A, V)$ {refresh task-local y and trace}
$F \leftarrow [\,]$ {candidate assessments, separate from trace Z}
for $t = 1, \ldots, T$ **do**
  if $\mathrm{Stop}(V, Z, F)$ **then**
    break
  end if
  $H \leftarrow \mathrm{Reflect}(Z, V, F)$
  $D \leftarrow \mathrm{Select}(V, H)$ {respect evolvability markers}
  $\widetilde{V} \leftarrow \mathrm{ImproveCandidate}(V, D)$
  $S \leftarrow \mathrm{EvaluateCandidate}(\widetilde{V}, \mathcal{G})$
  if $\mathrm{Accept}(S)$ **then**
    $V \leftarrow \mathrm{Commit}(\widetilde{V}, S)$
  else

Algorithm 1 Reflection-driven resource update (management specification) (continued)

```
    Keep V unchanged; record candidate rejection
  end if
  F ← AppendAssessment(F, S)
  if Stop(V, Z, F) or t = T then
    break
  end if
  if NeedsExecution(V, D, S) then
    (V, Z) ← ContinueTask(A, V, Z)
  end if
end for
return V
```

Continuation and stopping semantics. The objective/feedback policy defines STOP and NEEDSEXECUTION. Stopping concerns the retained state, not an uncommitted candidate. A completed solution-only edit retains its accepted y; it does not trigger another unconstrained answer generation. When resource changes require further execution, CONTINUETASK resumes the task using the retained resources and treats any accepted solution edit as part of its input. Only actual execution appends execution observations; candidate assessments remain separate diagnostic records.

The routine returns a managed state rather than guaranteeing task completion. A caller must schedule any remaining task execution within its overall budget. Candidate evaluation, proposal generation, stopping checks, and continuation can incur model or tool calls and must be included in accounting. The candidate/active-state boundary does not undo completed external actions.

G.3.5 TEXTUAL-FEEDBACK ADAPTER

Scope. This adapter is inspired by TextGrad (Yuksekgonul et al., 2025); distributing a shared critique to prompt variables is not evidence of reproducing every part of the original method. It is not used to establish the reported reflection results. The sketch synchronizes edits without a post-edit acceptance gate and therefore does not implement the complete evaluation-gated lifecycle.

Evolvable Variables. In this proposed adapter, optimizable modules are represented by prompt-valued TextGrad variables with role descriptions. Each value contains the current prompt text; its role describes the module's function and guides the textual rewrite. Restricting this sketch to prompts is an adapter choice, not a limitation of TextGrad's support for other variables in a computation graph.

Operator interfaces. The signatures in these extension sketches abbreviate invocations by listing their arguments; resource/output states lie in $\mathfrak{V}$, and sampled traces lie in $\mathcal{Z}$. Our mapping uses five operators: *Execute*, *Loss*, *Backward*, *Improve*, and *Commit*. Here the gradient is a natural-language critique, not a numeric derivative. Commit is our synchronization step for RSPL-managed prompts, not an interface specified by TextGrad. Sharing a critique provides neither graph-specific credit assignment nor an acceptance gate.

- **Execute (χ_{tg}).** $\chi_{\mathrm{tg}} : (A, \mathcal{V}_{\mathrm{evo}}, x, f) \to \mathcal{Z}$ runs the agent under the current prompt variables and produces an execution trace/outcome.
- **Loss (λ_{tg}).** $\lambda_{\mathrm{tg}} : \mathcal{Z} \to \mathcal{G}_{\mathrm{tg}}$, where $\mathcal{G}_{\mathrm{tg}}$ is a space of natural-language critiques (textual gradients). In this adapter, λ_{tg} is realized by `TextLoss`, which queries an evaluator LLM for feedback.
- **Backward (β_{tg}).** $\beta_{\mathrm{tg}} : \mathfrak{V} \times \mathcal{G}_{\mathrm{tg}} \to \mathfrak{V}$ assigns textual gradients to optimizable prompt variables by storing the critique (optionally with context) in a per-variable gradient buffer. The sketch distributes the same critique to each optimizable prompt variable.
- **Improve (ι_{tg}).** $\iota_{\mathrm{tg}} : \mathfrak{V} \to \mathfrak{V}$ rewrites prompt variables via a textual-gradient-descent step: it constructs an update instruction from each variable's role description, current value, and accumulated textual gradients, then queries an optimizer LLM and extracts the improved variable text from a constrained output format.
- **Commit (κ_{tg}).** $\kappa_{\mathrm{tg}} : \mathfrak{V} \to \mathfrak{V}$ synchronizes the updated prompt variables back into the running agent and clears caches, synchronizing the edited fields with the managed resource configuration.

The Evolutionary Loop. Algorithm 2 executes the current prompts to obtain $Z \in \mathcal{Z}$ and an evaluator critique $g \in \mathcal{G}_{\mathrm{tg}}$. It checks the stopping condition on those prompts before modifying them. Otherwise, the adapter shares the critique across prompt-variable buffers, rewrites the prompts, and synchronizes the edited values before the next execution. This ordering avoids an extra rewrite after a stopping decision; it does not add a post-edit acceptance gate. The final update at budget exhaustion may therefore remain unevaluated.

Algorithm 2 Textual-feedback adapter sketch (without a post-edit gate)

Input: Agentic system $\mathcal{A}$, task x, optional attachments f, update budget $K \geq 0$, evaluator/optimizer LLMs $M_{\text{eval}}, M_{\text{opt}}$
Output: Updated state $\mathcal{V}^{*}_{\text{evo}}$ (prompt variables updated via TextGrad)

// Phase 0: Setup
Set backward engine to M_{eval} ▷ *Evaluator used by TextLoss*
$\mathcal{V}^{(0)}_{\text{evo}} \leftarrow \text{VariableLifting}(\mathcal{A})$ ▷ *Lift optimizable prompts to TextGrad variables*
$\ell \leftarrow 0$
Initialize textual optimizer with M_{opt} ▷ *TextualGradientDescent over prompt vars*
// Optimization Cycle
for $k = 0, 1, \ldots, K-1$ **do**
  // Phase 1: Execute (Forward)
  $Z^{(k)} \leftarrow \chi_{\mathrm{tg}}(\mathcal{A}, \mathcal{V}^{(k)}_{\text{evo}}, x, f)$ ▷ *Run agent with current prompts*
  // Phase 2: Loss (Textual Gradient)
  Build evaluation instruction from $Z^{(k)}$ ▷ *Condition on success/error*
  $g^{(k)} \leftarrow \lambda_{\mathrm{tg}}(Z^{(k)})$ ▷ *TextLoss produces critique string*
  if $\text{Converged}(g^{(k)})$ **then**
    break
  end if
  // Phase 3: Backward (Assign Gradients)
  $\mathcal{V}^{(k)}_{\text{evo}} \leftarrow \beta_{\mathrm{tg}}(\mathcal{V}^{(k)}_{\text{evo}}, g^{(k)})$ ▷ *Assign critique to gradient buffers*
  // Phase 4: Improve (Textual Gradient Descent)
  $\widetilde{\mathcal{V}}^{(k+1)}_{\text{evo}} \leftarrow \iota_{\mathrm{tg}}(\mathcal{V}^{(k)}_{\text{evo}})$ ▷ *Rewrite prompts via textual GD*
  // Phase 5: Commit & Next Iteration
  $\mathcal{V}^{(k+1)}_{\text{evo}} \leftarrow \kappa_{\mathrm{tg}}(\widetilde{\mathcal{V}}^{(k+1)}_{\text{evo}})$ ▷ *Sync back; clear caches*
  $\ell \leftarrow k+1$
end for
return $\mathcal{V}^{(\ell)}_{\text{evo}}$

G.3.6 Reward-informed Editing Sketch (Reinforce++-inspired)

Scope and supervision. This is a heuristic resource-editing sketch, not standard Reinforce++ training (Hu, 2025). Text similarity is not a token-probability ratio, and no policy-gradient update is defined. The labeled inputs y^{*} and y_{sft} belong to a separate supervised setting, not the label-free GPQA-Diamond/AIME reflection results. The sketch also lacks a separate post-edit acceptance evaluation. It is retained as an unvalidated extension description, not empirical evidence.

Evolvable Variables. The sketch uses reward-derived diagnostics to propose edits to selected prompt and tool resources, with optional revision of the current solution. These diagnostics condition the editor rather than update model parameters through policy gradients.

Operator interfaces. The following heuristic combines a clipped score and a reference-text penalty to condition an editor. These quantities should not be interpreted as a likelihood-ratio objective for a trainable language-model policy.

Let $\eta(a, b) \in [0, 1]$ be a declared text-similarity measure, with finite task rewards, penalty weight $\beta \geq 0$, clipping parameter $0 \leq \epsilon < 1$, and numerical floor $0 < \epsilon_0 \leq 1$. Update budgets are nonnegative integers. These specify the admissible inputs, not fitted experimental hyperparameters. The same notation applies to the group-based sketch below.

The editable-resource projection $\mathfrak{V}_{\mathrm{edit}}$ contains only enabled persistent fields; GetEditable excludes task-local y. The editor's projection space $\mathfrak{U}$ contains either such a resource projection or a

task-local solution. REPLACEFIELDS and SETOUTPUT insert the respective edits into the full state. This convention also governs the group-based sketch; solutions remain task-local.

- **Sample (χ_{rpp}).** $\chi_{\text{rpp}} : (A, \mathcal{V}_{\text{evo}}, x, f) \to \mathcal{Z}$ samples a rollout under the current resources and yields an execution trace containing the produced answer.
- **Reward (ε_{rpp}).** $\varepsilon_{\text{rpp}} : (y^{(t)}, y^{(t-1)}, y^*, y_{\text{sft}}) \to (r^{(t)}, A^{(t)}, J^{(t)}, \pi^{(t)})$ computes the diagnostic signal tuple from the current solution $y^{(t)}$. Here $r^{(t)}$ is a task reward comparing $y^{(t)}$ with y^*, and $\pi^{(t)}$ is a heuristic score based on text similarity $\eta(\cdot,\cdot)$ as $\pi^{(t)} \triangleq \eta(y^{(t-1)}, y^{(t)})$ (the sketch does not use token-level probabilities). We define a penalty to a reference solution y_{sft} as $\text{pen}^{(t)} \triangleq \beta \left|\log \max(\eta(y_{\text{sft}}, y^{(t)}), \epsilon_0)\right|$ and set $A^{(t)} \triangleq r^{(t)} - \text{pen}^{(t)}$. The heuristic clipped score is
$$J^{(t)} \triangleq \min\big(\pi^{(t)} A^{(t)},\ \bar{\pi}^{(t)} A^{(t)}\big), \quad \bar{\pi}^{(t)} \triangleq \text{clip}(\pi^{(t)}, 1-\epsilon, 1+\epsilon).$$
- **Diagnose (δ_{rpp}).** $\delta_{\text{rpp}} : (\mathcal{Z}, \mathcal{V}_{\text{edit}}, r^{(t)}, A^{(t)}, J^{(t)}, \pi^{(t)}) \to \mathcal{H}$ produces an edit-oriented diagnosis that is explicitly conditioned on the reward-derived metrics and the execution trace.
- **Improve (ι_{rpp}).** $\iota_{\text{rpp}} : \mathfrak{U} \times \mathcal{H} \to \mathfrak{U}$ edits the selected resource projection or, when enabled, the task-local solution. The output retains the input projection's type; unselected fields remain unchanged.
- **Commit (κ_{rpp}).** $\kappa_{\text{rpp}} : \mathfrak{V}_{\text{edit}} \to \mathfrak{V}_{\text{edit}}$ synchronizes proposed persistent edits with RSPL without a separate post-edit acceptance test. Task-local solution edits use SETOUTPUT instead.

The Evolutionary Loop. Algorithm 3 organizes the heuristic into reward computation, resource editing, re-execution, optional solution revision, and early stopping. The clipped similarity score and reference-text penalty provide diagnostic inputs to the editor. No likelihood-ratio estimate or policy-gradient update is defined. Optional solution edits update y in the task state, not a persistent registry entry; the recorded execution trace remains unchanged.

The reward tuple at iteration t evaluates the pre-edit solution $y^{(t)}$. When reused after a resource update, it is historical diagnostic context, not a score for the newly generated or rewritten solution. The declared SATISFIED check must assess the current $y^{(t+1)}$; earlier trace success alone cannot certify a later text edit. Any new model or execution calls made by this check count toward the overall cost, even though T counts update iterations only.

Algorithm 3 Heuristic reward-informed editing sketch (not standard Reinforce++)

Input: Agentic system $\mathcal{A}$, task x, optional attachments f, ground truth y^*, reference solution y_{sft}, update budget $T \geq 0$
Output: Final solution $y^{(\ell)}$ and updated editable resources $\mathcal{V}_{\text{edit}}$

// Initialization
$\mathcal{V}_{\text{evo}}^{(0)} \leftarrow \text{VariableLifting}(\mathcal{A})$ ▷ *Expose editable resources*
$Z^{(0)} \leftarrow \chi_{\text{rpp}}(\mathcal{A}, \mathcal{V}_{\text{evo}}^{(0)}, x, f)$ ▷ *Sample once*
Extract solution $y^{(0)}$ from $Z^{(0)}$
$\mathcal{V}_{\text{evo}}^{(0)} \leftarrow \text{SetOutput}(\mathcal{V}_{\text{evo}}^{(0)}, y^{(0)})$
$\ell \leftarrow 0$
$y^{(-1)} \leftarrow y^{(0)}$ ▷ *Initialize previous solution*
for $t = 0, 1, \ldots, T-1$ **do**
  // Phase 1: Reward and heuristic similarity score
  $(r^{(t)}, A^{(t)}, J^{(t)}, \pi^{(t)}) \leftarrow \varepsilon_{\text{rpp}}(y^{(t)}, y^{(t-1)}, y^*, y_{\text{sft}})$
  // Phase 2: Improve editable resources (prompt and tool)
  $\mathcal{V}_{\text{edit}}^{(t)} \leftarrow \text{GetEditable}(\mathcal{V}_{\text{evo}}^{(t)})$
  $H_{\text{edit}}^{(t)} \leftarrow \delta_{\text{rpp}}(Z^{(t)}, \mathcal{V}_{\text{edit}}^{(t)}, r^{(t)}, A^{(t)}, J^{(t)}, \pi^{(t)})$
  $\widetilde{\mathcal{V}}_{\text{edit}}^{(t+1)} \leftarrow \iota_{\text{rpp}}(\mathcal{V}_{\text{edit}}^{(t)}, H_{\text{edit}}^{(t)})$
  $\mathcal{V}_{\text{edit}}^{(t+1)} \leftarrow \kappa_{\text{rpp}}(\widetilde{\mathcal{V}}_{\text{edit}}^{(t+1)})$ ▷ *Commit updates*
  $\mathcal{V}_{\text{evo}}^{(t+1)} \leftarrow \text{ReplaceFields}(\mathcal{V}_{\text{evo}}^{(t)}, \mathcal{V}_{\text{edit}}^{(t+1)})$
  // Phase 3: Re-execute under updated resources
  $Z^{(t+1)} \leftarrow \chi_{\text{rpp}}(\mathcal{A}, \mathcal{V}_{\text{evo}}^{(t+1)}, x, f)$

Algorithm 3 Heuristic reward-informed editing sketch (not standard Reinforce++) (continued)

```
Extract solution y^(t+1) from Z^(t+1)
// Phase 4: Optional solution refinement
if solution refinement is enabled then
    H_sol^(t) ← δ_rpp(Z^(t+1), {y^(t+1)}, r^(t), A^(t), J^(t), π^(t))
    ỹ^(t+1) ← ι_rpp(y^(t+1), H_sol^(t))        ▷ Edit solution text (candidate)
    y^(t+1) ← ỹ^(t+1)                          ▷ Accept task-local edit
end if
V_evo^(t+1) ← SetOutput(V_evo^(t+1), y^(t+1))
// Phase 5: Early stopping
ℓ ← t + 1
if Satisfied(Z^(t+1), y^(t+1)) then
    break
end if
end for
return (y^(ℓ), GetEditable(V_evo^(ℓ)))
```

G.3.7 Group-normalized Reward Editing Sketch (GRPO-inspired)

Scope and supervision. This is a heuristic using group-normalized reward summaries to condition an editor, not standard GRPO optimization (Shao et al., 2024). It shares the limitations of Appendix G.3.6: labeled feedback, similarity rather than likelihood ratios, no defined policy-gradient update, and no separate post-edit gate. It does not produce the main reflection results.

Evolvable Variables. The group-based sketch proposes edits to selected prompt and tool resources and, optionally, the current solution. It compares labeled rewards across sampled candidates to provide diagnostic context for the editor, without differentiating through a language-model policy.

Operator interfaces. The operators below sample candidate solutions, normalize their labeled rewards, and pass the summaries to an editor. Their signatures describe how the heuristic interacts with the resource interface; they do not define a GRPO policy-gradient objective.

For the update loop, $K \geq 2$ is an integer; the sampling interface also permits one rollout for initialization and re-execution. We use the population moments $\bar{r} = K^{-1}\sum_i r_i$ and $\sigma_r^2 = K^{-1}\sum_i (r_i - \bar{r})^2$. If every reward is equal, the advantages are zero rather than undefined. Appendix G.3.6 defines the shared similarity parameters and projection types, not experimental settings.

- **Sample (χ_{grpo}).** $\chi_{\text{grpo}} : (A, \mathcal{V}_{\text{evo}}, x, f, K) \mapsto \{Z_i\}_{i=1}^K$ samples K independent rollouts under the current resources, yielding K execution traces each containing a candidate solution y_i.
- **Reward ($\varepsilon_{\text{grpo}}$).** $\varepsilon_{\text{grpo}} : (\{y_i\}_{i=1}^K, y^*, y^{(t-1)}) \to (\{r_i\}_{i=1}^K, \{A_i\}_{i=1}^K, \{J_i\}_{i=1}^K, \{\pi_i\}_{i=1}^K)$ computes reward-derived diagnostic signals for all K candidates. For each candidate y_i, we compute a task reward r_i comparing y_i with y^*, a heuristic similarity score $\pi_i \triangleq \eta(y^{(t-1)}, y_i)$ approximated via text similarity $\eta(\cdot,\cdot)$ (the sketch does not use token-level probabilities), and a group normalized advantage A_i by normalizing rewards across the candidate set: $A_i = (r_i - \bar{r})/\max(\sigma_r, \epsilon_0)$ where $\bar{r}$ and σ_r are the mean and standard deviation of $\{r_i\}_{i=1}^K$, and $\epsilon_0 > 0$ avoids division by zero. The heuristic clipped score for each candidate is

$$J_i \triangleq \min\big(\pi_i A_i,\ \bar{\pi}_i A_i\big), \quad \bar{\pi}_i \triangleq \begin{cases} \min(\pi_i, 1+\epsilon) & \text{if } A_i \geq 0 \\ \max(\pi_i, 1-\epsilon) & \text{if } A_i < 0 \end{cases}.$$

- **Diagnose (δ_{grpo}).** $\delta_{\text{grpo}} : (\{Z_i\}_{i=1}^K, \mathcal{V}_{\text{edit}}, \{r_i, A_i, J_i, \pi_i\}_{i=1}^K) \to \mathcal{H}$ produces an edit-oriented diagnosis that is explicitly conditioned on the multiple candidate solutions and their reward-derived metrics, enabling the optimizer to identify patterns across candidates.
- **Improve (ι_{grpo}).** $\iota_{\text{grpo}} : \mathfrak{U} \times \mathcal{H} \to \mathfrak{U}$ edits the selected resource projection or, when enabled, the task-local solution, preserving the projection type and all unselected fields.
- **Commit (κ_{grpo}).** $\kappa_{\text{grpo}} : \mathfrak{V}_{\text{edit}} \to \mathfrak{V}_{\text{edit}}$ synchronizes proposed persistent edits with RSPL without a separate post-edit acceptance test. Task-local solution edits use SetOutput instead.

The Evolutionary Loop. Algorithm 4 samples K candidate solutions, constructs group-normalized reward summaries, and conditions resource edits on their traces. It then re-executes the updated resources, optionally revises the solution, and checks the stopping criterion. The group statistics guide the heuristic editor rather than a gradient update. They remain attached to the sampled pre-edit candidates, not to the later solution. As in Appendix G.3.6, solution edits update task-local y without altering prior observations, and stopping must assess the current solution.

Algorithm 4 Heuristic group-reward editing sketch (not standard GRPO)

Input: Agentic system $\mathcal{A}$, task x, optional attachments f, ground truth y^*, update budget $T \geq 0$, candidates per group $K \geq 2$
Output: Final solution $y^{(\ell)}$ and updated editable resources $\mathcal{V}_{\text{edit}}$

// Initialization
$\mathcal{V}_{\text{evo}}^{(0)} \leftarrow \text{VariableLifting}(\mathcal{A})$ ▷ *Expose editable resources*
$\{Z^{(0)}\} \leftarrow \chi_{\text{grpo}}(\mathcal{A}, \mathcal{V}_{\text{evo}}^{(0)}, x, f, 1)$ ▷ *Sample initial solution*
Extract solution $y^{(0)}$ from $Z^{(0)}$
$\mathcal{V}_{\text{evo}}^{(0)} \leftarrow \text{SetOutput}(\mathcal{V}_{\text{evo}}^{(0)}, y^{(0)})$
$\ell \leftarrow 0$
$y^{(-1)} \leftarrow y^{(0)}$ ▷ *Initialize previous solution*
for $t = 0, 1, \ldots, T-1$ **do**
  // Phase 1: Sample multiple candidates
  $\{Z_i^{(t)}\}_{i=1}^K \leftarrow \chi_{\text{grpo}}(\mathcal{A}, \mathcal{V}_{\text{evo}}^{(t)}, x, f, K)$ ▷ *Sample K rollouts*
  Extract candidate solutions $\{y_i^{(t)}\}_{i=1}^K$ from $\{Z_i^{(t)}\}_{i=1}^K$
  // Phase 2: Group rewards and heuristic scores
  $(\{r_i^{(t)}\}_{i=1}^K, \{A_i^{(t)}\}_{i=1}^K, \{J_i^{(t)}\}_{i=1}^K, \{\pi_i^{(t)}\}_{i=1}^K) \leftarrow \varepsilon_{\text{grpo}}(\{y_i^{(t)}\}_{i=1}^K, y^*, y^{(t-1)})$
  // Phase 3: Improve editable resources (prompt and tool)
  $\mathcal{V}_{\text{edit}}^{(t)} \leftarrow \text{GetEditable}(\mathcal{V}_{\text{evo}}^{(t)})$
  $H_{\text{edit}}^{(t)} \leftarrow \delta_{\text{grpo}}(\{Z_i^{(t)}\}_{i=1}^K, \mathcal{V}_{\text{edit}}^{(t)}, \{r_i^{(t)}, A_i^{(t)}, J_i^{(t)}, \pi_i^{(t)}\}_{i=1}^K)$
  $\widetilde{\mathcal{V}}_{\text{edit}}^{(t+1)} \leftarrow \iota_{\text{grpo}}(\mathcal{V}_{\text{edit}}^{(t)}, H_{\text{edit}}^{(t)})$
  $\mathcal{V}_{\text{edit}}^{(t+1)} \leftarrow \kappa_{\text{grpo}}(\widetilde{\mathcal{V}}_{\text{edit}}^{(t+1)})$ ▷ *Commit updates*
  $\mathcal{V}_{\text{evo}}^{(t+1)} \leftarrow \text{ReplaceFields}(\mathcal{V}_{\text{evo}}^{(t)}, \mathcal{V}_{\text{edit}}^{(t+1)})$
  // Phase 4: Re-execute under updated resources
  $\{Z^{(t+1)}\} \leftarrow \chi_{\text{grpo}}(\mathcal{A}, \mathcal{V}_{\text{evo}}^{(t+1)}, x, f, 1)$
  Extract solution $y^{(t+1)}$ from $Z^{(t+1)}$
  // Phase 5: Optional solution refinement
  if solution refinement is enabled **then**
    $H_{\text{sol}}^{(t)} \leftarrow \delta_{\text{grpo}}(\{Z_i^{(t)}\}_{i=1}^K, \{y^{(t+1)}\}, \{r_i^{(t)}, A_i^{(t)}, J_i^{(t)}, \pi_i^{(t)}\}_{i=1}^K)$
    $\widetilde{y}^{(t+1)} \leftarrow \iota_{\text{grpo}}(y^{(t+1)}, H_{\text{sol}}^{(t)})$ ▷ *Edit solution text (candidate)*
    $y^{(t+1)} \leftarrow \widetilde{y}^{(t+1)}$ ▷ *Accept task-local edit*
  end if
  $\mathcal{V}_{\text{evo}}^{(t+1)} \leftarrow \text{SetOutput}(\mathcal{V}_{\text{evo}}^{(t+1)}, y^{(t+1)})$
  // Phase 6: Early stopping
  $\ell \leftarrow t + 1$
  if Satisfied$(Z^{(t+1)}, y^{(t+1)})$ **then**
    break
  end if
end for
return $(y^{(\ell)}, \text{GetEditable}(\mathcal{V}_{\text{evo}}^{(\ell)}))$