SURGICAL META-OPTIMIZATION FOR AGENT EVOLUTION WITH MINIMAL SAMPLES

Anonymous authors
Paper under double-blind review

ABSTRACT

The transition of Large Language Models (LLMs) into autonomous agents requires ways to improve an agent after it is built, without retraining the backbone. However, prevailing paradigms are largely trapped in a brute-force cycle, relying on massive computational resources, extensive annotated data, and unbounded search spaces. This resource-heavy approach hits a strict "resource wall" in practical, locally deployed scenarios. To address the challenge of extreme compute and data starvation, we propose a frugal offline agent-evolution framework driven by *structure-aware surgical intervention*. Rather than pursuing computationally prohibitive full-scale global optimization, we restrict the search to two components of the agent—its system prompt and its controller logic—that can be edited and verified without touching model weights or tool APIs. This is achieved through a two-stage meta-optimization in which each stage optimizes one component independently while holding the other fixed at its baseline. Stage I evolves the system prompt directly, through diversification, recombination, and refinement, to discover a compact and transferable reasoning protocol. Stage II utilizes a contrastive reflection-guided process to independently harden the programmatic execution loop against structural brittleness. Remarkably, operating under a strict budget of merely around 140 candidate evaluations and 30 sparse evaluation samples, our evolved agent successfully tackles expert-level, unseen tasks. Extensive evaluations on exceptionally challenging benchmarks, including Humanity's Last Exam (HLE), demonstrate that strategic, localized component optimization enables stable generalization without requiring massive data or heavy training dependencies. Code is available at **Supplement**.

1 INTRODUCTION

The transition of Large Language Models (LLMs) Achiam et al. (2023); Yang et al. (2025); Liu et al. (2024) from static knowledge bases to autonomous agents capable of dynamic planning and execution marks a significant step toward more capable and general autonomous systems Gao et al. (2025); Xi et al. (2025). Central to this transition is the capacity for ***automated agent evolution***: the ability of an agent system to be restructured automatically from environmental feedback, without human intervention or retraining of the backbone Wang et al. (2023). However, current agent-improvement paradigms are largely trapped in a ***brute-force*** cycle. Methods relying on large-scale Online Direct Preference Optimization (DPO) Rafailov et al. (2023) or multi-round Reinforcement Learning (RL) Zhang et al. (2025a); Hu et al. (2025) typically necessitate thousands of reasoning trajectories and prohibitive re-training costs Li et al. (2025b). Training-free alternatives that optimize prompts or harness code offline Khattab et al. (2023); Agrawal et al. (2026); Hu et al. (2024); Lee et al. (2026) avoid retraining, but most still report search budgets in the thousands of rollouts (Section 2). In practical scenarios, such as private deployments or edge computing, this resource-heavy approach hits a ***resource wall***, where both compute budgets and high-quality annotated data are severely limited Chen & Li (2024).

This bottleneck is further exacerbated by the lack of attention to "practical deployment constraints" in existing research protocols. Most state-of-the-art methods operate under the assumption of unconstrained token budgets and infinite iterations, which are unrealistic for real-world deployment Xia et al. (2024). Furthermore, under such a strict budget, exhaustively optimizing the entire agent implementation is infeasible: the system spans a large, heterogeneous space of prompts,

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.

controllers, and supporting infrastructure, and spreading a small budget uniformly over it leaves too few evaluations per component to tell good candidates from noise Khattab et al. (2023).

To address these challenges, we propose a new paradigm for automated agent evolution under extreme data and compute starvation. Our framework is built around a fundamental **trade-off**: under a strict evaluation budget, pursuing a theoretical *global optimum* through full-scale optimization is generally impractical in expensive black-box settings and often at odds with the pragmatic role of metaheuristics themselves Talbi (2009); Hutter et al. (2011); Snoek et al. (2012). In this regime, the relevant measure of efficiency is the performance gained per candidate evaluation, which is the quantity we report throughout.

Building on this trade-off, we adopt a strategy of **structure-aware *surgical intervention***: we operate on two components of the agent and leave everything else untouched. Two components are shared by virtually all tool-using agents, and prior work has shown that each can be edited productively without retraining: the *system prompt*, which fixes the reasoning protocol Yao et al. (2022); Khattab et al. (2023); Agrawal et al. (2026), and the *controller logic*, which fixes how the agent searches, verifies, and terminates Wang et al. (2023); Xia et al. (2024); Lee et al. (2026). Rather than spreading a small budget over the entire implementation, we confine the search to these two components, in the spirit of hyper-heuristic search that evolves one component of an otherwise fixed algorithm Ye et al. (2024). Under a fixed evaluation budget, a search over a small, verifiable space can be sampled densely, whereas a search over the full system cannot; this is the same trade-off that motivates prompt optimization over weight updates Khattab et al. (2023); Agrawal et al. (2026). This choice may forgo optima that lie outside the two components, but it concentrates the available evaluations where every edit can be verified and where, as Section 6.1 shows, each component contributes measurably.

Supporting this strategy is a **two-stage meta-optimization framework** built on top of an open-weight large reasoning model (LRM) agent base. All backbone inference during evolution is served through a pay-per-token API, so the pipeline requires no GPU training infrastructure and its full cost can be accounted for exactly (Section 4). To keep each search small and verifiable, the optimization is explicitly decomposed into two stages, each targeting one component:

❶ **Stage I: Semantic Evolution.** We evolve the system prompt directly. At each generation the meta-optimizer proposes new prompts by diversifying, recombining, and refining the current population under explicit format constraints, and each candidate is scored on a sparse set of merely 10 task instances.

❷ **Stage II: Logic Re-coding.** We introduce a contrastive reflection-guided evolution for the execution code. Rather than employing random mutations, we systematically contrast successful and failed execution trajectories to extract actionable repair hints. These reflections steer the independent reconstruction of the execution logic, thereby hardening the agent against original structural brittleness utilizing merely 20 task instances.

We investigate a central question: can an open-weight 32B agent system, refined under such a budget, tackle expert-level tasks that were never used during evolution? Our results show that the evolved agent generalizes to the **Humanity's Last Exam (HLE)** Phan et al. (2025), a benchmark of extreme multidisciplinary difficulty, matching its RL-trained counterpart without any HLE-specific tuning or parameter update. This transition from sparse local experience to competitive cross-domain performance supports our central claim: meaningful agent evolution does not require massive data or precise global modeling. Instead, the targeted optimization of two verifiable components can yield substantial and transferable gains under a minimal budget.

Contributions. ❶ *Frugal Evolution Setting.* We formulate offline agent evolution under extreme compute-and-data starvation—a frozen backbone, no gradient training, and a search budget of order 10^2 candidate evaluations on a handful of evolution samples—capturing the constraints of locally and privately deployed agents that prevailing training- or search-heavy methods overlook. ❷ *Targeted Two-Component Formulation.* Rather than optimizing the entire agent or updating weights, we restrict evolution to two interface-stable components—the semantic scaffold P and the execution logic L—via an explicit decomposition $\mathcal{C} = \langle \mathcal{C}_{\text{fix}}, P, L \rangle$, a budget-constrained objective, and an explicit rationale for the intervention surface (Section 3.2). ❸ *Two Evolution Mechanisms.* We instantiate this localization with (i) a prompt evolution that searches research protocols through diversification, recombination, and refinement under explicit format constraints, and (ii) a contrastive

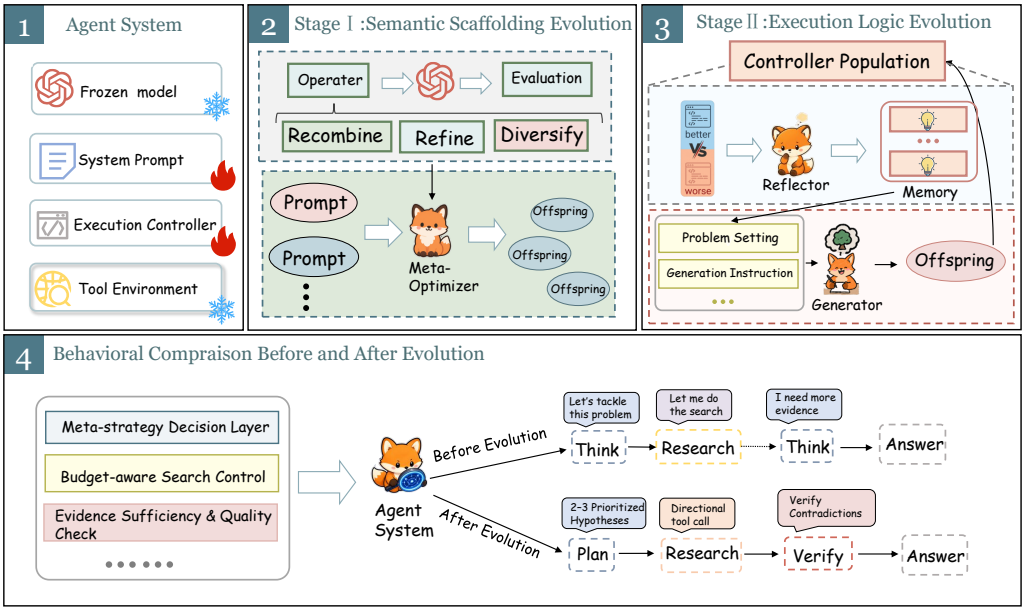


Figure 1: Overview of the dual-track parallel agent-evolution framework. **Stage I** evolves the semantic prompt scaffolding, while **Stage II** independently optimizes the code-level execution logic. The final agent is assembled by superimposing both stages to align reasoning and action.

reflection-guided controller reconstruction that converts good/bad trajectory contrasts into actionable repair hints under a persistent reflection memory. Each track optimizes one component against the fixed baseline of the other. ❹ *Empirical Results under a $10 Budget.* With a total optimization cost of about $10, our evolved 32B agent raises GAIA from 44.7 to 52.4 (best in its parameter class, surpassing the RL-trained WebThinker-32B-RL at 48.5) and reaches 15.6 on Humanity's Last Exam, on par with the RL-trained variant (15.8) and above o3-mini (14.0), generalizing to expert-level unseen tasks without domain-specific tuning. We further provide a unified, itemized cost accounting that substantiates the frugality claim.

2 Related Work

Training-based agent improvement. The dominant way to improve a tool-using agent is to train it. Online Direct Preference Optimization (Rafailov et al., 2023) and multi-round reinforcement learning (Zhang et al., 2025a; Hu et al., 2025) collect reasoning trajectories from the agent, score them, and update the backbone; WebThinker-RL (Li et al., 2025b) follows this recipe with thousands of trajectories and iterative online DPO. These methods raise the ceiling of what the backbone itself can do, but the cost is high—trajectory collection plus GPU training—and the resulting improvement is locked inside the weights, so it can neither be inspected before deployment nor carried over to a different backbone.

Self-evolving agents. A second line leaves the weights fixed and lets the agent rewrite parts of itself from its own experience during deployment. At the *context* layer, Reflexion converts feedback into language-level reflection and plan revision (Shinn et al., 2023), Self-Refine loops generation, self-feedback, and rewriting (Madaan et al., 2023), AdaPlanner replans from execution feedback (Sun et al., 2023), and more recent systems accumulate reusable playbooks or experience banks (Zhang et al., 2026; Huang et al., 2026). At the *tool* layer, Voyager grows a skill library from interaction (Wang et al., 2023), Alita expands the toolset when current tools are insufficient (Qiu et al., 2025), CREATOR separates abstract tool creation from concrete invocation (Qian et al., 2023), and ToolGen unifies retrieval and parameterized invocation (Wang et al., 2024). At the *source* layer, DGM performs open-ended code self-modification over a version archive (Zhang et al., 2025b), MOSS rewrites the agent substrate at source level (Cai et al., 2026), and GENOME and SPIN apply evolutionary or self-play updates (Zhang et al., 2025d; Chen et al., 2024). Because these methods improve *during*

deployment and are open-ended by design, the agent that ships is not the agent that will be running later, and their gains depend on the quality of self-generated reflections.

Offline optimization of agent components. Closest to our setting are methods that optimize an existing agent *once, offline*, and hand back a fixed configuration. At the *prompt* layer, DSPy searches over both instructions and few-shot demonstrations for each module (Khattab et al., 2023), GEPA evolves prompts by reflecting on rollouts in natural language (Agrawal et al., 2026), and SPEAR gives the optimizer a code sandbox so that it can author its own error analysis (Lu et al., 2026). At the *workflow* layer, ADAS has a meta agent write complete agent code, prompts included (Hu et al., 2024), and AFlow searches code-represented workflows with Monte Carlo tree search (Zhang et al., 2025c). At the *harness* layer, Meta-Harness searches harness code with per-iteration feedback contexts of up to millions of tokens (Lee et al., 2026), and Life-Harness adapts the runtime interface of a frozen agent (Xu et al., 2026). At the *configuration* layer, Artemis evolves prompts, model choice, and hyper-parameters through black-box interaction alone (Brookes et al., 2025). Prior work shows that prompts and harness code are useful optimization targets, but most tune one surface, regenerate agent code wholesale, or treat agents as black boxes; most report thousands of rollouts. To our knowledge, none separately evolves an existing agent's prompt and controller logic, verifying each against its own interface.

3 METHOD

3.1 PRELIMINARIES AND BASELINE AGENT

Baseline agent. Our work builds on the open-source *WebThinker* deep-research agent Li et al. (2025b), which equips a frozen large reasoning model with (i) a *Deep Web Explorer* for search and page navigation, and (ii) an *Autonomous Think, Search and Draft* workflow that interleaves reasoning with information seeking and report drafting, checking, editing.

Agent decomposition. We denote the frozen reasoning backbone as M with fixed parameters θ (QwQ-32B Team (2025) in our experiments), and the agent-side system implementation as $\mathcal{C}$ (the WebThinker codebase). To make the optimization scope explicit, we decompose:

$$\mathcal{C} = \langle \mathcal{C}_{\text{fix}}, P, L \rangle,$$

where $\mathcal{C}_{\text{fix}}$ contains the fixed infrastructure (tool wrappers, IO formatting, logging, etc.), P is the system prompt (semantic scaffold), and L is the execution controller (control flow and scheduling logic).

Given a user query Q and an external tool environment $\mathcal{E}$ (search and page-reading APIs), one execution produces an answer and an execution trace:

$$(y, \pi) = \text{Run}(M, \mathcal{C}_{\text{fix}}, P, L, Q; \mathcal{E}),$$

where π records tool calls, intermediate states, and termination decisions.

Evaluation. For a set of tasks $\mathcal{D}$, we measure task success by accuracy under the benchmark's official grading procedure:

$$\text{Acc}(P, L; \mathcal{D}) \triangleq \frac{1}{|\mathcal{D}|} \sum_{Q \in \mathcal{D}} \mathbb{1}\{\text{grade}(y(Q), Q) = 1\}.$$

3.2 FRUGAL MODULAR EVOLUTION: OBJECTIVE AND CONSTRAINTS

Choosing what to evolve. In a tool-using agent, the decisions that shape a trajectory are carried by two components: the system prompt, which fixes what the model is asked to do at each step, and the controller, which fixes what the system does with the model's output—whether to search again, what evidence to keep, when to stop. The rest of the implementation is plumbing: tool wrappers, IO formatting, logging. It determines how the agent runs, not what it decides. We therefore evolve P and L, and hold $\mathcal{C}_{\text{fix}}$, the backbone, and the tool APIs fixed.

Two properties make this pair workable under a small budget. Both are text that an LLM can rewrite directly, so no gradient is needed. And both can be checked *before* they are run—a prompt against

tool-call format and behavioral constraints, a controller by syntax and interface checking, sandboxed execution, and unit tests—so invalid candidates are discarded at negligible cost and the budget is spent only on candidates that execute.

External meta-optimizer. We use an external LLM meta-optimizer Φ (gpt-5-nano for Stage I and gpt-5-mini for Stage II in our experiments) to propose edits to P and L. Unlike self-evolving agents, which improve online from their own trajectories during deployment Shinn et al. (2023); Zhang et al. (2025b), our framework runs offline and once, uses an external meta-optimizer, and outputs a fixed agent configuration; the term *evolution* refers to the population-based search over agent components, not to autonomous self-modification.

Optimization problem under a strict budget. Let $\mathcal{D}_{\text{evo}}$ be a small evolution set, disjoint from the evaluation benchmarks (Section 5.1). We aim to maximize accuracy on $\mathcal{D}_{\text{evo}}$ subject to (i) a strict meta-optimization budget (number of candidate evaluations) and (ii) interface constraints:

$$\max_{P\in\mathcal{P},\, L\in\mathcal{L}} \ \mathrm{Acc}(P, L; \mathcal{D}_{\text{evo}}) \quad \text{s.t.} \quad \Omega(P, L) = 1, \ \ \mathrm{Budget}(P, L) \le B.$$

Ω enforces format/interface validity (e.g., tool-call syntax, controller function signature, and invariants required by WebThinker), and B limits the total number of candidate evaluations performed during evolution.

Two decoupled tracks. We evolve P and L in two decoupled tracks: (i) evolve P with the baseline controller L_0 fixed; (ii) evolve L with the baseline prompt P_0 fixed. Each track optimizes one component against the fixed baseline of the other, keeping both searches small and independent.

3.3 Stage I: Semantic Scaffolding Evolution

Search operators. Optimizing the system prompt is a search over high-level research protocols. At each generation, the meta-optimizer Φ proposes new candidate prompts from the current population using three operator types: (i) *Diversification* proposes a prompt whose research protocol differs substantially from existing ones, to escape local optima; (ii) *Recombination* fuses reusable instructions from multiple strong prompts into a composite protocol; (iii) *Local refinement* edits a strong prompt to remove ambiguity, eliminate redundancy, and strengthen constraints.

Constraint enforcement. Every proposed prompt P must satisfy Ω, which includes tool-call format requirements and mandatory behavioral constraints (e.g., evidence-first answering, citation discipline if applicable). Candidates that violate Ω are rejected before any evaluation budget is spent on them.

Selection. Each candidate prompt P is evaluated by running the baseline controller L_0 on a small prompt-evolution set $\mathcal{D}_P$, scored by $\mathrm{Acc}(P, L_0; \mathcal{D}_P)$. We apply elitist selection to retain the best prompts under the strict evaluation budget, and return the best prompt P^*.

3.4 Stage II: Execution Logic Evolution via Reflective Controller Reconstruction

Targeted controller scope. We only evolve a small set of modules in the WebThinker controller L (core loop, termination criteria, scheduling rules) while keeping the external interface unchanged. This yields a constrained and verifiable search space: each candidate is a patch or code snippet that can be executed within the same agent framework.

Contrastive diagnosis and text-guided repair. Let L^+ denote the controller with better performance and L^- the one with worse performance. We compute the performance gap on the evolution set $\mathcal{D}_L$:

$$\Delta = \mathrm{Acc}(P_0, L^+; \mathcal{D}_L) - \mathrm{Acc}(P_0, L^-; \mathcal{D}_L)$$

Guided by this gap, the model Φ generates a textual repair hint g_t:

$$g_t \leftarrow \Phi(L^+, L^-, \Delta)$$

g_t identifies the causes of the performance difference and concrete improvements in structured text (e.g., deduplication logic, evidence filtering, early stopping, query reformulation triggers) to guide targeted code rewriting.

Reflection memory mechanism. We maintain a persistent reflection memory $\mathcal{R}$ initialized as empty. In each iteration, the newly generated valid repair hint g_t is incorporated into the memory:

$$\mathcal{R} \leftarrow \mathcal{R} \cup \{g_t\}$$

The model Φ generates hints conditioned on $\mathcal{R}$ to avoid repeating past failure modes and continuously accumulate heuristic rules.

Validity checking and elitist search process. All candidate controllers $\tilde{L}$ must satisfy constraints Ω (interface compatibility, syntax/type checking, sandbox execution, unit tests, etc.). Among the valid candidate set $\mathcal{L}_{\text{valid}}$, we select the controller with the highest accuracy on $\mathcal{D}_L$:

$$L^* \leftarrow \arg\max_{L \in \mathcal{L}_{\text{valid}}} \text{Acc}(P_0, L; \mathcal{D}_L)$$

We search for the best-performing controller and finally output the optimal controller L^*.

4 Frugality and Resource Efficiency Analysis

The transition toward autonomous agents often incurs prohibitive computational costs. Section 2 surveys the budgets these methods report: parameter updates require thousands of trajectories plus GPU training, and the search-based methods that leave the weights untouched still consume thousands of rollouts or per-iteration feedback contexts of millions of tokens.

Using 30 evolution samples, our framework requires no parameter updates or training infrastructure and costs about $10 in API fees per full run. Table 1 itemizes our exact two-stage costs: candidate evaluations, meta-optimizer and agent-inference calls, external tool calls, token consumption, and spending. This places our optimization at the low end of search-based methods.

Table 1: Cost breakdown of the two-stage evolution run. Token prices are input/output USD per million tokens.

Dimension	Stage I (Semantic)	Stage II (Controller)	Total
Evolution Configuration			
Evolution samples	10	20	30
Candidate evaluations	77	60	137
Meta-Optimizer (Code Generation)			
Model	gpt-5-nano	gpt-5-mini	—
API calls	81	88	169
Input tokens	∼1.6M	∼1.8M	∼3.4M
Output tokens	0.43M	0.59M	1.02M
Unit price ($/M tok)	0.04 / 0.35	0.22 / 1.76	—
Meta-optimizer cost	∼$0.2	∼$1.4	∼**$1.6**
Agent Inference (Evaluation)			
Model	QwQ-32B (API)	QwQ-32B (API)	—
Total evaluation runs	770	1,200	1,970
Inference tokens	∼3M	∼4.8M	∼7.8M
Unit price ($/M tok)	0.22 / 0.56	0.22 / 0.56	—
Agent-inference cost	∼$1.4	∼$2.2	∼**$3.6**
External Tool Calls			
Search + scraping cost (Serper, Jina)	∼$2	∼$3	∼**$5**
Total API cost			∼**$10.2**

5 Experiments

5.1 Datasets and Benchmarks

We evaluate cross-domain generalization and research depth on two complementary benchmarks.

❶ ***GAIA Mialon et al. (2023).*** Our primary benchmark tests multi-step reasoning and tool use through tiered tasks resembling professional office work.

❷ ***HLE Phan et al. (2025).*** Humanity's Last Exam tests expert-level knowledge discovery across disciplines, requiring graduate-level expertise and advanced search and reasoning. Most systems in Table 3 score below 15%, making HLE a rigorous test of generalized agent intelligence.

Evaluation protocol. For both benchmarks we follow the evaluation setup of WebThinker (Li et al., 2025b): the same subsets, the same number of questions, and the same LLM-as-judge grading procedure. Our numbers are therefore directly comparable with the baseline results reported in that work.

Evolution set. The 30 questions used during evolution (10 for Stage I, 20 for Stage II) are drawn from sources disjoint from both evaluation benchmarks. No GAIA or HLE question is used at any point during evolution, so both benchmarks are held out in full.

5.2 Baselines

We compare our proposed framework against the following three categories of methodologies:

❶ ***Direct Reasoning.*** Reasoning models that rely exclusively on their internalized knowledge without access to external search capabilities. This category includes open-weight models such as Qwen2.5-32B/72B-Instruct, DeepSeek-R1-32B, QwQ-32B Team (2025), and the large-scale DeepSeek-R1-671B Liu et al. (2024), as well as proprietary models including GPT-4o Hurst et al. (2024), o3-mini, and Gemini-2.0-Flash-Thinking.

❷ ***Retrieval-Augmented Reasoning.*** Methods that leverage search engines to acquire external knowledge. These specifically include: (i) **Standard RAG** (retrieval based on the original query); (ii) **RAG with Query Planning** (decomposing the query into sub-queries for targeted retrieval before final generation); and (iii) **Iterative RAG** (performing sequential information retrieval throughout the reasoning process).

❸ ***Autonomous Search within Reasoning.*** Systems that natively integrate search into reasoning include open-source Search-o1 Li et al. (2025a) and WebThinker Li et al. (2025b) (training-free WebThinker-32B and RL-trained WebThinker-32B-RL), as well as proprietary OpenAI Deep Research OpenAI (2025).

5.3 Implementation Details

This study adopts **QwQ-32B** Team (2025) as the core reasoning backbone of the agent. All backbone inference, both during evolution and for the final benchmark evaluation, is served through a pay-per-token API rather than local GPUs; the resulting cost is itemized in Section 4. For external knowledge acquisition, we utilize the **Serper API** for high-concurrency web searches and the **Jina Reader API** for content retrieval, ensuring efficient context ingestion. To accommodate deep reasoning traces and extensive evidence, the maximum generation limit is set to 32768 tokens. The optimization process is conducted under strict operational constraints to validate our frugal paradigm: ❶ ***Stage I (Semantic Evolution).*** Under a limited token budget, we optimize high-level heuristic instructions governing the agent's research behavior and planning logic, evaluating **77 candidate prompts** over **5 generations** to balance diversity and exploration efficiency. ❷ ***Stage II (Logic Re-coding).*** This stage reconstructs the execution logic's core control functions through reflective sampling, efficiently extracting actionable repair hints from contrastive environmental feedback to propose logic refinements. We evaluate **60 candidate controllers** over **16 generations**. For all baselines, we report results from the original papers or evaluate their official implementations.

5.4 Benchmark Results

GAIA Benchmark. Table 2 compares our method with direct reasoning, retrieval-augmented, and agentic search baselines. Without updating model parameters, our method achieves the highest overall score among 32B systems (52.4), exceeding WebThinker-32B (44.7) and WebThinker-32B-RL (48.5) by 7.7 and 3.9 points, respectively. The latter was trained with over 7,800 trajectories and iterative online DPO. Relative to WebThinker-32B, our method improves on all three difficulty levels, with

the largest gain on Level 3 (16.7 to 41.7). WebThinker-32B-RL remains stronger on Level 2 (50.0 vs. 46.2).

Table 2: Main results on the GAIA benchmark. We highlight the QwQ-based reasoning baselines and our evolved framework. Red frames denote the best results in the 32B-parameter class. †: proprietary system. ‡: trained with 7.8k+ trajectories and iterative online DPO.

Method	GAIA (General AI Assistant) Score ↑			
	Level 1	Level 2	Level 3	**Avg.**
Direct Reasoning (Zero-shot / w/o Retrieval)				
Qwen2.5-32B	20.5	9.6	8.3	13.6
DeepSeek-R1-32B	23.1	17.3	0.0	17.5
QwQ-32B	30.8	15.4	25.0	22.3
Reference Models (Vastly larger parameters or Closed-source)				
Qwen2.5-72B	20.5	13.5	0.0	14.6
GPT-4o	23.1	15.4	8.3	17.5
DeepSeek-R1-671B	43.6	26.9	8.3	31.1
Enhancing Reasoning with RAG Workflow				
RAG-Qwen2.5-32B	12.8	11.8	8.3	11.8
w/ Query Planning	30.8	17.3	0.0	20.4
w/ Iterative RAG	35.9	19.2	8.3	24.3
RAG-QwQ-32B	33.3	36.5	8.3	32.0
w/ Query Planning	48.7	25.0	8.3	32.0
w/ Iterative RAG	51.3	28.8	8.3	35.0
Autonomous Search within Reasoning (Agentic Evolution)				
OpenAI Deep Research†	74.3	69.1	47.6	67.4
Search-o1-32B	53.8	34.6	16.7	39.8
WebThinker-32B	53.8	44.2	16.7	44.7
WebThinker-32B-RL‡	56.4	**50.0**	16.7	48.5
Ours (Evolved-32B)	**64.1**	46.2	**41.7**	**52.4**

HLE Benchmark. We evaluate cross-domain transfer by applying the GAIA-optimized components to HLE without further tuning. Our agent scores **15.6**, exceeding WebThinker-32B (13.0) by **2.6** points and o3-mini (14.0), while approaching WebThinker-32B-RL (15.8). It improves over WebThinker-32B in Math, Bio/Medicine, Physics, CS, and Humanities, but performs worse in Chemistry, Engineering, and Other. These results suggest that the evolved components transfer beyond GAIA. The optimization costs roughly $10 and requires no gradient updates, compared with thousands of training trajectories for WebThinker-32B-RL.

6 Further Analysis

6.1 Ablation Study

To evaluate our two-stage framework, we conducted an ablation study on the GAIA benchmark (Figure 2a). Compared to the baseline (44.7%), applying only Stage I (semantic evolution) or Stage II (logic re-coding) improves accuracy to 47.5% and 48.7%, respectively. The full framework achieves the best performance at 52.4%. These results show that each component contributes on its own (+2.8 and +4.0 points) and that the gains are largely additive when combined (+7.7 points), which supports evolving the two components in independent tracks.

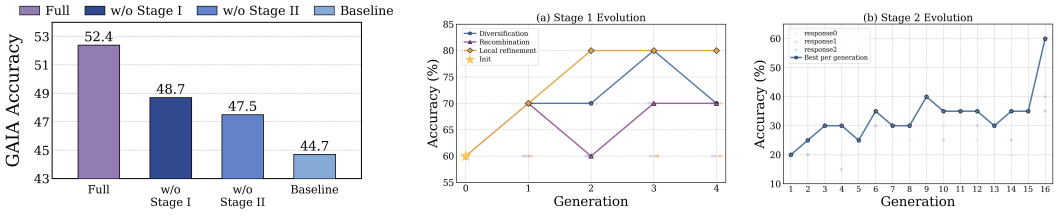
6.2 Evolution Trajectory Analysis

Figures 2b and 2c show the best candidate per generation on the evolution sets; note that these are accuracies on $\mathcal{D}_P$ (10 items) and $\mathcal{D}_L$ (20 items), not on the benchmarks. Stage I (semantic evolution) starts from the baseline prompt at 60% and reaches 80% by generation 2: prompt edits deliver

Table 3: Humanity's Last Exam (HLE) Pass@1 (%) by specialized reasoning task. Red frames mark the best 32B-model results, including ties. † Reported scores for closed-source or large-scale models. ‡ Trained with 7.8k+ trajectories and iterative online DPO;

Method	Humanity's Last Exam (Extremely Hard Reasoning Tasks)								
	Math	B/M	Phys	CS	Hum	Chem	Eng	Oth	Avg.
Direct Reasoning (Zero-shot / w/o Retrieval)									
Qwen2.5-32B	6.0	7.0	2.0	3.2	10.0	0.0	5.3	4.4	5.4
DeepSeek-R1-32B	6.9	8.3	3.5	4.5	7.3	7.4	7.0	5.7	6.4
QwQ-32B	12.6	14.0	4.0	7.9	6.0	13.3	5.3	4.4	9.6
Reference Models (Closed-source or Large-scale)									
GPT-4o†	2.4	5.3	2.2	1.2	2.9	1.9	1.3	3.5	2.6
Gemini-2.0-F-T†	8.5	7.4	5.3	5.8	7.1	6.5	3.8	4.0	7.1
DeepSeek-R1-671B†	9.3	8.6	5.8	7.4	11.0	5.6	10.3	7.5	8.6
o3-mini (High)†	18.8	11.1	14.2	11.1	6.2	10.2	7.7	8.0	14.0
Enhancing Reasoning with RAG Workflow									
RAG-Qwen2.5-32B	4.2	7.0	8.2	1.6	8.0	13.3	5.3	0.0	4.8
RAG-QwQ-32B	7.9	14.0	2.0	4.8	14.0	0.0	0.0	4.4	7.2
w/ Query Planning	11.2	16.3	4.0	4.8	12.0	6.7	0.0	11.1	9.6
w/ Iterative RAG	10.2	14.0	4.0	7.9	10.0	13.3	10.5	8.9	9.6
Autonomous Search within Reasoning (Agentic Evolution)									
OpenAI Deep Research†	–	–	–	–	–	–	–	–	26.6
Search-o1-32B	12.1	11.6	2.0	7.9	14.0	6.7	10.5	**15.6**	10.8
WebThinker-32B	14.9	16.3	6.0	9.5	6.0	20.0	**21.1**	**15.6**	13.0
WebThinker-32B-RL‡	**16.7**	**25.6**	2.0	12.7	18.0	**26.7**	15.8	**15.6**	**15.8**
Ours (Evolved-32B)	15.8	18.6	**10.0**	**15.9**	**20.0**	13.3	15.8	13.3	15.6

their gain almost immediately and then saturate. Stage II (logic re-coding) behaves differently. The best controller rises from 20% to 40% by generation 9, fluctuates between 30% and 35% through generation 15, and reaches 60% in the final generation. Because both evolution sets are small, one item corresponds to 10 and 5 points respectively, so we read these curves as descriptions of the search dynamics rather than as estimates of test performance; the held-out results are those in Tables 2 and 3.



(a) Ablation study (b) Stage I evolution (c) Stage II evolution

Figure 2: Ablation study and evolution processes across different stages.

7 Conclusion

We present a frugal approach to agent evolution, in contrast to methods that rely on thousands of optimization trajectories or extensive simulation memories. By optimizing only two verifiable components, the system prompt and controller logic, our framework achieves substantial gains with 30 sparse samples and about 140 candidate evaluations. These results suggest that meaningful agent improvement need not require massive data or compute, offering a practical path for resource-constrained deployment.

AI USE STATEMENT

LLMs are components of the proposed framework: QwQ-32B serves as the agent backbone, while GPT-5-nano and GPT-5-mini generate and refine candidate prompts and controller logic in Stages I and II, respectively. Generative AI tools were also used to improve the clarity and concision of the manuscript. The authors reviewed the text and code, verified the reported results and technical claims.

REFERENCES

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

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. arXiv:2507.19457.

Paul Brookes, Vardan Voskanyan, Rafail Giavrimis, Matthew Truscott, Mina Ilieva, Chrystalla Pavlou, Alexandru Staicu, Manal Adham, Will Evers-Hood, Jingzhi Gong, Kejia Zhang, Matvey Fedoseev, Vishal Sharma, Roman Bauer, Zheng Wang, Hema Nair, Wei Jie, Tianhua Xu, Aurora Constantin, Carmine Ventre, Leslie Kanthan, and Michail Basios. Evolving excellence: Automated optimization of LLM-based agents. *arXiv preprint arXiv:2512.09108*, 2025.

Qianshu Cai, Yonggang Zhang, Xianzhang Jia, Huajiang Zheng, Wei Xue, Jun Song, Xinmei Tian, and Yike Guo. MOSS: Self-evolution through source-level rewriting in autonomous agent systems. *arXiv preprint arXiv:2605.22794*, 2026.

Wei Chen and Zhiyuan Li. Octopus v2: On-device language model for super agent. *arXiv preprint arXiv:2404.01744*, 2024.

Zixiang Chen, Yihe Deng, Huizhuo Yuan, Kaixuan Ji, and Quanquan Gu. Self-play fine-tuning converts weak language models to strong language models. *arXiv preprint arXiv:2401.01335*, 2024.

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

Hanjiang Hu, Changliu Liu, Na Li, and Yebin Wang. Training task reasoning llm agents for multi-turn task planning via single-turn reinforcement learning. *IEEE Control Systems Letters*, 9:2879–2884, 2025.

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

Yue Huang, Wenjie Wang, Han Bao, Yuchen Ma, Xiaonan Luo, Yi Nian, Haomin Zhuang, Zheyuan Liu, Yue Zhao, and Xiangliang Zhang. MemoHarness: Agent harnesses that learn from experience. *arXiv preprint arXiv:2607.14159*, 2026.

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*, 2024.

Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *International conference on learning and intelligent optimization*, pp. 507–523. Springer, 2011.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Saiful Haq, Ashutosh Sharma, Thomas T Joshi, Hanna Moazam, Heather Miller, et al. Dspy: compiling declarative language model calls into state-of-the-art pipelines. In *The Twelfth International Conference on Learning Representations*, 2023.

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

Xiaoxi Li, Guanting Dong, Jiajie Jin, Yuyao Zhang, Yujia Zhou, Yutao Zhu, Peitian Zhang, and Zhicheng Dou. Search-o1: Agentic search-enhanced large reasoning models. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 5420–5438, 2025a.

Xiaoxi Li, Jiajie Jin, Guanting Dong, Hongjin Qian, Yongkang Wu, Ji-Rong Wen, Yutao Zhu, and Zhicheng Dou. Webthinker: Empowering large reasoning models with deep research capability. *arXiv preprint arXiv:2504.21776*, 2025b.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

Mengyin Lu, Cong Feng, Huimin Han, Guangming Lu, Yu Sun, Xiaonan Ding, Shihui Long, Fengyi Li, and Tanvi Motwani. SPEAR: Code-augmented agentic prompt optimization. *arXiv preprint arXiv:2605.26275*, 2026.

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.

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*, 2023.

OpenAI. Deep research system card. Technical Report, 2025. URL https://cdn.openai.com/deep-research-system-card.pdf.

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.

Cheng Qian, Chi Han, Yi Fung, Yujia Qin, Zhiyuan Liu, and Heng Ji. Creator: Tool creation for disentangling abstract and concrete reasoning of large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 6922–6939, 2023.

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.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. *Advances in neural information processing systems*, 36:53728–53741, 2023.

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.

Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. *Advances in neural information processing systems*, 25, 2012.

Haotian Sun, Yuchen Zhuang, Lingkai Kong, Bo Dai, and Chao Zhang. Adaplanner: Adaptive planning from feedback with language models. *Advances in neural information processing systems*, 36:58202–58245, 2023.

El-Ghazali Talbi. *Metaheuristics: from design to implementation*. John Wiley & Sons, 2009.

Qwen Team. Qwq-32b: The power of scaling rl, March 2025. URL https://qwenlm.github.io/blog/qwq-32b/.

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

Renxi Wang, Xudong Han, Lei Ji, Shu Wang, Timothy Baldwin, and Haonan Li. Toolgen: Unified tool retrieval and calling via generation. *arXiv preprint arXiv:2410.03439*, 2024.

Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, et al. The rise and potential of large language model based agents: A survey. *Science China Information Sciences*, 68(2):121101, 2025.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489*, 2024.

Tianshi Xu, Huifeng Wen, and Meng Li. Adapting the interface, not the model: Runtime harness adaptation for deterministic LLM agents. *arXiv preprint arXiv:2605.22166*, 2026.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

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 *The eleventh international conference on learning representations*, 2022.

Haoran Ye, Jiarui Wang, Zhiguang Cao, Federico Berto, Chuanbo Hua, Haeyeon Kim, Jinkyoo Park, and Guojie Song. Reevo: Large language models as hyper-heuristics with reflective evolution. *Advances in neural information processing systems*, 37:43571–43608, 2024.

Hanchen Zhang, Xiao Liu, Bowen Lv, Xueqiao Sun, Bohao Jing, Iat Long Iong, Zhenyu Hou, Zehan Qi, Hanyu Lai, Yifan Xu, et al. Agentrl: Scaling agentic reinforcement learning with a multi-turn, multi-task framework. *arXiv preprint arXiv:2510.04206*, 2025a.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin godel machine: Open-ended evolution of self-improving agents. *arXiv preprint arXiv:2505.22954*, 2025b.

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*, 2025c. arXiv:2410.10762.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and Kunle Olukotun. Agentic context engineering: Evolving contexts for self-improving language models. In *International Conference on Learning Representations*, 2026. arXiv:2510.04618.

Yiqun Zhang, Peng Ye, Xiaocui Yang, Shi Feng, Shufei Zhang, Lei Bai, Wanli Ouyang, and Shuyue Hu. Nature-inspired population-based evolution of large language models. *arXiv preprint arXiv:2503.01155*, 2025d.

Appendix

TABLE OF CONTENTS

A Reasoning Trace Comparison **14**

B Extended Empirical Analysis of the Evolution Framework **16**

B.1 Correlation Analysis: Code/Prompt Complexity versus Agent Performance 16

B.2 Evolutionary Dynamics of Module Complexity 16

B.3 Computational Cost and Runtime Analysis 17

C System Prompts and Implementations **18**

C.1 Original WebThinker Multi-QA Prompt 18

C.2 New Algorithm: Synthesize-Query-Reason-Verify (SQRV) Prompt 19

C.3 Baseline Web Reasoning Pipeline 19

C.4 Evolved V2 Web Reasoning Pipeline (Stage 2) 23

C.5 Meta-Optimizer Prompts for Two-Stage Evolution 31

C.6 Dual-Track Evolution Procedure 32

D Limitations and Broader Impacts **32**

D.1 Limitations 32

D.2 Broader Impacts 33

A Reasoning Trace Comparison

Task Definition

Question: Who did the actor who played Ray in the Polish language version of Everybody Loves Raymond play in Magda M.? Give only the first name.

Baseline Agent Reasoning Trace (Before Evolution)

Okay, let's tackle this question step by step. The user wants to know who the actor who voiced Ray in the Polish-language version of "Everybody Loves Raymond" played in "Magda M.", and they just want the first name.

First, I need to figure out who the Polish voice actor for Ray Barone is in the Polish-dubbed version of the American sitcom "Everybody Loves Raymond." The main actor in the original show is Ray Romano as Ray Barone, but in Poland, they dub the shows, so there's a different actor providing the voice.

I'll start by searching for the Polish voice actor of Ray Barone in "Everybody Loves Raymond."

(The first search returns no relevant information about the Polish dub voice actor.)

Hmm, the first search didn't give me any concrete information. The system suggests checking specific Polish databases. Since I can't access external links directly, maybe I should try another approach.

Let me think again. Maybe I can look for the title of the Polish version of "Everybody Loves Raymond", which might be "Wszyscy kochają Raymonda". If I search for the Polish dub cast of that show, specifically the voice for Ray Barone, that should lead me to the actor's name.

(The second search highlights a Polish adaptation "Wszyscy kochają Romana" where Bartłomiej Kasprzykowski plays Roman, the counterpart of Ray.)

Wait, this is confusing. The question refers to a Polish-language version, which could mean either a dub or a local adaptation. Since I cannot find the dubbing cast, perhaps the question is referring to the adaptation.

Assuming the user meant the adaptation, then the actor is Bartłomiej Kasprzykowski.

Now, the next step is to find out what he played in "Magda M."

(Search results show that Bartłomiej Kasprzykowski played Wojciech Płaska in Magda M.)

Okay, so this seems promising. The actor is Bartłomiej Kasprzykowski, and in *Magda M.* he played Wojciech Płaska.

However, the question might instead be interpreted as asking for the first name of the actor. In that case, the actor's first name is:

Final Answer: Bartłomiej ✗ Incorrect (Hallucinated Intent)

Evolved Agent Reasoning Trace (After Evolution)

Okay, let me try to figure this out. The question is asking about an actor who played Ray in the Polish-language version of "Everybody Loves Raymond" and then wants to know what character he played in "Magda M." The key here is that we're dealing with voice actors for the Polish dub, not the original actors.

First, I need to find out who voiced Ray Barone in the Polish dub of "Everybody Loves Raymond." The original Ray is played by Ray Romano, but since it's the Polish version, it's a

different actor. Let me start by searching for the Polish voice cast of the show. Maybe there's a list somewhere.

So, I'll do a search for the Polish dub of "Everybody Loves Raymond" and look for the voice actor for Ray. Let me try that.

Wait, the user mentioned "Polish-language version," so maybe the term used is "Polish dub" or "Polish voice actors." Also, "Magda M." sounds like a Polish movie or TV series. I might need to check what "Magda M." is as well. Maybe it's a Polish film or another sitcom?

Once I get the name of the actor from the first part, I can then search for his roles in "Magda M." But first, focus on finding the Polish voice actor for Ray.

Hmm, after the first search result comes back, let's see if there's any info. Suppose the search gives me the name of the actor. Let's say it's someone like Piotr Adamczyk or another common Polish voice actor. Wait, actually, I'm not sure. Maybe I should look up specific sources like Polish Wikipedia entries for the show's dub.

Alternatively, maybe there's a website that lists international dubs. For example, sometimes sites like "Dubbing Poland" or "IMDb" have information on voice actors for different countries. Let me try searching for "Polish dub of everybody loves raymond cast."

Wait, I should structure my thoughts properly. Let's follow the algorithm: Plan, Reflect, Query, Validate.

Plan: First, identify the Polish voice actor for Ray Barone in "Everybody Loves Raymond." Second, find out which character he played in "Magda M."

Hypotheses for the first part (Ray in Polish dub): 1. The actor might be Tomasz Kozłowski. 2. It could be Marcin Kosiński. 3. Maybe Arkadiusz Jachym.

Not sure yet. Let me search.

(Search results indicate that the Polish version is titled "Wszyscy kochają Romana" and the actor who plays the equivalent of Ray is Bartłomiej Kasprzykowski.)

Alright, so the first search gave me that the Polish dub/adaptation uses Bartłomiej Kasprzykowski. Now I need to find out what role he played in "Magda M." Since the initial search didn't mention it, I should look specifically into his filmography related to that movie or series.

Next, I should search for Bartłomiej Kasprzykowski's roles in "Magda M." Let me form the query.

(Search results show that Bartłomiej Kasprzykowski played the role of Wojciech Płaska in the Polish TV series Magda M.)

Okay, let me tackle this step by step. The user wants to know the first name of the actor who played Ray in the Polish dub of "Everybody Loves Raymond" and then what character that same actor portrayed in "Magda M."

First, I needed to find the Polish voice actor for Ray Barone. The original show's main actor is Ray Romano, but in the Polish version, the name might be localized. My initial search found that the Polish version is titled "Wszyscy kochają Romana," and the actor is Bartłomiej Kasprzykowski.

Now, moving to the second part: checking Bartłomiej Kasprzykowski's role in "Magda M." The search results show he played Wojciech Płaska.

The question asks for the first name of the character he played in *Magda M.*, so the answer is:

Final Answer: Wojciech ✓ Correct (Accurate Execution)

B Extended Empirical Analysis of the Evolution Framework

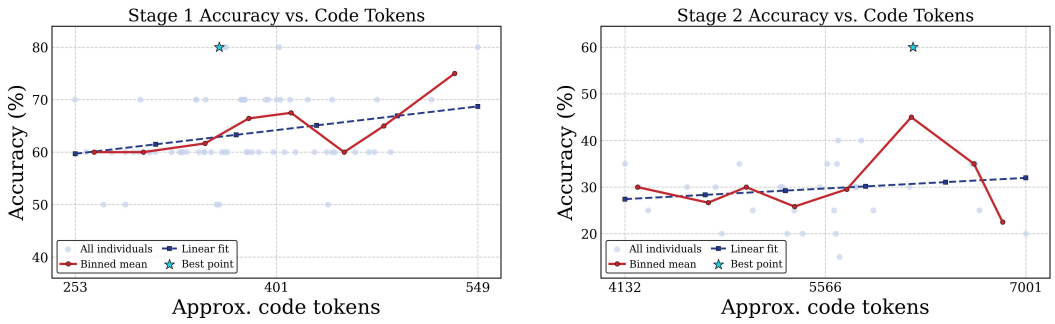
B.1 Correlation Analysis: Code/Prompt Complexity versus Agent Performance

To further understand the dynamics of our evolution framework, we investigate the relationship between the structural complexity of the evolved modules (measured by the approximate number of code or prompt tokens) and their resulting task accuracy. Figure 3 illustrates this correlation for both Stage I (Semantic Evolution) and Stage II (Logic Re-coding).

Stage I: Positive Correlation in Semantic Scaffolding. As shown in Figure 3(a), the token length of the semantic prompt in Stage I ranges roughly from 250 to 550 tokens. The linear fit (blue dashed line) reveals a mild positive correlation between prompt length and the agent's accuracy on $\mathcal{D}_P$ (from about 60% to 69% across the range), and the binned mean (red line) rises toward the upper end of the range. At the same time, the best prompt (star, 80%) sits at roughly 370 tokens, well below the maximum, and prompts of 80% accuracy also appear near 400 and 550 tokens. This indicates that, during the semantic evolution phase, prompts with more explicit constraints and operational guidelines (e.g., specific rules for evidence-first answering or error recovery) tend to perform better on average, but the best-performing prompts are not the longest ones. The entire useful range remains compact (under 600 tokens), consistent with our strategy of extracting a concise, high-density reasoning protocol rather than an ever-longer instruction list.

Stage II: The Danger of Code Bloat in Execution Logic. In contrast, Figure 3(b) presents a different pattern for the execution controller logic evolved in Stage II. Here, the codebase is significantly larger, spanning from about 4,100 to 7,000 tokens. The linear fit (blue dashed line) is nearly flat (about 27% to 32%), and the binned mean (red line) is non-monotonic: it stays around 25–30% for compact controllers below roughly 5,800 tokens, peaks at about 45% near 6,200 tokens—where the best controller (star, 60% on $\mathcal{D}_L$) is located—and then falls to 35% and 22% for the longest controllers (6,700–7,000 tokens).

Two observations follow. First, the best controller is neither the shortest nor the longest: accuracy peaks in the upper-middle of the length range, where targeted logic (e.g., error handling, deduplication, early stopping) has been added on top of the baseline loop. Second, further expansion beyond that point hurts. Unconstrained addition of code leads to "code bloat," introducing new bugs, over-constraining the execution loop, or creating brittle dependencies that fail under edge cases. This supports the design of our **contrastive reflection-guided intervention**: rather than expanding the codebase through unguided mutations, the method adds targeted, patch-like repairs and discards those that do not pay off. The most effective controller is the one that incorporates a limited set of precise repairs to structural vulnerabilities, not the one with the most code.



(a) Stage I: Accuracy vs. Code Tokens

(b) Stage II: Accuracy vs. Code Tokens

Figure 3: Comparison of Accuracy vs. Code Tokens across Stage I and Stage II.

B.2 Evolutionary Dynamics of Module Complexity

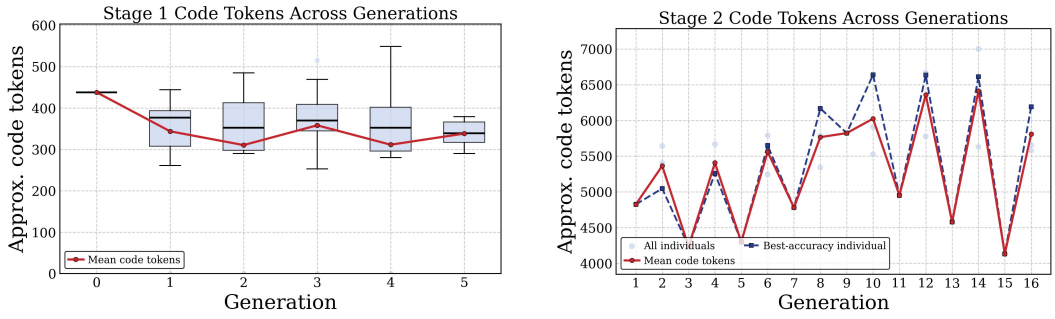
To gain deeper insights into the behavior of our meta-optimization process, we track the structural complexity of the evolved components (measured in approximate code tokens) across successive

generations, as depicted in Figure 4. The evolutionary trajectories of Stage I and Stage II exhibit fundamentally distinct dynamics, reflecting their different optimization landscapes.

Stage I: Semantic Compression and Distillation. Figure 4(a) displays the token distribution of the system prompt over 5 generations. Interestingly, rather than monotonically expanding the prompt with additional instructions, the evolution process exhibits a clear trend of structural compression. Starting from an initial length of over 400 tokens in Generation 0, the mean token count drops rapidly and stabilizes around 300–350 tokens in subsequent generations. This indicates that prompt evolution acts as an effective distillation mechanism. The meta-optimizer successfully identifies and prunes verbose, redundant, or confusing natural language instructions, refining the semantic scaffolding into a highly compact and potent reasoning mindset. This compression phenomenon demonstrates that a high-performing agent prompt relies on the precise articulation of constraints rather than the sheer volume of text.

Stage II: Alternating Expansion and Contraction in Code Space. Conversely, Figure 4(b) reveals a highly oscillatory, zigzag trajectory for the execution logic evolved in Stage II. Across 16 generations, the mean codebase length alternates between expansion generations at roughly 5,400–6,400 tokens (generations 2, 4, 6, 8–10, 12, 14, 16) and contraction generations at roughly 4,100–4,950 tokens (generations 3, 5, 7, 11, 13, 15). In expansion generations the meta-optimizer inserts substantial logical blocks (e.g., error-handling loops or stricter scheduling constraints); in contraction generations the population is rolled back toward a compact parent after the added logic fails validation or evaluation.

The "best-accuracy individual" (blue dashed line) tracks the *upper* envelope of this pattern in the later generations: from generation 8 onward, the best controller of each expansion generation lies at 6,150–6,650 tokens, and the overall best controller (generation 16, about 6,200 tokens) is the one identified as the accuracy peak in Figure 3(b). In other words, the gains in Stage II come from expansion steps whose added logic survives evaluation, while the contraction steps serve to discard additions that did not. Finding the optimal execution logic is therefore neither monotone feature accumulation nor pure pruning, but an alternating cycle of proposing a patch, testing it, and keeping only the patches that pay off.



(a) Stage I: Code Tokens by Generation

(b) Stage II: Code Tokens by Generation

Figure 4: Evolution of Code Tokens length across different generations in both stages.

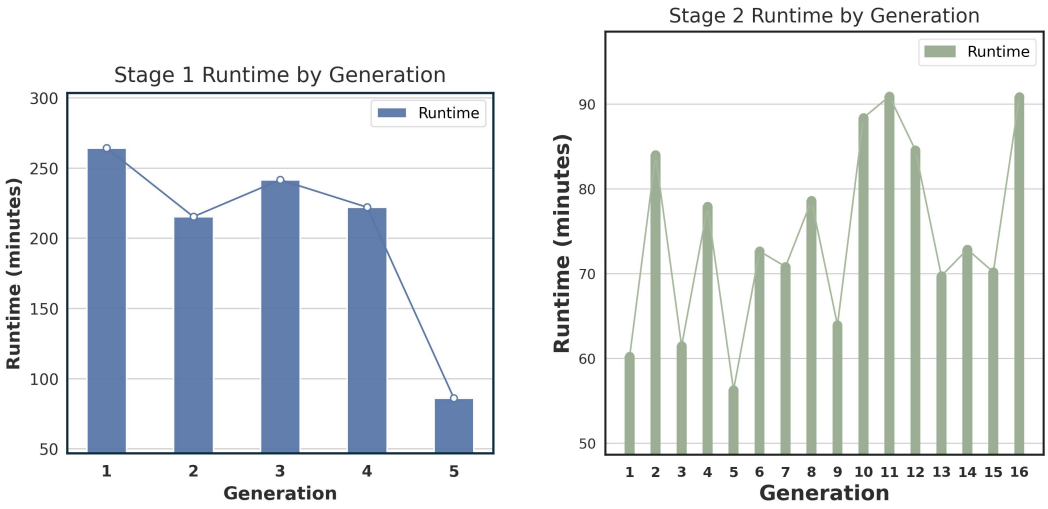
B.3 Computational Cost and Runtime Analysis

A fundamental claim of our frugal evolution framework is its viability under strict compute constraints. To empirically validate this, we analyze the wall-clock runtime (in minutes) for each generation during both evolutionary stages, as presented in Figure 5. The disparate runtime profiles of the two stages provide further insight into their underlying optimization mechanics.

Stage I: From High-Exploration to Efficient Convergence. Figure 5(a) shows the runtime for the semantic evolution stage. The initial generations exhibit a relatively high computational cost, averaging over 200 minutes per generation. This is primarily because evaluating unoptimized or exploratory reasoning prompts often leads the agent into deep, inefficient tool-use trajectories, extensive web searches, or repetitive error-correction loops, thereby consuming significant wall-clock

time. In Generation 5 the runtime drops to under 100 minutes. Two factors contribute, and the figure does not separate them: later-generation prompts produce shorter trajectories with fewer redundant searches, and the number of candidates evaluated per generation is not constant across the run. We therefore read the runtime curve as descriptive of the cost profile of Stage I rather than as evidence of convergence on its own; the accuracy trajectory in Figure 2b is the relevant signal for that.

Stage II: Agile Iteration and Fast-Fail Execution. Figure 5(b) shows that the logic re-coding stage runs at a consistently lower per-generation cost, fluctuating between 55 and 90 minutes. Two factors keep these numbers low. First, each Stage II generation evaluates fewer candidate–question pairs than a Stage I generation (about three to four controllers on 20 items, versus a larger prompt population on 10 items), so the per-generation figures of the two stages are not directly comparable. Second, programmatic execution fails fast: when the meta-optimizer injects a brittle patch, the execution loop typically raises an exception and exits early, so flawed controllers are cheap to reject. The oscillation in runtime follows the alternation between expansion and contraction generations described in Appendix B.2: generations that add substantial logic take longer to evaluate than those that roll back to a compact parent.



(a) Stage I: Generation Runtime

(b) Stage II: Generation Runtime

Figure 5: Comparison of computational runtime per generation between Stage I and Stage II.

C System Prompts and Implementations

C.1 Original WebThinker Multi-QA Prompt

Original WebThinker Multi-QA Prompt

```
def get_instruction(max_step):
    return (
        "You are a reasoning assistant with the ability to perform web searches to help "
        "you answer the user's question accurately. You have special tools:\n\n"
        "- To perform a search: write <|begin_search_query|> your query here <|
    end_search_query|>.\n"
        "Then, the system will search and analyze relevant web pages, then provide you
    with helpful information in the format <|begin_search_result|> ...search results...
    <|end_search_result|>.\n\n"
        f"You can repeat the search process multiple times if necessary. The maximum
    number of search attempts is limited to {max_step}.\n\n"
        "Once you have all the information you need, continue your reasoning.\n\n"
        "Example:\n"
        "Question: \"Alice David is the voice of Lara Croft in a video game developed by
    which company?\"\n"
        "Assistant thinking steps:\n"
        "- I need to find out who voices Lara Croft in the video game.\n"
```

```
        "- Then, I need to determine which company developed that video game.\n\n"
        "Assistant:\n"
        "<|begin_search_query|>Alice David Lara Croft voice<|end_search_query|>\n\n"
        "(System returns processed information from relevant web pages)\n\n"
        "Assistant thinks: The search results indicate that Alice David is the voice of Lara Croft in a specific video game. Now, I need to find out which company developed that game.\n\n"
        "Assistant:\n"
        "<|begin_search_query|>video game developed by Alice David Lara Croft<|end_search_query|>\n\n"
        "(System returns processed information from relevant web pages)\n\n"
        "Assistant continues reasoning with the new information...\n\n"
        "Remember:\n"
        "- Use <|begin_search_query|> to request a web search and end with <|end_search_query|>.\n"
        "- When done searching, continue your reasoning.\n\n"
    )
```

C.2 New Algorithm: Synthesize-Query-Reason-Verify (SQRV) Prompt

Synthesize-Query-Reason-Verify (SQRV) Prompt

```
def get_instruction(max_step):
    algorithm_sentence = "{New algorithm: Synthesize-Query-Reason-Verify, a loop that (1) creates a concise research plan with 2-3 prioritized hypotheses, (2) runs focused searches for each hypothesis using the required tags, (3) compiles a cross-source evidence matrix and flags contradictions, (4) applies logical consistency checks and credibility scoring, and (5) delivers a carefully qualified final answer with explicit uncertainties and citations.}"
    instruction = f"""You are a reasoning assistant with web-search capability to help answer complex questions accurately. You have special tools:

- To perform a search: write <|begin_search_query|> your query here <|end_search_query|>.
- Then, the system will search and analyze relevant web pages, returning results wrapped as <|begin_search_result|> ...search results... <|end_search_result|>.

You can repeat the search process multiple times if necessary. The maximum number of search attempts is limited to {max_step}.

{algorithm_sentence}

Algorithmic approach: Synthesize-Query-Reason-Verify loop. Before each search, generate a concise plan and 2-3 prioritized hypotheses. For each hypothesis, perform a focused search using the required tags. After collecting results, synthesize evidence into a compact matrix, identify contradictions, and note uncertainties. Cross-validate conclusions against at least two independent sources, weigh source credibility, and produce a final answer with caveats and citations.

Remember:
- Use <|begin_search_query|> to request a web search and end with <|end_search_query|>.
- After the last search, reflect and verify before giving the final answer.
- If results conflict, present the strongest supported conclusion and clearly state remaining uncertainties.
- Include brief source citations in the final answer where possible."""
    return instruction
```

C.3 Baseline Web Reasoning Pipeline

Baseline Web Reasoning Pipeline

```
async def process_single_sequence(
    seq: Dict,
    client: AsyncOpenAI,
    aux_client: AsyncOpenAI,
    semaphore: asyncio.Semaphore,
    args: argparse.Namespace,
    search_cache: Dict,
    url_cache: Dict,
    batch_output_records: List[Dict],
) -> Dict:
```

```
    """Process a single sequence through its entire reasoning chain with MAX_TOKENS limit
     """

    MAX_TOKENS = 40000
    total_tokens = len(seq['prompt'].split())

    # Initialize web explorer interactions list
    seq['web_explorer'] = []

    # First response uses chat completion
    formatted_prompt, response = await generate_response(
        client=client,
        model_name=args.model_name,
        prompt=seq['prompt'],
        semaphore=semaphore,
        temperature=args.temperature,
        top_p=args.top_p,
        max_tokens=args.max_tokens,
        repetition_penalty=args.repetition_penalty,
        top_k=args.top_k_sampling,
        min_p=args.min_p,
        stop=[END_SEARCH_QUERY],
    )

    # Update token count and sequence fields
    tokens_this_response = len(response.split())
    total_tokens += tokens_this_response

    seq['output'] += response.replace('</think>\n', '')
    seq['history'].append(response.replace('</think>\n', ''))
    seq['original_prompt'] = formatted_prompt
    seq['prompt'] = formatted_prompt + response.replace('</think>\n', '')

    while not seq['finished']:
        # Check if sequence is finished
        if not seq['output'].rstrip().endswith(END_SEARCH_QUERY):
            seq['finished'] = True
            break

        search_query = extract_between(response, BEGIN_SEARCH_QUERY, END_SEARCH_QUERY)
        seq['search_count'] += 1

        if seq['search_count'] < args.max_search_limit and total_tokens < MAX_TOKENS:
            if search_query is None or len(search_query) <= 5 or END_SEARCH_QUERY in search_query or search_query in invalid_search_queries:
                continue

            if search_query in seq['executed_search_queries']:
                # If search query was already executed, append message and continue
                append_text = f"\n\n{BEGIN_SEARCH_RESULT}You have already searched for this query.{END_SEARCH_RESULT}\n\nOkay,"
                seq['prompt'] += append_text
                seq['output'] += append_text
                seq['history'].append(append_text)
                total_tokens += len(append_text.split())
                continue

            _, search_intent = await generate_response(
                client=aux_client,
                model_name=args.aux_model_name,
                max_tokens=1000,
                prompt=get_search_intent_instruction(seq['output']),
                semaphore=semaphore,
            )

            #
            if search_query in search_cache:
                results = search_cache[search_query]
            else:
                try:
                    if args.search_engine == "bing":
                        results = await bing_web_search_async(search_query, args.bing_subscription_key, args.bing_endpoint)
                    elif args.search_engine == "serper":
                        results = await google_serper_search_async(search_query, args.serper_api_key)
                    else: # Should not happen
                        results = {}
                    search_cache[search_query] = results
```

```
                except Exception as e:
                    print(f"Error during search query '{search_query}' using {args.search_engine}: {e}")
                    results = {}
            print(f'Searched for: "{search_query}" using {args.search_engine}')

            if args.search_engine == "bing":
                relevant_info = extract_relevant_info(results)[:args.top_k]
            elif args.search_engine == "serper":
                relevant_info = extract_relevant_info_serper(results)[:args.top_k]
            else: # Should not happen
                relevant_info = []

            # Process documents
            urls_to_fetch = []
            for doc_info in relevant_info:
                url = doc_info['url']
                if url not in url_cache:
                    urls_to_fetch.append(url)

            if urls_to_fetch:
                try:
                    contents = await fetch_page_content_async(
                        urls_to_fetch,
                        use_jina=args.use_jina,
                        jina_api_key=args.jina_api_key,
                        keep_links=args.keep_links
                    )
                    for url, content in contents.items():
                        # Only cache content if it doesn't contain error indicators
                        has_error = (any(indicator.lower() in content.lower() for indicator in error_indicators) and len(content.split()) < 64) or len(content) < 50 or len(content.split()) < 20
                        if not has_error:
                            url_cache[url] = content
                except Exception as e:
                    print(f"Error fetching URLs: {e}")

            # Get web page information for each result
            for doc_info in relevant_info:
                url = doc_info['url']
                if url not in url_cache:
                    raw_content = ""
                else:
                    raw_content = url_cache[url]
                    is_success, raw_content = extract_snippet_with_context(raw_content, doc_info['snippet'], context_chars=2000)

                    # Check if content has error indicators
                    has_error = any(indicator.lower() in raw_content.lower() for indicator in error_indicators) or raw_content == ""

                    if has_error:
                        # If content has error, use it directly as summary
                        doc_info['page_info'] = "Can not fetch the page content."
                    else:
                        # Use raw content directly as page info
                        doc_info['page_info'] = raw_content

            formatted_documents = format_search_results(relevant_info)

            # Generate deep web exploration with interactions
            analysis, explorer_prompt = await generate_deep_web_explorer(
                client=client,
                aux_client=aux_client,
                search_query=search_query,
                search_intent=search_intent,
                document=formatted_documents,
                args=args,
                search_cache=search_cache,
                url_cache=url_cache,
                semaphore=semaphore,
            )

            extracted_info = extract_answer_fn(analysis, mode='summary')

            # Store web explorer input/output with all interactions
            seq['web_explorer'].append({
                "search_query": search_query,
```

```
                "Input": explorer_prompt,
                "Output": analysis,
                "Extracted_info": extracted_info
            })

            # Update sequence with search results
            append_text = f"\n\n{BEGIN_SEARCH_RESULT}{extracted_info}{END_SEARCH_RESULT}\n\n"
            seq['prompt'] += append_text
            seq['output'] += append_text
            seq['history'].append(append_text)

            seq['executed_search_queries'].add(search_query)
            total_tokens += len(append_text.split())

            # Subsequent responses use completion mode
            _, response = await generate_response(
                client=client,
                model_name=args.model_name,
                prompt=seq['prompt'],
                semaphore=semaphore,
                temperature=args.temperature,
                top_p=args.top_p,
                max_tokens=args.max_tokens,
                repetition_penalty=args.repetition_penalty,
                top_k=args.top_k_sampling,
                min_p=args.min_p,
                stop=[END_SEARCH_QUERY],
                generate_mode="completion"
            )

            # Update token count and sequence fields
            tokens_this_response = len(response.split())
            total_tokens += tokens_this_response

            seq['output'] += response.replace('</think>\n', '')
            seq['history'].append(response.replace('</think>\n', ''))
            seq['prompt'] += response.replace('</think>\n', '')
            continue

        else:
            append_text = f"\n\n{BEGIN_SEARCH_RESULT}You have reached the search limit. You are not allowed to search.{END_SEARCH_RESULT}\n\n"
            seq['prompt'] += append_text
            seq['output'] += append_text
            seq['history'].append(append_text)

            _, final_response = await generate_response(
                client=client,
                prompt=seq['prompt'],
                semaphore=semaphore,
                temperature=args.temperature,
                top_p=args.top_p,
                max_tokens=args.max_tokens,
                repetition_penalty=1.1,
                top_k=args.top_k_sampling,
                min_p=args.min_p,
                model_name=args.model_name,
                generate_mode="completion",
                bad_words=[f"{END_SEARCH_RESULT}\n\n{tokenizer.eos_token}", f"{END_SEARCH_QUERY}{tokenizer.eos_token}"]
            )

            seq['output'] += final_response
            seq['history'].append(final_response)
            seq['finished'] = True
            break

    return seq
```

C.4 Evolved V2 Web Reasoning Pipeline (Stage 2)

Evolved V2 Web Reasoning Pipeline (Stage 2)

```python
async def process_single_sequence_evolved_v2(
    seq: Dict,
    client,
    aux_client,
    semaphore: asyncio.Semaphore,
    args: argparse.Namespace,
    search_cache: Dict,
    url_cache: Dict,
    batch_output_records: List[Dict],
) -> Dict:
    """
    Evolved process_single_sequence focusing on an improved meta_strategy helper.
    Preserves main control flow (search -> extract -> reason) and async behavior.

    Primary evolution: meta_strategy() which decides whether to continue, refine,
    broaden, narrow, skip, or stop searches based on token budget, accumulated
    evidence, redundancy, cache hits, and query specificity.
    """

    # Configuration & defaults
    DEFAULT_MAX_TOKENS = int(getattr(args, "max_token_budget", 40000))
    MAX_SEARCH_LIMIT = int(getattr(args, "max_search_limit", 10))
    TOP_K = int(getattr(args, "top_k", getattr(args, "top_k_search", 5)))
    SEARCH_ENGINE = getattr(args, "search_engine", "").lower() if hasattr(args, "search_engine") else ""

    # Initialize sequence safely
    seq.setdefault("web_explorer", [])
    seq.setdefault("history", [])
    seq.setdefault("output", seq.get("output", "") or "")
    seq.setdefault("prompt", seq.get("prompt", "") or "")
    seq.setdefault("search_count", int(seq.get("search_count", 0) or 0))
    seq.setdefault("finished", bool(seq.get("finished", False)))
    seq.setdefault("executed_search_queries", seq.get("executed_search_queries", set()) or set())
    if isinstance(seq["executed_search_queries"], list):
        seq["executed_search_queries"] = set(seq["executed_search_queries"])
    seq.setdefault("original_prompt", seq.get("original_prompt", seq.get("prompt", "")))

    total_tokens = max(0, len(seq["prompt"].split()))

    # Small tokenization utility
    def tokset(text: str) -> set:
        if not text:
            return set()
        toks = []
        for t in text.split():
            t = t.strip(".,()[]{}:;\"'").lower()
            if t and len(t) > 2:
                toks.append(t)
        return set(toks)

    def overlap_ratio(a: str, b: str) -> float:
        sa = tokset(a)
        sb = tokset(b)
        if not sa or not sb:
            return 0.0
        return len(sa & sb) / float(len(sa | sb))

    def mean_overlap_with_history(q: str, history_queries: list) -> float:
        if not history_queries:
            return 0.0
        vals = [overlap_ratio(q, h) for h in history_queries if h]
        return sum(vals) / len(vals) if vals else 0.0

    def accumulated_chars(web_explorer_list=None) -> int:
        we = web_explorer_list if web_explorer_list is not None else seq.get("web_explorer", [])
        return sum(len(x.get("Extracted_info", "") or "") for x in we)

    def novelty_score(candidate: str, explored: list) -> float:
        if not candidate:
            return 0.0
        overlaps = []
```

```
        for e in explored:
            prev = e.get("search_query", "") or ""
            overlaps.append(overlap_ratio(candidate, prev))
        if not overlaps:
            return 1.0
        max_ov = max(overlaps)
        return max(0.0, 1.0 - max_ov)

    # ------------------------------------------------------------------------------
    # EVOLVED META STRATEGY
    # ------------------------------------------------------------------------------
    def meta_strategy(context: Dict) -> Dict:
        """
        Decide what to do with a proposed search query.

        Returns a dict with keys:
          - action: 'continue'|'modify_query'|'skip_search'|'stop'|'broaden_query'|'narrow_query'
          - modify_query: optional new query string
          - reason: explanation (for logging)
        """
        search_query = (context.get("search_query") or "").strip()
        web_explorer_local = context.get("web_explorer", []) or []
        total_tokens_local = int(context.get("total_tokens", 0) or 0)
        max_tokens_local = int(context.get("max_tokens", DEFAULT_MAX_TOKENS) or DEFAULT_MAX_TOKENS)
        search_count_local = int(context.get("search_count", 0) or 0)
        output = context.get("output", "") or ""
        search_cache_local = context.get("search_cache", {}) or {}
        executed = set(context.get("executed_search_queries", set()) or set())
        max_search_limit = int(context.get("max_search_limit", MAX_SEARCH_LIMIT) or MAX_SEARCH_LIMIT)

        # Basic guard
        if not search_query or len(search_query) < 3:
            return {"action": "skip_search", "reason": "Empty or trivial query"}

        # Derived metrics
        token_ratio = float(total_tokens_local) / float(max_tokens_local) if max_tokens_local else 1.0
        acc_chars = accumulated_chars(web_explorer_local)
        history_queries = [e.get("search_query", "") for e in web_explorer_local]
        searches_done = len(executed)

        # Always allow at least one search if none performed yet
        if searches_done == 0 and search_count_local >= max_search_limit:
            return {"action": "continue", "reason": "Allow initial search despite configured limit"}

        # If critically out of budget, stop. But allow one search if none done.
        if token_ratio >= 0.995:
            if searches_done == 0:
                return {"action": "continue", "reason": "Near exhaustion but no searches done yet"}
            return {"action": "stop", "reason": f"Token budget nearly exhausted ({token_ratio:.3f})"}

        # If lots of evidence and moderate token usage, stop
        if token_ratio > 0.92 and acc_chars >= 1000 and searches_done >= 1:
            return {"action": "stop", "reason": "High token usage and sufficient evidence collected"}

        # Respect search count limit (but allow first search)
        if search_count_local >= max_search_limit:
            if searches_done == 0:
                return {"action": "continue", "reason": "Allow initial search despite count limit"}
            return {"action": "stop", "reason": "Reached max search limit"}

        # Avoid exact duplicates
        if search_query in executed:
            return {"action": "skip_search", "reason": "Exact query already executed"}

        # Avoid high-overlap queries when we have reasonable evidence
        mean_ov = mean_overlap_with_history(search_query, history_queries)
        if mean_ov > 0.85 and acc_chars >= 400:
            return {"action": "skip_search", "reason": f"High overlap ({mean_ov:.2f}) with prior queries; avoid redundancy"}

```

```
        # Encourage novel queries if evidence is weak
        nov = novelty_score(search_query, web_explorer_local)
        if acc_chars < 300 and nov > 0.35:
            return {"action": "continue", "reason": "Low evidence and query is novel; proceed"}

        # Detect generic queries and try to refine
        lower_q = search_query.lower()
        q_tokens = tokset(search_query)
        generic_indicators = ["what is", "who is", "tell me", "about", "information", "explain", "definition", "overview", "?"]
        is_generic = any(ind in lower_q for ind in generic_indicators) or len(search_query) < 30 or len(q_tokens) <= 4

        if is_generic and search_count_local < max_search_limit:
            # Use recent local context to add keywords
            source_text = (output + " " + seq.get("prompt", ""))[-2000:]
            context_kws = [t for t in sorted(list(tokset(source_text)), key=lambda x: -len(x)) if t.isalpha() and len(t) >= 4]
            refined_kws = []
            seen = set()
            for k in context_kws:
                if k not in q_tokens and k not in seen:
                    refined_kws.append(k)
                    seen.add(k)
                if len(refined_kws) >= 4:
                    break
            if refined_kws:
                new_q = f"{search_query} {' '.join(refined_kws[:3])}"
                return {"action": "modify_query", "modify_query": new_q.strip(), "reason": "Refined generic query with contextual keywords"}
            # fallback to an overview-style query
            return {"action": "modify_query", "modify_query": f"{search_query} overview", "reason": "Generic query -> request overview"}

        # If cache exists but shows no results, broaden query
        cached = search_cache_local.get(search_query)
        if cached is not None:
            try:
                result_count = 0
                if isinstance(cached, dict):
                    if "webPages" in cached:
                        result_count = len(cached.get("webPages", {}).get("value", []))
                    elif "organic_results" in cached:
                        result_count = len(cached.get("organic_results", []))
                    else:
                        # best-effort
                        result_count = len(str(cached))
                else:
                    result_count = len(str(cached))
            except Exception:
                result_count = 0
            if result_count == 0 and search_count_local < max_search_limit:
                broadened = f"{search_query} review OR overview OR survey"
                return {"action": "broaden_query", "modify_query": broadened, "reason": "Cached results empty; broaden to overviews/reviews"}

        # Long/unfocused queries -> narrow to authoritative domains
        if len(search_query) > 120 or (len(q_tokens) > 12 and any(search_query.lower().startswith(w) for w in ["how", "why", "what", "who"])):
            narrowed = f"site:.gov OR site:.edu {search_query}"
            return {"action": "narrow_query", "modify_query": narrowed.strip(), "reason": "Long/complex query -> narrow to authoritative domains"}

        # If we have abundant diverse evidence, stop
        if acc_chars >= 1600 and searches_done >= 2 and mean_ov < 0.5:
            return {"action": "stop", "reason": "Sufficient diverse evidence gathered"}

        # Moderate token pressure: guide towards authoritative focused queries
        if 0.92 <= token_ratio < 0.995:
            if acc_chars >= 800 and searches_done >= 1:
                return {"action": "stop", "reason": "Elevated token usage and adequate evidence"}
            return {"action": "modify_query", "modify_query": f"{search_query} site:.edu OR site:.gov OR review", "reason": "Token usage elevated -> prefer focused authoritative results"}

        # Default: continue
        return {"action": "continue", "reason": "Default: perform search"}
```

```
    # ------------------------------------------------------------------------
    # INITIAL MODEL CALL (prompt -> response)
    # ------------------------------------------------------------------------
    formatted_prompt, response = await generate_response(
        client=client,
        model_name=getattr(args, "model_name", None),
        prompt=seq.get("prompt", ""),
        semaphore=semaphore,
        temperature=getattr(args, "temperature", 0.0),
        top_p=getattr(args, "top_p", 1.0),
        max_tokens=getattr(args, "max_tokens", 512),
        repetition_penalty=getattr(args, "repetition_penalty", None),
        top_k=getattr(args, "top_k_sampling", None),
        min_p=getattr(args, "min_p", None),
        stop=[END_SEARCH_QUERY],
    )

    tokens_this_response = len((response or "").split())
    total_tokens += tokens_this_response
    seq["output"] += (response or "").replace("</think>\n", "")
    seq["history"].append((response or "").replace("</think>\n", ""))
    seq["original_prompt"] = formatted_prompt
    seq["prompt"] = formatted_prompt + (response or "").replace("</think>\n", "")

    consecutive_skips = 0

    # ------------------------------------------------------------------------
    # MAIN LOOP: search -> extract -> reason
    # ------------------------------------------------------------------------
    while not seq.get("finished", False):
        # If model didn't propose a search, finish
        if not seq["output"].rstrip().endswith(END_SEARCH_QUERY):
            seq["finished"] = True
            break

        # Extract proposed search query
        search_query = extract_between(response, BEGIN_SEARCH_QUERY, END_SEARCH_QUERY)
        seq["search_count"] += 1

        # If no valid query proposed, nudge the model
        if search_query is None or len(search_query.strip()) == 0:
            nudge = "\n\n[Note]: The proposed search query was empty or invalid. Please refine.\n\n"
            seq["prompt"] += nudge
            seq["output"] += nudge
            seq["history"].append(nudge)
            total_tokens += len(nudge.split())
            _, response = await generate_response(
                client=client,
                model_name=getattr(args, "model_name", None),
                prompt=seq["prompt"],
                semaphore=semaphore,
                temperature=getattr(args, "temperature", 0.0),
                top_p=getattr(args, "top_p", 1.0),
                max_tokens=getattr(args, "max_tokens", 512),
                repetition_penalty=getattr(args, "repetition_penalty", None),
                top_k=getattr(args, "top_k_sampling", None),
                min_p=getattr(args, "min_p", None),
                stop=[END_SEARCH_QUERY],
            )
            tokens_this_response = len((response or "").split())
            total_tokens += tokens_this_response
            seq["output"] += (response or "").replace("</think>\n", "")
            seq["history"].append((response or "").replace("</think>\n", ""))
            seq["prompt"] += (response or "").replace("</think>\n", "")
            continue

        search_query = search_query.strip()

        # Already executed?
        if search_query in seq.get("executed_search_queries", set()):
            append_text = f"\n\n{BEGIN_SEARCH_RESULT}You have already searched for this query.{END_SEARCH_RESULT}\n\nOkay,"
            seq["prompt"] += append_text
            seq["output"] += append_text
            seq["history"].append(append_text)
            total_tokens += len(append_text.split())
            _, response = await generate_response(
```

```
                client=client,
                model_name=getattr(args, "model_name", None),
                prompt=seq["prompt"],
                semaphore=semaphore,
                temperature=getattr(args, "temperature", 0.0),
                top_p=getattr(args, "top_p", 1.0),
                max_tokens=getattr(args, "max_tokens", 512),
                repetition_penalty=getattr(args, "repetition_penalty", None),
                top_k=getattr(args, "top_k_sampling", None),
                min_p=getattr(args, "min_p", None),
                stop=[END_SEARCH_QUERY],
            )
            tokens_this_response = len((response or "").split())
            total_tokens += tokens_this_response
            seq["output"] += (response or "").replace("</think>\n", "")
            seq["history"].append((response or "").replace("</think>\n", ""))
            seq["prompt"] += (response or "").replace("</think>\n", "")
            continue

        # Build context and call meta_strategy before executing search
        decision = meta_strategy({
            "search_query": search_query,
            "web_explorer": seq.get("web_explorer", []),
            "total_tokens": total_tokens,
            "search_count": seq["search_count"],
            "output": seq["output"],
            "max_tokens": DEFAULT_MAX_TOKENS,
            "max_search_limit": MAX_SEARCH_LIMIT,
            "executed_search_queries": seq.get("executed_search_queries", set()),
            "search_cache": search_cache,
            "url_cache": url_cache,
        })

        action = decision.get("action", "continue")
        reason = decision.get("reason", "")

        if action == "stop":
            seq["finished"] = True
            append_text = f"\n\n{BEGIN_SEARCH_RESULT}Search stopped: {reason}{
END_SEARCH_RESULT}\n\n"
            seq["prompt"] += append_text
            seq["output"] += append_text
            seq["history"].append(append_text)
            total_tokens += len(append_text.split())
            break

        if action == "skip_search":
            consecutive_skips += 1
            append_text = f"\n\n{BEGIN_SEARCH_RESULT}Skipped search: {reason}{
END_SEARCH_RESULT}\n\n"
            seq["prompt"] += append_text
            seq["output"] += append_text
            seq["history"].append(append_text)
            total_tokens += len(append_text.split())
            # Avoid deadlock: after several skips, allow forced search attempt
            if consecutive_skips >= 4 and len(seq.get("executed_search_queries", set()))
== 0:
                consecutive_skips = 0
            _, response = await generate_response(
                client=client,
                model_name=getattr(args, "model_name", None),
                prompt=seq["prompt"],
                semaphore=semaphore,
                temperature=getattr(args, "temperature", 0.0),
                top_p=getattr(args, "top_p", 1.0),
                max_tokens=getattr(args, "max_tokens", 512),
                repetition_penalty=getattr(args, "repetition_penalty", None),
                top_k=getattr(args, "top_k_sampling", None),
                min_p=getattr(args, "min_p", None),
                stop=[END_SEARCH_QUERY],
            )
            tokens_this_response = len((response or "").split())
            total_tokens += tokens_this_response
            seq["output"] += (response or "").replace("</think>\n", "")
            seq["history"].append((response or "").replace("</think>\n", ""))
            seq["prompt"] += (response or "").replace("</think>\n", "")
            continue

        if action in ("modify_query", "broaden_query", "narrow_query"):
```

```
            new_q = decision.get("modify_query", "").strip()
            if new_q and new_q != search_query:
                seq["history"].append(f"[meta_strategy modified] '{search_query}' -> '{new_q}'. Reason: {reason}")
                search_query = new_q

        # Reset consecutive skip counter since we will execute (or refine then execute)
        consecutive_skips = 0

        # If short and budget allows, request aux LLM to refine into a targeted query
        if len(search_query.split()) <= 4 and total_tokens < DEFAULT_MAX_TOKENS * 0.99:
            refine_prompt = (
                "You are a search-query optimizer. Given the recent reasoning context and a proposed query,\n"
                "produce a single short, targeted search query designed to return high-quality, relevant results.\n"
                "Return only the query text.\n\n"
                f"Context (recent reasoning):\n{(seq.get('output','') or '')[-1200:]}\n\n"
                f"Original query:\n{search_query}\n\n"
                "If the original is already specific, make a minimal improvement; otherwise add specificity such as\n"
                "'study', 'review', 'meta-analysis', 'site:.gov', 'site:.edu', or 'case study' as appropriate."
            )
            try:
                _, refined = await generate_response(
                    client=aux_client,
                    model_name=getattr(args, "aux_model_name", getattr(args, "model_name", None)),
                    max_tokens=80,
                    prompt=refine_prompt,
                    semaphore=semaphore,
                    temperature=0.0,
                    top_p=1.0,
                )
                if refined:
                    refined_line = (refined or "").strip().splitlines()[0].strip().strip('"')
                    if refined_line and len(refined_line) > 3 and refined_line.lower() != search_query.lower():
                        seq["history"].append(f"[meta_strategy refined] '{search_query}' -> '{refined_line}'")
                        search_query = refined_line
                        total_tokens += len(refined_line.split())
            except Exception:
                # best-effort refinement only
                pass

        # Prepare search intent using aux LLM (best-effort)
        try:
            _, search_intent = await generate_response(
                client=aux_client,
                model_name=getattr(args, "aux_model_name", getattr(args, "model_name", None)),
                max_tokens=1000,
                prompt=get_search_intent_instruction(seq["output"]),
                semaphore=semaphore,
            )
        except Exception:
            search_intent = ""

        # Execute search (use cache)
        if search_query in search_cache:
            results = search_cache[search_query]
        else:
            try:
                engine_name = getattr(args, "search_engine", "").lower() if hasattr(args, "search_engine") else ""
                if engine_name == "bing":
                    results = await bing_web_search_async(search_query, getattr(args, "bing_subscription_key", None), getattr(args, "bing_endpoint", None))
                elif engine_name == "serper":
                    results = await google_serper_search_async(search_query, getattr(args, "serper_api_key", None))
                else:
                    results = {}
                search_cache[search_query] = results
            except Exception:
```

```
                results = {}
                search_cache[search_query] = results

        try:
            print(f'Searched for: "{search_query}" using {getattr(args, "search_engine", "unknown")}')
        except Exception:
            pass

        # Extract top-k relevant results
        try:
            top_k = TOP_K
        except Exception:
            top_k = 5

        engine = getattr(args, "search_engine", "")
        if engine and engine.lower() == "bing":
            relevant_info = extract_relevant_info(results)[:top_k]
        elif engine and engine.lower() == "serper":
            relevant_info = extract_relevant_info_serper(results)[:top_k]
        else:
            relevant_info = []

        # Fetch and cache page content for new URLs
        urls_to_fetch = []
        for doc_info in relevant_info:
            url = doc_info.get("url")
            if url and url not in url_cache:
                urls_to_fetch.append(url)

        if urls_to_fetch:
            try:
                contents = await fetch_page_content_async(
                    urls_to_fetch,
                    use_jina=getattr(args, "use_jina", False),
                    jina_api_key=getattr(args, "jina_api_key", None),
                    keep_links=getattr(args, "keep_links", False),
                )
                for url, content in (contents or {}).items():
                    has_error = (any(ind.lower() in (content or "").lower() for ind in error_indicators) and len((content or "").split()) < 64) or len((content or "")) < 50 or len((content or "").split()) < 20
                    if not has_error:
                        url_cache[url] = content
            except Exception:
                pass

        # Attach page content to docs and sanitise
        for doc_info in relevant_info:
            url = doc_info.get("url")
            if url not in url_cache:
                raw_content = ""
            else:
                raw_content = url_cache[url]
                try:
                    ok, raw_content = extract_snippet_with_context(raw_content, doc_info.get("snippet", ""), context_chars=2000)
                    if not ok:
                        raw_content = url_cache[url]
                except Exception:
                    raw_content = url_cache[url]
            has_error = any(indicator.lower() in (raw_content or "").lower() for indicator in error_indicators) or raw_content == ""
            if has_error:
                doc_info["page_info"] = "Can not fetch the page content."
            else:
                doc_info["page_info"] = raw_content

        formatted_documents = format_search_results(relevant_info)

        # Deep exploration (preserved)
        analysis, explorer_prompt = await generate_deep_web_explorer(
            client=client,
            aux_client=aux_client,
            search_query=search_query,
            search_intent=search_intent,
            document=formatted_documents,
            args=args,
            search_cache=search_cache,
```

```
            url_cache=url_cache,
            semaphore=semaphore,
        )

        extracted_info = extract_answer_fn(analysis, mode="summary")

        # Store exploration result
        seq["web_explorer"].append({
            "search_query": search_query,
            "Input": explorer_prompt,
            "Output": analysis,
            "Extracted_info": extracted_info
        })

        append_text = f"\n\n{BEGIN_SEARCH_RESULT}{extracted_info}{END_SEARCH_RESULT}\n\n"
        seq["prompt"] += append_text
        seq["output"] += append_text
        seq["history"].append(append_text)
        seq["executed_search_queries"].add(search_query)
        total_tokens += len(append_text.split())

        # If extracted info is low-quality and budget allows, suggest follow-up
        low_quality = len((extracted_info or "").split()) < 30
        if low_quality and total_tokens < DEFAULT_MAX_TOKENS * 0.985 and seq["search_count"] < getattr(args, "max_search_limit", MAX_SEARCH_LIMIT):
            follow_up_q = f'{search_query} deeper evidence sources credibility'
            if follow_up_q not in seq["executed_search_queries"]:
                append_text2 = f"\n\n{BEGIN_SEARCH_RESULT}Low-quality extraction detected; consider follow-up search: {follow_up_q}{END_SEARCH_RESULT}\n\n"
                seq["prompt"] += append_text2
                seq["output"] += append_text2
                seq["history"].append(append_text2)
                total_tokens += len(append_text2.split())

        # Model produces next reasoning / possible next search
        _, response = await generate_response(
            client=client,
            model_name=getattr(args, "model_name", None),
            prompt=seq["prompt"],
            semaphore=semaphore,
            temperature=getattr(args, "temperature", 0.0),
            top_p=getattr(args, "top_p", 1.0),
            max_tokens=getattr(args, "max_tokens", 512),
            repetition_penalty=getattr(args, "repetition_penalty", None),
            top_k=getattr(args, "top_k_sampling", None),
            min_p=getattr(args, "min_p", None),
            stop=[END_SEARCH_QUERY],
        )

        tokens_this_response = len((response or "").split())
        total_tokens += tokens_this_response
        seq["output"] += (response or "").replace("</think>\n", "")
        seq["history"].append((response or "").replace("</think>\n", ""))
        seq["prompt"] += (response or "").replace("</think>\n", "")

        # Termination guards
        if seq["search_count"] >= getattr(args, "max_search_limit", MAX_SEARCH_LIMIT) or total_tokens >= DEFAULT_MAX_TOKENS:
            seq["finished"] = True
            break

    # Normalize executed_search_queries to list for compatibility
    try:
        seq["executed_search_queries"] = list(seq["executed_search_queries"])
    except Exception:
        pass

    return seq
```

C.5 META-OPTIMIZER PROMPTS FOR TWO-STAGE EVOLUTION

Stage 1 Meta-Optimizer Prompt

System Prompt:
I need help designing a novel prompt generation function for an AI Agent called "WebThinker". This function should return a string instruction that guides the Agent to answer complex user questions by performing web searches. The Agent uses `<|begin_search_query|>...<|end_search_query|>` to search. The goal is to maximize the accuracy of the Agent's answers on difficult benchmarks.

Stage 2 Meta-Optimizer Prompt

System Prompt:
Evolve the entire `process_single_sequence_evolved` function to improve GAIA benchmark accuracy.
PRIMARY FOCUS: Evolve the `meta_strategy()` helper function defined inside the main function.
SECONDARY: You can adjust WHERE and HOW to call `meta_strategy()` within the main control flow.
PRESERVE: Keep the main control flow (search → extract → reason) intact. DO NOT remove core functionality.
GOAL: Maximize answer accuracy. The objective is to MINIMIZE error rate (1 - accuracy).
CONSTRAINTS:

- DO NOT remove the search functionality entirely
- DO NOT skip deep exploration unless truly necessary (e.g., token budget exhausted)
- DO NOT change the function signature
- DO NOT modify external API calls (search, fetch, generate_response)
- DO NOT break the async/await pattern

GOOD CHANGES (encouraged):

- Optimize search query formulation
- Decide intelligently when to do deep exploration
- Manage token budget wisely (but don't be too conservative!)
- Determine when to stop vs continue based on information quality
- Improve information synthesis
- Deduplicate redundant searches
- Refine search queries when results are poor

BAD CHANGES (will hurt performance):

- Blocking all searches or most searches
- Stopping too early before gathering sufficient information
- Removing deep exploration entirely
- Being overly conservative with token budget (e.g., stopping at 85% usage)
- Breaking the main control flow

IMPORTANT NOTES:

1. The agent NEEDS to search to answer questions correctly. Don't prevent it from searching.
2. Token budget of 40000 is generous - using 90-95% is fine.
3. Deep exploration is valuable - only skip it when budget is truly tight (>95% tokens used).
4. The baseline (empty `meta_strategy`) achieves ∼40% accuracy. Your goal is to IMPROVE this.
5. Focus on WHEN to search and HOW to search, not WHETHER to search.

EVALUATION:

- You will be evaluated on 20 questions from GAIA
- Fitness = error_rate = 1 - accuracy

- Lower fitness is better
- Constraints: If searches=0, fitness is heavily penalized (2.0)

C.6 Dual-Track Evolution Procedure

Algorithm 1 summarizes the two independent evolution stages and the final agent assembly.

Algorithm 1 Dual-Track Agent-Evolution Framework

Require: Base $\langle P_0, L_0 \rangle$, Data $\mathcal{D}_P, \mathcal{D}_L$, Budgets B_P, B_L, Meta-Opt Φ
Stage I: Semantic Evolution *// Optimize Prompt P*
Pop $\mathcal{X} \leftarrow \{P_0\}$
for $t = 1$ **to** B_P **do**
  $\mathcal{E} \leftarrow \text{SelectElites}(\mathcal{X}, \mathcal{D}_P)$
  $\mathcal{X} \leftarrow \mathcal{E} \cup \{\text{Mutate}(\mathcal{E}, \Phi)\}$
end for
Stage II: Logic Re-coding *// Optimize Logic L*
Pop $\mathcal{L} \leftarrow \{L_0\}$, Mem $\mathcal{R} \leftarrow \emptyset$
for $t = 1$ **to** B_L **do**
  $g \leftarrow \text{ContrastiveReflect}(L^{\text{good}}, L^{\text{bad}} \in \mathcal{L}, \Phi)$
  $\mathcal{L} \leftarrow \mathcal{L} \cup \{\text{Reconstruct}(L^{\text{good}}, g, \mathcal{R}, \Phi)\}, \quad \mathcal{R} \leftarrow \mathcal{R} \cup \{g\}$
end for
return $\langle \arg\max_{P \in \mathcal{X}} \text{Acc}(P, L_0), \arg\max_{L \in \mathcal{L}} \text{Acc}(P_0, L) \rangle$

D Limitations and Broader Impacts

D.1 Limitations

While our frugal evolution framework dramatically reduces the compute and data barriers for agent optimization, it inherently possesses several limitations that warrant future investigation:

❶ **Dependency on Meta-Optimizer Capability.** The efficacy of the framework, particularly Stage II (Logic Re-coding), heavily relies on the reasoning and coding proficiency of the external meta-optimizer (Φ). If Φ struggles with complex code comprehension or frequently generates structurally invalid patches, the limited evaluation budget will be squandered on syntax errors rather than meaningful logical exploration.

❷ **Scope of Surgical Interventions.** Our contrastive reflection-guided mechanism excels at identifying structural brittleness and applying patch-like repairs (e.g., adding early-stopping, deduplication, or error-handling). However, it is fundamentally an optimization of an existing architecture. It may struggle to autonomously invent entirely novel, complex algorithmic paradigms from scratch without human prior.

❸ **Requirement for Architectural Transparency.** The framework requires a "gray-box" setup where the agent's controller logic and system prompts are accessible and modifiable. It cannot be directly applied to completely black-box commercial agents where the internal execution loop is hidden behind restrictive APIs.

❹ **Task Domain Generalization.** Although extensively evaluated on rigorous reasoning and tool-use benchmarks (GAIA, HLE), the applicability of this frugal meta-optimization to highly subjective tasks (e.g., creative writing) or continuous-control embodied robotics remains untested and represents an exciting avenue for future work.

❺ **Choice of Intervention Surface.** We select P and L on the grounds given in Section 3.2 and do not compare against optimizing alternative surfaces (e.g., retrieval hyper-parameters or result formatting) under an equal budget. Other surfaces may also yield gains, and an equal-budget comparison across surfaces is left for future work.

D.2 Broader Impacts

❶ **Democratization of Agentic Research.** On a positive note, our findings advocate for a highly resource-efficient paradigm. By demonstrating that massive datasets and GPU clusters are not strict prerequisites for automated agent improvement, this work lowers the barrier to entry. It empowers smaller research laboratories, academic institutions, and individual developers to build and refine capable ASI-trajectory systems, fostering a more inclusive and diverse AI research community.

❷ **Safety and Alignment Risks.** Conversely, automatically modifying an agent's execution logic introduces safety considerations. An agent whose controller is rewritten purely for benchmark accuracy might inadvertently (or convergently) alter or remove hardcoded safety guardrails within its controller. In a locally deployed environment with real tool access, unrestricted code evolution could lead to unintended or harmful actions. Therefore, it is imperative that our framework operates under strict interface constraints (Ω). Any future deployment of code-evolving agents must be conducted within rigorously isolated sandbox environments to prevent unaligned behaviors.