MARS: MECHANISM-AWARE REDESIGN SEARCH FOR AGENTIC SYSTEMS

Anonymous authors
Paper under double-blind review

ABSTRACT

Automated agent design reduces manual engineering by searching for effective agentic systems. Representing agents as executable code enables a broad range of designs, but also creates a vast search space that is difficult to navigate. In this expansive space, coarse-grained evaluation feedback makes it difficult to attribute performance bottlenecks to specific parts of an agent's design, resulting in a lack of targeted guidance during search. We introduce **Mechanism-Aware Redesign Search (MARS)**, which establishes an explicit correspondence between executable agent programs and a mechanism-level semantic representation of their composition, instantiation, and interaction. This representation aligns agent design, failure attribution, and redesign at a common mechanism-level granularity. Grounded in executable code and observed execution, MARS attributes failures to individual mechanisms and their interactions, using the resulting diagnoses to guide targeted redesign. Across five tasks, MARS discovers agents that outperform the strongest automated-design baseline on each task while consuming fewer inference-time LLM tokens. Notably, MARS surpasses the best results achieved by all compared search baselines within the first two generations.

1 INTRODUCTION

Recent large language models (LLMs) exhibit strong reasoning and problem-solving capabilities (DeepSeek-AI, 2026) and increasingly serve as core components of compound agentic systems that combine reasoning, tool use, reflection, verification, aggregation, and collaboration (Yao et al., 2023; Shinn et al., 2023; Hong et al., 2024; Wu et al., 2024; Chen et al., 2024). The effectiveness of such systems depends not only on the underlying model but also on how computational components are selected, implemented, and organized, making manual agent design labor-intensive and task-specific (Shang et al., 2025; Zhou et al., 2026). This has motivated growing interest in automated agent design and workflow search (Zhuge et al., 2024; Hu et al., 2025; Zhang et al., 2025c; Shang et al., 2025; Zhou et al., 2026; Saad-Falcon et al., 2025; Li et al., 2024).

Existing approaches to automated agent design differ in how they define the search space. Structured approaches incorporate human-designed structural priors into graph-, module-, topology-, team-, or supernet-based design spaces (Zhuge et al., 2024; Liu et al., 2024; Shang et al., 2025; Zhang et al., 2025b; Zhou et al., 2026; Zhang et al., 2025a; Saad-Falcon et al., 2025). These priors narrow the search space, making search more tractable but also constraining the range of agent designs that can be discovered. In contrast, code-based approaches such as ADAS and AFlow offer greater implementation flexibility (Hu et al., 2025; Zhang et al., 2025c), but must navigate a substantially larger space of possible designs. Code-level search can vary the functional steps an agent performs, how these steps are implemented, and how they are orchestrated, creating numerous possible directions for redesign. Effectively exploring this expansive space therefore remains a central challenge for automated agent search.

The expressiveness of programming languages enables a broad range of agent designs, encompassing prompts, tool use, workflows, and arbitrary combinations of these elements (Hu et al., 2025). This flexibility creates numerous possibilities for redesign, making it difficult to determine which changes can effectively address observed failures. Aggregate evaluation scores provide only coarse-

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

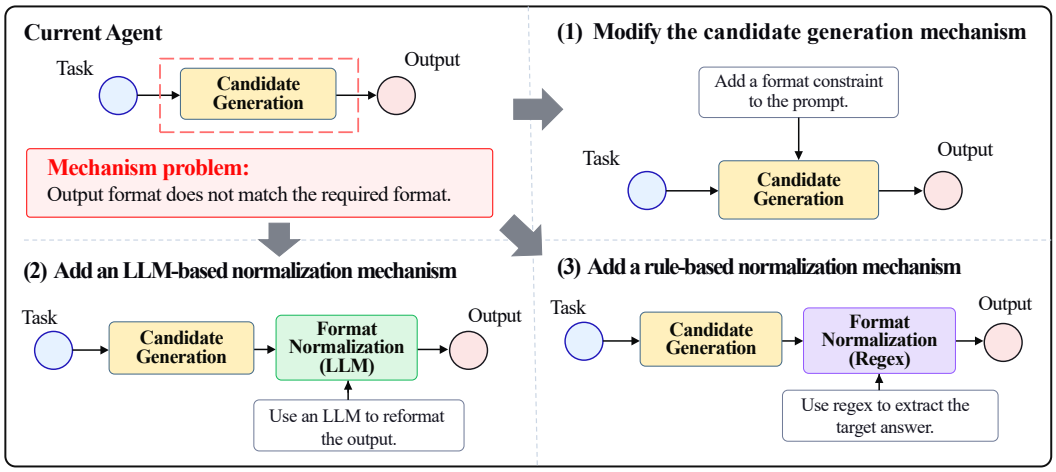


Figure 1: **Alternative redesigns for a localized mechanism problem.** A candidate generation mechanism produces an output that violates the required format. The identified problem can be addressed by modifying the existing mechanism or introducing an LLM-based or deterministic normalization step, illustrating how a specific diagnosis can guide different redesign strategies.

grained feedback (Hu et al., 2025), making it difficult to attribute performance bottlenecks to specific parts of an agent's design. Although prior work has explored textual feedback, module-level optimization, and execution traces (Yüksekgönül et al., 2025; Opsahl-Ong et al., 2024; Cheng et al., 2024), mapping observed failures to editable functional parts of an open-ended agent program remains challenging. This ambiguity also affects subsequent search decisions, as evaluation scores alone offer little guidance on which parts of an agent should be redesigned. Consequently, evidence accumulated from previous candidates remains difficult to translate into targeted design changes. Without such a correspondence, search lacks the attribution needed to guide targeted redesign.

To address this gap, we introduce a mechanism-level semantic representation that characterizes agent designs along three dimensions: mechanism *composition*, *instantiation*, and *interaction*. By establishing an explicit correspondence between this representation and executable agent code, we align agent design, failure attribution, and redesign at a common mechanism-level granularity. Building on this representation, we propose **Mechanism-Aware Redesign Search (MARS)**, which analyzes executable code and observed execution to trace information flow and diagnose problems in individual mechanisms and their interactions. These diagnoses guide targeted redesign of the identified bottlenecks, enabling MARS to explore different ways to improve an agent's functional design. As illustrated in Figure 1, an identified problem can motivate alternative redesigns, from modifying an existing procedure to introducing additional processing steps.

Across five tasks, MARS discovers agents that outperform the strongest automated-design baseline on each task while consuming fewer inference-time LLM tokens. Notably, MARS surpasses the best results achieved by all compared search baselines within the first two generations. These results demonstrate the effectiveness of mechanism-aware redesign in discovering high-performing, inference-efficient agents.

The key contributions are:

- We introduce a mechanism-level semantic representation that captures mechanism *composition*, *instantiation*, and *interaction*, establishing an explicit correspondence between functional agent designs and their executable implementations.
- Building on this representation, we propose **MARS**, which aligns agent design, failure attribution, and redesign at a common mechanism-level granularity and uses execution-grounded diagnoses to guide targeted search.
- Across five tasks, MARS discovers agents that outperform the strongest automated-design baseline on each task while consuming fewer inference-time LLM tokens, surpassing all compared search baselines within the first two generations. Moreover, on every task, MARS discovers at least one candidate that achieves a higher test score than the highest-

scoring representative agent among ADAS, AFlow, and MaAS, while consuming fewer inference-time LLM tokens than the least expensive of these three agents.

2 MECHANISM-LEVEL REPRESENTATION OF AGENT DESIGNS

MARS maintains each candidate through two complementary views: executable agent code A, which determines actual behavior, and a mechanism-level semantic representation R, which exposes the functional design decisions used for diagnosis and redesign. Higher-level representations and programming abstractions have similarly been used to expose structure above implementation-specific agent code for specification, execution, or analysis (Amini et al., 2026; Hou et al., 2026; Wang et al., 2026b; Zheng et al., 2024b). In MARS, R provides a unified semantic foundation that connects agent design with execution-grounded diagnosis. By describing mechanism instances, their concrete realizations, and their inputs and outputs, R enables MARS to reconstruct the answer flow from executable code and observed behavior, attribute observed failures to specific mechanisms or their interactions, and use these diagnoses to guide subsequent redesign.

2.1 AGENTIC MECHANISMS

Agentic systems often organize computation into procedures that serve distinct purposes in solving a task (Yao et al., 2023; Park et al., 2023; Shen et al., 2023; Wang et al., 2024a). We define an *agentic mechanism* as a functional unit characterized by its design intent: what information it consumes, what computation or operation it performs, and what result it is expected to produce. This level of abstraction captures functional design decisions across different code implementations and provides a common unit for describing agent designs, analyzing execution behavior, and attributing observed failures.

Based on their design intents and functional roles, MARS organizes mechanisms into eight *mechanism families*: intermediate analysis, candidate generation, procedure construction, execution and operation, evaluation and revision, aggregation and decision, routing and control, and output processing and normalization. These families provide common semantic units for describing and comparing agent designs across different implementations. Each family characterizes a functional role that can be realized in different ways. For example, aggregation and decision may involve using an LLM to select among multiple candidate answers, applying majority voting, or synthesizing complementary information from multiple upstream results. The definitions of all mechanism families are provided in Appendix A.1.

A *mechanism instance* is a concrete realization of a mechanism family within an agent. The same family can be instantiated in different ways to serve different task-specific purposes, with each instance characterized by its specific function, inputs, outputs, and implementation. An LLM-based instance is realized through a single LLM invocation, whereas a deterministic instance performs its function through programmatically specified computations or actions, such as rule-based processing, information extraction, or program execution. An agent composes multiple mechanism instances and orchestrates their interactions to accomplish a task.

2.2 SEMANTIC REPRESENTATION

Let R represent n uniquely identified mechanism instances:

$$R = \{m_i\}_{i=1}^{n}, \qquad m_i = \big(i, f_i, \mathrm{type}_i, \mathrm{in}_i, \mathrm{out}_i, \theta_i\big), \tag{1}$$

where m_i denotes the i-th mechanism instance, i is its unique identifier, f_i its mechanism family, type_i its implementation type, in_i and out_i its inputs and outputs, and θ_i its concrete realization. We distinguish $\mathrm{type}_i \in \{\mathrm{LLM}, \mathrm{Det}\}$. The identifier distinguishes different instances, even when they share the same mechanism family and implementation.

We represent agent designs along three dimensions: *composition*, *instantiation*, and *interaction*.

Composition. Composition specifies which mechanism families are present in an agent and the number of mechanisms belonging to each family. It characterizes the functional structure of the agent design.

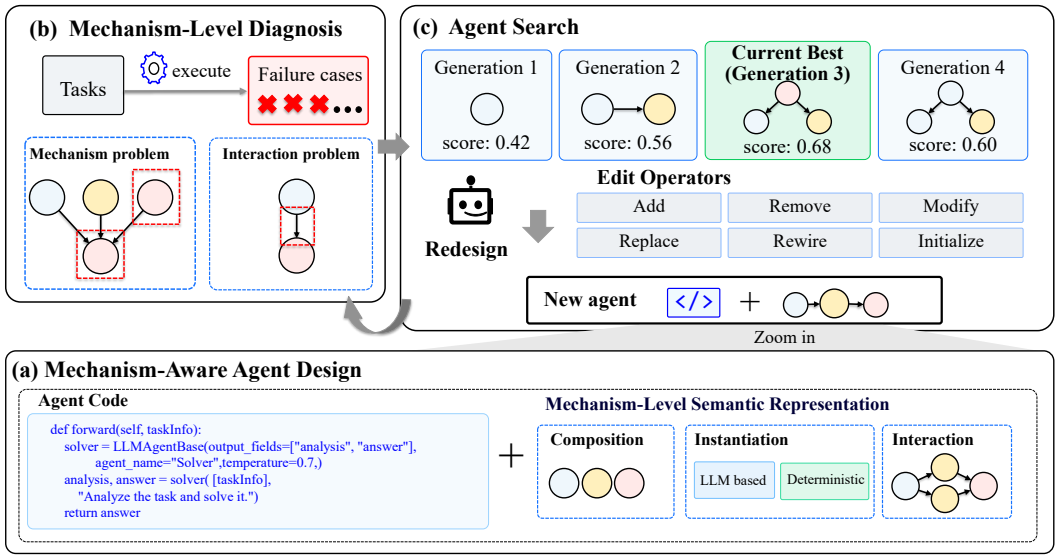


Figure 2: **Overview of MARS.** (a) Each candidate agent contains executable code and a mechanism-level semantic representation. (b) Execution evidence is used to diagnose mechanism and interaction problems. (c) Diagnoses guide the selection of edit operators to generate new candidate agents.

Instantiation. Instantiation specifies how each mechanism realizes its intended function:

$$\theta_i^{\mathrm{LLM}} = (P_i, \tau_i, \Phi_i), \qquad \theta_i^{\mathrm{Det}} = D_i, \tag{2}$$

where P_i, τ_i, and Φ_i denote the prompt, temperature, and structured output fields of an LLM invocation, respectively. Each LLM-based mechanism instance corresponds to a single LLM invocation in the agent program. D_i specifies programmatic computations or actions, such as rule-based processing, information extraction, or program execution. The mechanism-level inputs and outputs in_i and out_i describe the information consumed and produced by the mechanism, while Φ_i specifies the structured output fields of its LLM invocation.

Interaction. Interaction describes how mechanism instances are orchestrated through *information flow* and *control flow* (Garlan & Shaw, 1994; Allen & Garlan, 1997). Information flow is organized through the inputs and outputs of mechanisms, with the actual transfer and use of information specified in the executable code A. Control flow is expressed through the conditional logic and control structures in A, including branching, iteration, stopping, fallback, and return paths. Thus, mechanism interactions are described by the declared inputs and outputs in R together with their concrete realization in A.

MARS grounds mechanism-level diagnosis in the semantic representation R, executable code A, and execution traces. By jointly examining these sources, MARS reconstructs the answer flow and attributes observed failures to specific mechanisms or their interactions, providing targeted guidance for subsequent redesign.

3 MARS: Mechanism-Aware Redesign Search

MARS extends candidate-level evaluation with mechanism-level diagnosis to guide agent redesign. As shown in Figure 2, MARS attributes observed failures to individual mechanisms or their interactions and uses the resulting diagnoses to guide the selection of edit operators for generating new candidate agents.

3.1 Mechanism-Level Diagnosis

Validation scores summarize task performance without revealing which parts of an agent design are responsible for observed failures. Existing methods exploit textual feedback and execution traces to

provide richer optimization signals, yet attributing failures to specific components and their interactions remains challenging (Yüksekgönül et al., 2025; Opsahl-Ong et al., 2024; Cheng et al., 2024; Lee et al., 2025). Execution traces typically record LLM invocations and externally observable operations or actions, while internal deterministic computations or operations may not be explicitly recorded. Furthermore, execution traces can be fragmented and disorganized, making relevant evidence difficult to identify and interpret (Chang et al., 2026). MARS therefore uses mechanism-level semantics to connect execution evidence with the functional design of an agent and support more specific failure attribution.

For each candidate evaluated on the validation set, MARS summarizes its task-level evaluation outcomes and analyzes failure cases to reconstruct the answer flow through the mechanisms involved in producing the result. This analysis examines how intermediate information is generated, propagated, and used, and identifies the problems contributing to each failure (Reiter, 1987; de Kleer & Williams, 1987). These case-level analyses are then aggregated into a generation-level diagnosis that identifies recurring problems in the evaluated design.

MARS distinguishes two types of diagnosed failure causes: *mechanism problems* and *interaction problems*. A mechanism problem occurs when an individual mechanism fails to fulfill its intended function. For example, a candidate generation mechanism may fail to produce a correct candidate, an evaluation and revision mechanism may fail to detect errors in an existing candidate, or an output normalization mechanism may fail to correctly extract the target answer. An interaction problem arises when the relationships among mechanisms prevent information from being appropriately propagated or used, such as when useful upstream information is lost, ignored, overwritten, or prevented from reaching the final answer path. Related work has also investigated component importance and failure localization through counterfactual or trajectory-based analysis (Chen et al., 2025; Chang et al., 2026). To support consistent attribution, MARS categorizes diagnosed problems according to execution and output reliability, implementation fidelity, mechanism activation, mechanism quality, and cross-mechanism interaction. Cases lacking sufficient evidence for reliable attribution are explicitly recorded as such. The complete category definitions are provided in Appendix A.2.

Diagnosis identifies which parts of an agent design contribute to observed limitations and characterizes the problems they exhibit. The meta-searcher uses these diagnoses to select edit operators for targeted redesign, allowing modifications to individual mechanisms, their interactions, or multiple parts of the design within a single redesign.

3.2 Edit Operators

Prior work has employed explicit edits to optimize agent workflows (Zhang et al., 2025c). To support targeted redesign, MARS defines six mechanism-level edit operators:

$$\Omega = \{\textsc{Add}, \textsc{Remove}, \textsc{Modify}, \textsc{Replace}, \textsc{Rewire}, \textsc{Initialize}\}. \tag{3}$$

Add introduces a new mechanism; Remove deletes an existing mechanism; Modify changes a mechanism's instantiation while preserving its functional role; Replace substitutes an existing mechanism with a different mechanism; and Rewire modifies the interactions among existing mechanisms. Initialize constructs a new architecture without using the current best candidate as its implementation base.

These operators enable the meta-searcher to modify mechanism composition, instantiation, and interaction. Guided by the diagnosed problems and search history, the meta-searcher selects one or more edit operators and determines the corresponding changes to the agent design.

3.3 Search Algorithm

MARS uses a meta-searcher to iteratively generate executable agents based on previously evaluated designs (Romera-Paredes et al., 2024; Hu et al., 2025). It maintains a Search History that records evaluated candidates, their validation outcomes, mechanism-level diagnoses, and redesign records to guide subsequent search.

The search begins with an empty Search History. The meta-searcher first generates an initial candidate $c_0 = (A_0, R_0)$ using Initialize, establishing a simple single-call agent designated Genera-

Algorithm 1 Mechanism-Aware Redesign Search

Require: Validation set $\mathcal{D}_{\text{val}}$, search budget T, edit operators Ω
Ensure: Best candidate agent
1: Generate initial candidate c_0 using INITIALIZE
2: Evaluate and diagnose c_0 on $\mathcal{D}_{\text{val}}$
3: Initialize $\mathcal{H}_0$ with c_0 and its evaluation and diagnosis
4: **for** $t = 0, \ldots, T-1$ **do**
5: Select current best $c_t^\star$ from $\mathcal{H}_t$
6: Extract diagnostic context Γ_t from $\mathcal{H}_t$
7: $c_{t+1} \leftarrow \mathcal{S}(c_t^\star, \Gamma_t, \mathcal{H}_t; \Omega)$
8: Evaluate and diagnose c_{t+1} on $\mathcal{D}_{\text{val}}$
9: Update $\mathcal{H}_{t+1}$ with the new candidate and its evaluation, diagnosis, and edits
10: **end for**
11: **return** the best candidate in $\mathcal{H}_T$

tion 0. After evaluation and diagnosis, the candidate and its results are recorded in $\mathcal{H}_0$. At iteration t, the candidate with the highest validation score in $\mathcal{H}_t$ is selected as the current best $c_t^\star$, with ties broken by lower average runtime token consumption. Let Γ_t denote the diagnostic context extracted from the Search History. The next candidate is generated as

$$c_{t+1} = \mathcal{S}(c_t^\star, \Gamma_t, \mathcal{H}_t; \Omega), \tag{4}$$

where $c_{t+1} = (A_{t+1}, R_{t+1})$ contains executable code and its mechanism-level semantic representation.

The meta-searcher selects the implementation base and edit operators during candidate generation. It may refine the current best candidate through diagnosis-guided edits or use INITIALIZE to explore substantially different architectures, allowing the search to move beyond the current design. Each generated candidate is evaluated on the validation set and undergoes mechanism-level diagnosis. The candidate, its validation outcome, diagnosis, and redesign record are then added to the Search History. Algorithm 1 summarizes the search procedure.

4 EXPERIMENTS

4.1 EXPERIMENTAL SETUP

Task Setup and Metrics. We evaluate MARS on five benchmarks: E2H-Codeforces (Ding et al., 2024), DROP (Dua et al., 2019), GPQA-Diamond (Rein et al., 2024), HotpotQA (Yang et al., 2018), and Logic-Q from QuestBench (Li et al., 2025). We report pass rate for E2H-Codeforces, F1 for DROP and HotpotQA, and accuracy for GPQA-Diamond and Logic-Q. We also report average inference tokens per test instance, including input and output tokens from all LLM calls during agent execution. Further details on dataset construction and data splits are provided in Appendix A.3.

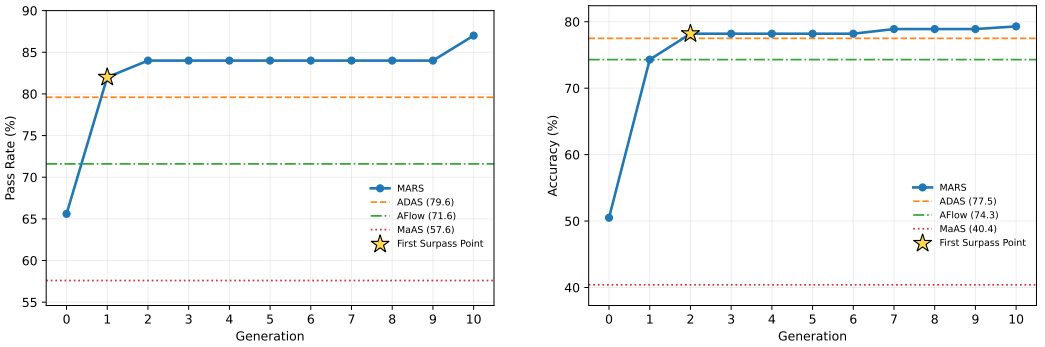
Baselines and Models. We compare against seven manually designed agents: Chain-of-Thought (CoT) (Wei et al., 2022), Self-Consistency CoT (Wang et al., 2023), Reflexion (Madaan et al., 2023; Shinn et al., 2023), LLM Debate (Du et al., 2024), Step-back Abstraction (Zheng et al., 2024a), Quality-Diversity (Lu et al., 2025), and Dynamic Roles (Xu et al., 2025). These also form the initial agent pool of ADAS. We further compare with ADAS (Hu et al., 2025), AFlow (Zhang et al., 2025c), and MaAS (Zhang et al., 2025a). We use DeepSeek-V4-Pro (Preview) as the search model and DeepSeek-V4-Flash (Preview) as the execution model. Further implementation details are provided in Appendix A.3.

4.2 EXPERIMENTAL RESULTS

Main Results. Table 1 shows that MARS achieves the highest test performance on all five benchmarks while using fewer execution tokens than the strongest-performing baseline on each task. Across the five benchmarks, MARS improves test performance by an average of 13.5 points while reducing inference token consumption by an average of 77.3%, relative to the highest-scoring baseline on each benchmark.

Table 1: Comparison of test performance and inference token usage across five benchmarks. E2H-Codeforces is evaluated by pass rate, DROP and HotpotQA by F1, and GPQA-Diamond and Logic-Q by accuracy. All performance metrics are reported on a 0–100 scale. Tokens denotes the average number of input and output tokens consumed across all LLM calls per test instance.

Method	**E2H-Codeforces**		**DROP**		**GPQA**		**HotpotQA**		**Logic-Q**	
	Pass	**Tokens**	**F1**	**Tokens**	**Acc.**	**Tokens**	**F1**	**Tokens**	**Acc.**	**Tokens**
CoT	52.6	2052	74.3	481	75.3	1796	75.6	1539	40.4	3506
Self-Consistency	65.4	15160	74.4	3291	76.3	8947	78.5	9647	43.4	31923
Reflexion	70.2	20973	72.6	1181	74.1	5344	75.0	3477	42.4	7853
LLM Debate	71.8	28050	74.0	4145	76.7	26203	78.0	11555	38.0	54587
Step-back	61.0	5143	74.2	1166	73.1	3751	75.9	3316	42.2	7221
Quality-Diversity	63.8	13474	74.7	3229	76.9	10512	77.5	8476	39.8	20475
Dynamic Roles	58.8	3858	73.1	891	73.7	1887	75.7	2983	36.0	5748
ADAS	79.6	87275	79.7	10119	77.5	28653	78.2	14353	48.8	15204
AFlow	71.6	19342	86.8	1023	74.3	1982	77.5	13739	48.2	6077
MaAS	57.6	12439	53.6	2964	40.4	6826	64.9	9063	32.4	6929
MARS (Ours)	**87.0**	8932	**91.9**	620	**79.3**	2651	**80.7**	3205	**99.8**	0



(a) E2H-Codeforces **(b) GPQA**

Figure 3: Best-so-far test performance of MARS across generations on (a) E2H-Codeforces and (b) GPQA. Horizontal dashed lines indicate the test performance of ADAS, AFlow, and MaAS. Stars mark the first generation at which MARS surpasses all three baselines.

Search Trajectory. To examine search progress, we track the best test performance observed up to each generation. Figure 3 illustrates the trajectories on E2H-Codeforces and GPQA. Across all five benchmarks, MARS first exceeds the test performance of ADAS, AFlow, and MaAS at Generation 1 on E2H-Codeforces and DROP, and at Generation 2 on GPQA, HotpotQA, and Logic-Q.

Trade-off between Test Performance and Inference Tokens. Figure 4 examines the test performance and inference token usage of MARS candidates generated throughout the search. Among the evaluated candidates, 8/10 on E2H-Codeforces, 8/10 on DROP, 1/10 on GPQA, and 3/10 on HotpotQA achieve both higher test performance and lower inference token usage than all three search baselines. The same criterion is met by 3/5 candidates on Logic-Q. These results show that MARS discovers multiple candidates with favorable performance–token trade-offs on several benchmarks.

4.3 Discovered Agent Designs

Figure 5(a) illustrates the evolution of the Logic-Q agent from a single LLM-based reasoning solver to a fully deterministic agent. By Generation 5, MARS discovers a fully deterministic agent that achieves 99.8% test accuracy with zero LLM inference tokens. A detailed account of the evolution process is provided in Appendix A.6. Figure 5(b) presents the best discovered E2H-Codeforces agent, which integrates candidate generation, sample-test execution, execution-feedback-driven revision, and bounded revision control into an iterative code-refinement process.

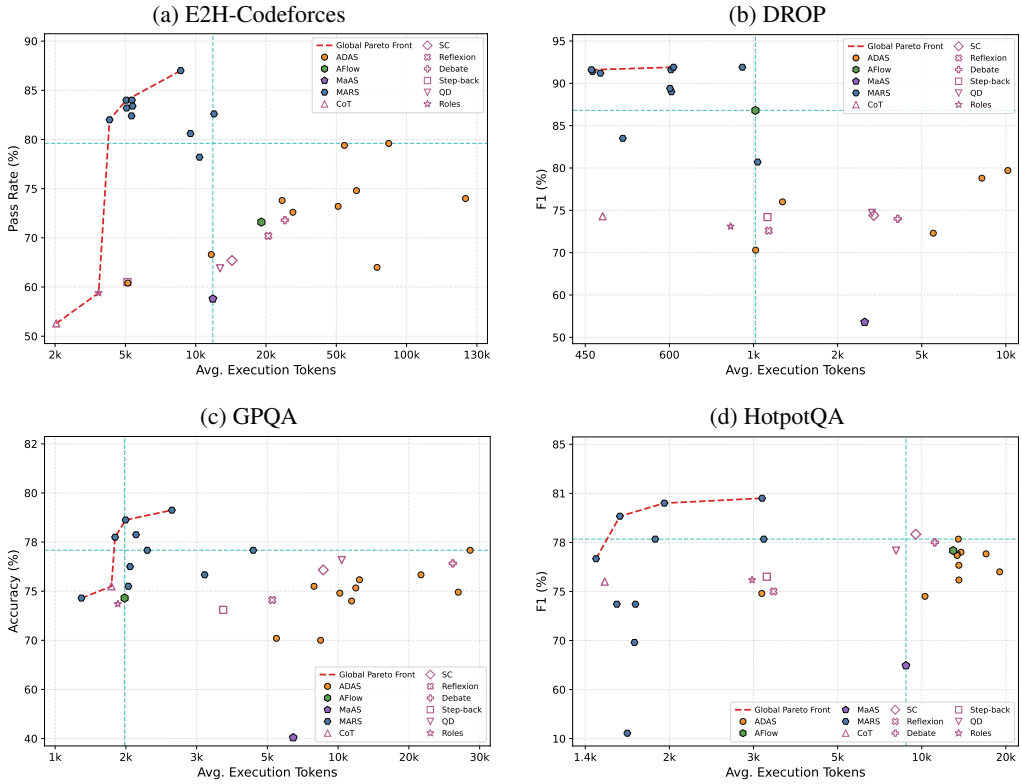
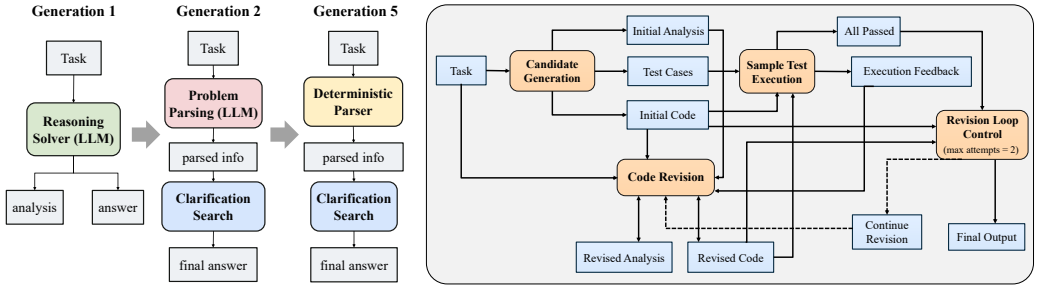


Figure 4: Test performance and average inference token usage of MARS candidates across four benchmarks. For each benchmark, the horizontal dashed line marks the highest test score among the ADAS best candidate, the AFlow best workflow, and the MaAS final workflow, while the vertical dashed line marks the lowest average inference token usage among these same three baselines. The two thresholds are determined independently and may correspond to different baselines. MARS candidates in the upper-left region strictly outperform all three baselines on both metrics.



(a) Evolution of the Logic-Q agent. (b) Best discovered E2H-Codeforces agent.

Figure 5: **Discovered agent designs on Logic-Q and E2H-Codeforces.** (a) Evolution of the Logic-Q agent from LLM-based reasoning to deterministic parsing. (b) Architecture of the best discovered E2H-Codeforces agent.

5 Related Work

5.1 Automated Agent Design and Search

Automated agent design differs in the search spaces explored and the strategies used to generate candidates. Structured approaches incorporate human-designed priors into agent architectures. GPTSwarm optimizes agent graphs, DyLAN adapts agent teams, and AgentPrune prunes communication topologies (Zhuge et al., 2024; Liu et al., 2024; Zhang et al., 2025b). AgentSquare recombines

modular components, MASS jointly optimizes prompts and workflows, and MaAS explores query-dependent architectures through an agentic supernet (Shang et al., 2025; Zhou et al., 2026; Zhang et al., 2025a). Archon combines inference-time techniques and language models under computational budgets (Saad-Falcon et al., 2025). These approaches constrain the search space and aim to improve search efficiency through human-designed priors.

Such priors make search more tractable but constrain the range of discoverable designs. Other approaches automate the construction and evolution of agent workflows (Li et al., 2024; Yuan et al., 2025; Abuzakuk et al., 2026). Among code-based methods, ADAS uses a meta-agent to generate new candidates based on an archive of previous discoveries (Hu et al., 2025), whereas AFlow applies Monte Carlo tree search to code-represented workflows, using execution feedback and tree-structured experience to guide subsequent modifications (Zhang et al., 2025c). MARS likewise searches over executable agent designs but establishes an explicit correspondence between agent code and a mechanism-level semantic representation to guide failure attribution and redesign without prescribing a fixed library of executable mechanisms.

Beyond their search spaces, existing methods also differ in whether they search for reusable agent designs or adapt computation to individual queries. MaAS samples query-dependent architectures, FlowReasoner generates agentic systems for individual queries, FlowBank retrieves workflows from a precomputed portfolio, and MAS^2 combines query-specific generation with runtime rectification (Zhang et al., 2025a; Gao et al., 2025; Yuan et al., 2026; Wang et al., 2026a). These methods explore how agent computation can be selected or constructed for individual queries. In contrast, MARS uses evidence from previously evaluated candidates to guide iterative agent redesign during search.

5.2 Feedback-Guided Agent Optimization

A growing body of work uses feedback beyond scalar evaluation scores to optimize prompts and compound AI systems (Lee et al., 2025). ProTeGi and PromptAgent use natural-language feedback to guide prompt optimization, OPRO uses previous solutions and their scores to inform new proposals, and Reflexion converts task feedback into linguistic reflections (Pryzant et al., 2023; Wang et al., 2024b; Yang et al., 2024; Shinn et al., 2023). Beyond individual prompts, DSPy provides abstractions for multi-stage language-model programs, while MIPRO jointly optimizes instructions and demonstrations across modules (Khattab et al., 2024; Opsahl-Ong et al., 2024). TextGrad propagates textual feedback through compound systems, Trace uses execution traces and rich feedback for system optimization, and GEPA reflects on execution trajectories to propose targeted textual updates (Yüksekgönül et al., 2025; Cheng et al., 2024; Agrawal et al., 2026).

Execution feedback also guides agent workflow search in AFlow, ScoreFlow, and FlowReasoner (Zhang et al., 2025c; Wang et al., 2025; Gao et al., 2025). For failure attribution, counterfactual approaches estimate the importance of individual agents, while STRACE identifies representative failure patterns and performs causal localization within execution trajectories to identify root-cause modules (Chen et al., 2025; Chang et al., 2026). These studies demonstrate the value of execution evidence for optimization and failure localization. MARS addresses a complementary problem in open-ended agent search by connecting mechanism-level semantics with executable code, grounding failure attribution in observed execution, and using the resulting diagnoses to guide redesign of mechanism composition, instantiation, and interaction.

6 Conclusion

We introduced MARS, which establishes an explicit correspondence between executable agent programs and a mechanism-level semantic representation, aligning agent design, failure attribution, and redesign at a common granularity. Grounded in executable code and observed execution, MARS uses mechanism-level diagnoses to guide targeted redesign during search. Across five tasks, MARS discovers agents that outperform the strongest automated-design baseline on each task while consuming fewer inference-time LLM tokens, surpassing the best results of all compared search baselines within the first two generations. Analysis of the discovered agents reveals diverse designs, ranging from iterative LLM-based refinement to fully deterministic pipelines, illustrating how mechanism-aware redesign can guide changes to both computational structure and implementation.

AI USE STATEMENT

Generative AI tools, including ChatGPT and Codex, were used to provide feedback on the research methodology and experimental design, assist with the implementation of the proposed method, and translate manuscript content into English. Additionally, these tools were used to assist with general code development and debugging, summarizing and analyzing existing literature, information retrieval, and identifying relevant research papers. Generative AI tools were also consulted for suggestions on figure layout and color schemes. The authors take full responsibility for the final content of this work.

REPRODUCIBILITY STATEMENT

We provide code in an anonymous repository: `https://anonymous.4open.science/r/repo-57EC/`. Additional implementation details and experimental settings are provided in the Appendix.

REFERENCES

Sami Abuzakuk, Anne-Marie Kermarrec, Rishi Sharma, Rasmus Moorits Veski, and Martijn de Vos. Optimizing agentic workflows using meta-tools. arXiv preprint arXiv:2601.22037, 2026. URL https://arxiv.org/abs/2601.22037.

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, Alex Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. GEPA: Reflective prompt evolution can outperform reinforcement learning. In C. Vondrick, B. Hariharan, C. Raffel, L. Pinto, D. Yang, and A. Faust (eds.), *International Conference on Learning Representations*, volume 2026, pp. 8479–8565, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/0e9e708b6f48e14fd0ac29e167413f76-Paper-Conference.pdf.

Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, 1997. doi: 10.1145/258077.258078. URL https://doi.org/10.1145/258077.258078.

Soufiane Amini, Yassine Benajiba, Cesare Bernardis, Paul Cayet, Hassan Chafi, Abderrahim Fathan, Louis Faucon, Damien Hilloulin, Sungpack Hong, Ingo Kossyk, Tirthankar Lahiri, Tran Minh Son Le, Rhicheek Patra, Sujith Ravi, Jonas Schweizer, Jyotika Singh, Shailender Singh, Weiyi Sun, Kartik Talamadupula, and Jerry Xu. Open agent specification: Enabling cross-framework comparison of AI agents. In *Proceedings of the ACM Conference on AI and Agentic Systems, CAIS 2026, San Jose, CA, USA, May 26-29, 2026*, pp. 404–418. ACM, 2026. doi: 10.1145/3786335.3813130. URL https://doi.org/10.1145/3786335.3813130.

Ying Chang, Jiahang Xu, Xuan Feng, Chenyuan Yang, Peng Cheng, and Yuqing Yang. From noisy traces to root causes: Structural trajectory analysis and causal extraction for agent optimization. *arXiv preprint arXiv:2607.07702*, 2026. URL https://arxiv.org/abs/2607.07702.

Jianming Chen, Yawen Wang, Junjie Wang, Xiaofei Xie, Jun Hu, Qing Wang, and Fanjiang Xu. Understanding individual agent importance in multi-agent system via counterfactual reasoning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 39(15):15785–15794, 2025. doi: 10.1609/aaai.v39i15.33733. URL https://doi.org/10.1609/aaai.v39i15.33733.

Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chi-Min Chan, Heyang Yu, Yaxi Lu, Yi-Hsin Hung, Chen Qian, Yujia Qin, Xin Cong, Ruobing Xie, Zhiyuan Liu, Maosong Sun, and Jie Zhou. AgentVerse: Facilitating multi-agent collaboration and exploring emergent behaviors. In B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun (eds.), *International Conference on Learning Representations*, volume 2024, pp. 20094–20136, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/578e65cdee35d00c708d4c64bce32971-Paper-Conference.pdf.

Ching-An Cheng, Allen Nie, and Adith Swaminathan. Trace is the next AutoDiff: Generative optimization with rich feedback, execution traces, and LLMs. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 71596–71642. Curran Associates, Inc., 2024. doi: 10.52202/079017-2287. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/83ba7056bce2c3c3c27e17397cf3e1f0-Paper-Conference.pdf.

Johan de Kleer and Brian C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1): 97–130, 1987. doi: 10.1016/0004-3702(87)90063-4. URL https://doi.org/10.1016/0004-3702(87)90063-4.

DeepSeek-AI. DeepSeek-V4: Towards highly efficient million-token context intelligence. arXiv preprint arXiv:2606.19348, 2026. URL https://arxiv.org/abs/2606.19348.

Mucong Ding, Chenghao Deng, Jocelyn Choo, Zichu Wu, Aakriti Agrawal, Avi Schwarzschild, Tianyi Zhou, Tom Goldstein, John Langford, Anima Anandkumar, and Furong Huang. Easy2Hard-Bench: Standardized difficulty labels for profiling LLM performance and generalization. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 44323–44365. Curran Associates, Inc., 2024. doi: 10.52202/079017-1407. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/4e6f223052759665139 90f53cec908e0-Paper-Datasets_and_Benchmarks_Track.pdf.

Yilun Du, Shuang Li, Antonio Torralba, Joshua B. Tenenbaum, and Igor Mordatch. Improving factuality and reasoning in language models through multiagent debate. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 11733–11763. PMLR, 21–27 Jul 2024. URL https://proceedings.mlr.press/v235/du24e.html.

Dheeru Dua, Yizhong Wang, Pradeep Dasigi, Gabriel Stanovsky, Sameer Singh, and Matt Gardner. DROP: A reading comprehension benchmark requiring discrete reasoning over paragraphs. In Jill Burstein, Christy Doran, and Thamar Solorio (eds.), *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 2368–2378, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1246. URL https://aclanthology.org/N19-1246/.

Hongcheng Gao, Yue Liu, Yufei He, Longxu Dou, Chao Du, Zhijie Deng, Bryan Hooi, Min Lin, and Tianyu Pang. FlowReasoner: Reinforcing query-level meta-agents. arXiv preprint arXiv:2504.15257, 2025. URL https://arxiv.org/abs/2504.15257.

David Garlan and Mary Shaw. An introduction to software architecture. Technical Report CMU-CS-94-166, Carnegie Mellon University, January 1994.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta programming for a multi-agent collaborative framework. In B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun (eds.), *International Conference on Learning Representations*, volume 2024, pp. 23247–23275, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/6507b115562bb0a305f1958ccc87355a-Paper-Conference.pdf.

Xinyi Hou, Shenao Wang, Yanjie Zhao, and Haoyu Wang. When agents do not stop: Uncovering infinite agentic loops in LLM agents. *arXiv preprint arXiv:2607.01641*, 2026. URL https://arxiv.org/abs/2607.01641.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 21344–21377, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/36b7acf6f6010652b3f2a433774a66fe-Paper-Conference.pdf.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan A, Saiful Haq, Ashutosh Sharma, Thomas Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. DSPy: Compiling declarative language model calls into state-of-the-art pipelines. In B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun (eds.), *International Conference on Learning Representations*, volume 2024, pp. 54928–54958, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/f1cf02ce09757f57c3b93c0db83181e0-Paper-Conference.pdf.

Yu-Ang Lee, Guan-Ting Yi, Mei-Yi Liu, Jui-Chao Lu, Guan-Bo Yang, and Yun-Nung Chen. Compound AI systems optimization: A survey of methods, challenges, and future directions. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (eds.), *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 28760–28775, Suzhou, China, November 2025. Association for Computational Linguistics. ISBN 979-8-89176-332-6. doi: 10.18653/v1/2025.emnlp-main.1463. URL https://aclanthology.org/2025.emnlp-main.1463/.

Belinda Z. Li, Been Kim, and Zi Wang. QuestBench: Can LLMs ask the right question to acquire information in reasoning tasks? In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, pp. 149610–149659. Curran Associates, Inc., 2025. doi: 10.52202/085713-4503. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/c42c8d51556fabb4b57fc86d3d3d0d09-Paper-Datasets_and_Benchmarks_Track.pdf. Datasets and Benchmarks Track.

Zelong Li, Shuyuan Xu, Kai Mei, Wenyue Hua, Balaji Rama, Om Raheja, Hao Wang, He Zhu, and Yongfeng Zhang. AutoFlow: Automated workflow generation for large language model agents. arXiv preprint arXiv:2407.12821, 2024. URL https://arxiv.org/abs/2407.12821.

Zijun Liu, Yanzhe Zhang, Peng Li, Yang Liu, and Diyi Yang. A dynamic LLM-powered agent network for task-oriented agent collaboration. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=XII0Wp1XA9.

Cong Lu, Shengran Hu, and Jeff Clune. Intelligent Go-Explore: Standing on the shoulders of giant foundation models. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 21276–21301, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/369a30aac2765950865efd318cef7f76-Paper-Conference.pdf.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 46534–46594. Curran Associates, Inc., 2023. doi: 10.52202/075280-2019. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/91edff07232fb1b55a505a9e9f6c0ff3-Paper-Conference.pdf.

Krista Opsahl-Ong, Michael J Ryan, Josh Purtell, David Broman, Christopher Potts, Matei Zaharia, and Omar Khattab. Optimizing instructions and demonstrations for multi-stage language model programs. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (eds.), *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 9340–9366, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.525. URL https://aclanthology.org/2024.emnlp-main.525/.

Joon Sung Park, Joseph C. O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative agents: Interactive simulacra of human behavior. In Sean Follmer, Jeff Han, Jürgen Steimle, and Nathalie Henry Riche (eds.), *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology, UIST 2023, San Francisco, CA, USA, 29 October 2023-1 November 2023*, pp. 2:1–2:22. ACM, 2023. doi: 10.1145/3586183.3606763. URL https://doi.org/10.1145/3586183.3606763.

Reid Pryzant, Dan Iter, Jerry Li, Yin Lee, Chenguang Zhu, and Michael Zeng. Automatic prompt optimization with "gradient descent" and beam search. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 7957–7968, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.494. URL https://aclanthology.org/2023.emnlp-main.494/.

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R. Bowman. GPQA: A graduate-level google-proof q&a benchmark. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=Ti67584b98.

Raymond Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1): 57–95, 1987. doi: 10.1016/0004-3702(87)90062-2. URL https://doi.org/10.1016/0004-3702(87)90062-2.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024. doi: 10.1038/S41586-023-06924-6. URL https://doi.org/10.1038/s41586-023-06924-6.

Jon Saad-Falcon, Adrian Gamarra Lafuente, Shlok Natarajan, Nahum Maru, Hristo Todorov, Etash Kumar Guha, E. Kelly Buchanan, Mayee F Chen, Neel Guha, Christopher Re, and Azalia Mirhoseini. An architecture search framework for inference-time techniques. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 52475–52507. PMLR, 13–19 Jul 2025. URL https://proceedings.mlr.press/v267/saad-falcon25a.html.

Yu Shang, Yu Li, Keyu Zhao, Likai Ma, Jiahe Liu, Fengli Xu, and Yong Li. AgentSquare: Automatic LLM agent search in modular design space. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 3841–3865, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/0ae94013da7cd459402fd77874e09ee3-Paper-Conference.pdf.

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. HuggingGPT: Solving AI tasks with ChatGPT and its friends in Hugging Face. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 38154–38180. Curran Associates, Inc., 2023. doi: 10.52202/075280-1657. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/77c33e6a367922d003ff102ffb92b658-Paper-Conference.pdf.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 8634–8652. Curran Associates, Inc., 2023. doi: 10.52202/075280-0377. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/1b44b878bb782e6954cd888628510e90-Paper-Conference.pdf.

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. *Transactions on Machine Learning Research*, 2024a. ISSN 2835-8856. URL https://openreview.net/forum?id=ehfRiF0R3a.

Kun Wang, Guibin Zhang, ManKit Ye, Xinyu Deng, Dongxia Wang, Xiaobin Hu, Jinyang Guo, Yang Liu, and Yufei Guo. MAS^2: Self-generative, self-configuring, self-rectifying multi-agent systems. In *The Fourteenth International Conference on Learning Representations*, 2026a. URL https://openreview.net/forum?id=qumy27hMDY.

Pengcheng Wang, Jerry Huang, Jiarui Yao, Rui Pan, Peizhi Niu, Yaowenqi Liu, Ruida Wang, Renhao Lu, Yuwei Guo, and Tong Zhang. AgentSPEX: An agent SPecification and EXecution language. arXiv preprint arXiv:2604.13346, 2026b. URL https://arxiv.org/abs/2604.13346.

Xinyuan Wang, Chenxi Li, Zhen Wang, Fan Bai, Haotian Luo, Jiayou Zhang, Nebojsa Jojic, Eric Xing, and Zhiting Hu. PromptAgent: Strategic planning with language models enables expert-level prompt optimization. In B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun (eds.), *International Conference on Learning Representations*, volume 2024, pp. 23967–24001, 2024b. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/686a3f32067838c8dbb68da6e9e3cf69-Paper-Conference.pdf.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=1PL1NIMMrw.

Yinjie Wang, Ling Yang, Guohao Li, Mengdi Wang, and Bryon Aragam. ScoreFlow: Mastering LLM agent workflows via score-based preference optimization. *arXiv preprint arXiv:2502.04306*, 2025. URL https://arxiv.org/abs/2502.04306.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems*, volume 35, pp. 24824–24837. Curran Associates, Inc., 2022. doi: 10.52202/068431-1800. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/9d5609613524ecf4f15af0f7b31abca4-Paper-Conference.pdf.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. AutoGen: Enabling next-gen LLM applications via multi-agent conversations. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=BAakY1hNKS.

Benfeng Xu, An Yang, Junyang Lin, Quan Wang, Chang Zhou, Yongdong Zhang, and Zhendong Mao. ExpertPrompting: Instructing large language models to be distinguished experts. arXiv preprint arXiv:2305.14688, 2025. URL https://arxiv.org/abs/2305.14688v2.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun (eds.), *International Conference on Learning Representations*, volume 2024, pp. 12028–12068, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/3339f19c5fcee3ad74502947a32be9e6-Paper-Conference.pdf.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In Ellen Riloff, David Chiang, Julia Hockenmaier, and Jun'ichi Tsujii (eds.), *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pp. 2369–2380, Brussels, Belgium, October-November 2018. Association for Computational Linguistics. doi: 10.18653/v1/D18-1259. URL https://aclanthology.org/D18-1259/.

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*, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

Lingzhi Yuan, Chenghao Deng, Fangxu Yu, Souradip Chakraborty, Mohammad Rostami, and Furong Huang. FlowBank: Query-adaptive agentic workflows optimization through precompute-and-reuse. *arXiv preprint arXiv:2606.11290*, 2026. URL https://arxiv.org/abs/2606.11290.

Siyu Yuan, Kaitao Song, Jiangjie Chen, Xu Tan, Dongsheng Li, and Deqing Yang. EvoAgent: Towards automatic multi-agent generation via evolutionary algorithms. In Luis Chiruzzo, Alan Ritter, and Lu Wang (eds.), *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 6192–6217, Albuquerque, New Mexico, April 2025. Association for Computational Linguistics. ISBN 979-8-89176-189-6. doi: 10.18653/v1/2025.naacl-long.315. URL https://aclanthology.org/2025.naacl-long.315/.

Mert Yüksekgönül, Federico Bianchi, Joseph Boen, Sheng Liu, Pan Lu, Zhi Huang, Carlos Guestrin, and James Zou. Optimizing generative AI by backpropagating language model feedback. *Nature*, 639(8055):609–616, 2025. doi: 10.1038/S41586-025-08661-4. URL https://doi.org/10.1038/s41586-025-08661-4.

Guibin Zhang, Luyang Niu, Junfeng Fang, Kun Wang, Lei Bai, and Xiang Wang. Multi-agent architecture search via agentic supernet. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 75834–75852. PMLR, 13–19 Jul 2025a. URL https://proceedings.mlr.press/v267/zhang25bi.html.

Guibin Zhang, Yanwei Yue, Zhixun Li, Sukwon Yun, Guancheng Wan, Kun Wang, Dawei Cheng, Jeffrey Yu, and Tianlong Chen. Cut the crap: An economical communication pipeline for LLM-based multi-agent systems. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 75389–75428, 2025b. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/bbc461518c59a2a8d64e70e2c38c4a0e-Paper-Conference.pdf.

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 Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 34040–34077, 2025c. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/5492ecbce4439401798dcd2c90be94cd-Paper-Conference.pdf.

Huaixiu Steven Zheng, Swaroop Mishra, Xinyun Chen, Heng-Tze Cheng, Ed H. Chi, Quoc V Le, and Denny Zhou. Take a step back: Evoking reasoning via abstraction in large language models. In B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun (eds.), *International Conference on Learning Representations*, volume 2024, pp. 20279–20316, 2024a. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/592da1445a51e54a3987958b5831948f-Paper-Conference.pdf.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. SGLang: Efficient execution of structured language model programs. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 62557–62583. Curran Associates, Inc., 2024b. doi: 10.52202/079017-2000. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/724be4472168f31ba1c9ac630f15dec8-Paper-Conference.pdf.

Han Zhou, Xingchen Wan, Ruoxi Sun, Hamid Palangi, Shariq Iqbal, Ivan Vulić, Anna Korhonen, and Sercan Ö. Arık. Multi-agent design: Optimizing agents with better prompts and topologies. In *International Conference on Learning Representations*, 2026. URL https://research.google/pubs/multi-agent-design-optimizing-agents-with-better-prompts-and-topologies/.

Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. GPTSwarm: Language agents as optimizable graphs. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 62743–62767. PMLR, 21–27 Jul 2024. URL https://proceedings.mlr.press/v235/zhuge24a.html.

A Appendix

A.1 Search Space and Mechanisms

Executable agent representation. Following ADAS (Hu et al., 2025), we represent each candidate agent as an executable Python program. Each agent implements a unified entry point, `forward(self, taskInfo)`, within which its complete behavior is specified, including LLM calls, deterministic computation, and interactions among different components. We adapt the code-based agent representation and runtime interface of ADAS to our framework, while introducing our own mechanism-level search space and search procedure.

The executable program constitutes the actual implementation of an agent. In addition, we maintain a structured mechanism representation, referred to as the mechanism IR, which explicitly describes the mechanisms instantiated in the program. This representation provides a structured view of the generated code and enables the search process to reason about agent designs at the mechanism level while retaining the flexibility of executable code.

Illustrative runtime structure of an executable agent

```
def forward(self, taskInfo):
    # Mechanism 1: candidate generation
    solver = LLMAgentBase(
        output_fields=["analysis", "answer"],
        agent_name="Solver",
        temperature=0.7,
    )
    analysis, answer = solver(
        [taskInfo],
        "Analyze the problem and produce a candidate answer."
    )

    # Mechanism 2: evaluation and revision
    reviewer = LLMAgentBase(
        output_fields=["feedback", "revised_answer"],
        agent_name="Reviewer",
        temperature=0.5,
    )
    feedback, revised_answer = reviewer(
        [taskInfo, analysis, answer],
        "Review the candidate answer and revise it if necessary."
    )

    return revised_answer
```

The example above is only illustrative. In our framework, the generated code may instantiate different numbers and types of mechanisms, use different prompts and output fields, and realize substantially different information-flow and control-flow structures. Each LLM-based mechanism instance corresponds to a single LLM invocation, while deterministic mechanisms are realized through programmatic computations or actions.

Mechanism-level search space. Rather than restricting agent construction to a fixed library of predefined operators, we organize the design space around *mechanisms*. A mechanism denotes a functional component of an agent program, while its concrete implementation is determined through code generation. Consistent with the semantic representation defined in Section 2, the resulting search space spans three complementary dimensions: mechanism composition, mechanism instantiation, and mechanism interaction.

Mechanism composition. The search can modify which mechanism families are present in an agent and the number of mechanism instances belonging to each family. Mechanisms may be added, removed, or replaced as the agent evolves. Multiple mechanism instances may belong to the same family while serving different purposes or using different concrete implementations. Therefore, a mechanism family specifies a functional role in the overall agent design rather than a fixed reusable operator.

Mechanism instantiation. Given a selected mechanism, the search further determines how it is concretely implemented. We distinguish two implementation types: LLM-based and deterministic.

An LLM-based mechanism instance is realized through a single LLM invocation. Its concrete instantiation specifies the prompt, temperature, and structured output fields of that invocation. The mechanism-level inputs and outputs describe the information consumed and produced by the mechanism, while the structured output fields specify the outputs returned by its LLM invocation. These properties are reflected in the generated executable code.

A deterministic mechanism is implemented directly through programmatically specified computations or actions. Importantly, deterministic mechanisms are not selected from a predefined operator library and are not restricted to a fixed collection of allowable operations. Their implementation is open-ended and may contain arbitrary deterministic computation expressible in the generated agent program. For example, a deterministic mechanism may implement conditional logic, parsing or transformation, aggregation or selection rules, branching, loops, retry logic, stopping criteria, fallback behavior, programmatic checking, or other computations. These examples are illustrative rather than exhaustive and do not define the boundary of the deterministic search space.

Mechanism interaction. The search also determines how instantiated mechanisms interact to form the complete agent program. These interactions encompass both information flow and control flow. Information flow describes how mechanisms consume upstream information and produce results for downstream computation. The mechanism IR records their declared inputs and outputs, while the executable code specifies how information is actually transferred and used. Control flow is expressed through the conditional logic and control structures in the executable code, including execution order, activation conditions, branching, iteration, stopping, fallback, and return paths. Consequently, agents containing similar mechanism families can exhibit substantially different behavior through different mechanism instantiations and interactions. The search space therefore covers not only which functional components are used and how they are implemented, but also how information and control are organized across the complete agent program.

Mechanism representation. To expose the structure of code-based agent designs to the search process, each mechanism instantiated in an agent is accompanied by a structured entry in the mechanism IR. Each entry contains the fields `mechanism_id`, `name`, `family`, `implementation_type`, `inputs`, `outputs`, and `instantiation`.

The mechanism ID uniquely identifies a mechanism within an agent, while the name describes its concrete role in the current design. The family field assigns the mechanism to one of the functional families defined below, and the implementation type specifies whether it is realized through a single LLM invocation or deterministic computation. The inputs and outputs summarize the information consumed and produced by the mechanism. Finally, the instantiation records the concrete realization of the mechanism.

For an LLM-based mechanism, the instantiation specifies the prompt, temperature, and structured output fields of its single LLM invocation. These structured output fields are distinguished from the mechanism-level inputs and outputs, which describe the information consumed and produced by the mechanism within the overall agent design. For a deterministic mechanism, the instantiation describes the corresponding programmatic computation or action without constraining it to a predefined set of operations.

The structured representation is required to remain consistent with the executable program. In particular, mechanisms recorded in the mechanism IR should correspond to computations actually instantiated in the generated code, and their declared inputs, outputs, and LLM-call configurations should reflect their concrete implementation. The executable code remains the authoritative specification of agent behavior, while the mechanism IR provides a structured abstraction that supports mechanism-level search and analysis.

Mechanism families. We define eight mechanism families that capture recurring functional roles in agentic problem solving. These families characterize *what role* a mechanism serves within an agent rather than prescribing *how* that role must be implemented. The concrete realization of a mechanism remains open-ended and is determined through generated executable code.

- **Intermediate Analysis.** Produces intermediate analysis for subsequent mechanisms, such as task interpretation, decomposition, planning, problem representation, or other intermediate reasoning, without directly producing a final-answer candidate.

- **Candidate Generation.** Generates one or more candidate solutions that may serve as, or contribute to, the final answer.
- **Procedure Construction.** Constructs an executable intermediate procedure or artifact, such as a program, algorithm, query, command, test procedure, or action sequence, to support solving, computation, checking, or verification.
- **Execution and Operation.** Executes a constructed procedure, invokes an available operation or execution environment, and makes the resulting observation or execution result available for subsequent computation.
- **Evaluation and Revision.** Evaluates, critiques, verifies, repairs, refines, modifies, or rewrites an existing candidate, intermediate result, or execution result.
- **Aggregation and Decision.** Jointly uses multiple upstream results or signals to compare, select, synthesize, reconcile, aggregate, or otherwise derive a downstream result.
- **Routing and Control.** Controls the execution of other mechanisms, including ordering, activation, branching, iteration, stopping, fallback behavior, retry behavior, and final return paths.
- **Output Processing and Normalization.** Extracts, parses, converts, or adjusts an already generated result so that it conforms to the required task or evaluator output format.

These mechanism families deliberately operate at a higher level of abstraction than conventional predefined operators. For example, **Aggregation and Decision** does not prescribe a particular voting or selection algorithm; it may be instantiated as majority voting, LLM-based judgment, synthesis of complementary candidates, deterministic selection, or another generated implementation. Similarly, **Evaluation and Revision** identifies the functional role of evaluating or improving an existing result without imposing a fixed review or revision procedure. This separation between functional roles and concrete implementations allows the search process to explore diverse executable realizations while retaining a structured description of the resulting agent designs.

A.2 Mechanism-Level Diagnosis and Attribution

Diagnosis objective. After evaluating each generated agent, we perform a mechanism-level diagnosis to identify the concrete problems that limit its validation performance. The diagnosis is based on the executable agent code, its mechanism IR, the evaluation summary, task-specific evaluation context, and representative case-level execution summaries. Rather than directly specifying how the next agent should be constructed, the diagnoser characterizes problems in the current design. These diagnoses are recorded in Search History and subsequently serve as evidence for the search process when constructing new agent designs.

The diagnosis focuses on both individual mechanisms and their interactions. In particular, it distinguishes failures originating within a mechanism from failures caused by information or control flow across mechanisms. This distinction enables the search process to identify not only whether a component behaves poorly, but also whether otherwise useful signals are lost, overwritten, misused, or incorrectly routed as they propagate through the agent.

Diagnosis granularity. A problem associated with the behavior of a single mechanism is attached directly to that mechanism. Such problems may concern, for example, whether its implementation matches the declared design, whether it is activated appropriately, or whether the signals it produces are sufficiently useful.

Cross-mechanism problems are represented separately as interaction diagnoses between a source mechanism and a target mechanism. Interaction diagnosis is used when the failure arises from the transfer, preservation, combination, routing, or downstream use of information rather than from the quality of one mechanism in isolation. For example, if a candidate generation mechanism produces a correct candidate but a subsequent decision mechanism fails to select it, the problem is attributed to the interaction between the two mechanisms rather than to candidate generation itself.

Problem taxonomy. We use six diagnosis categories. Each diagnosed problem is assigned to one of the following types.

- **Output or Runtime Reliability.** This category captures failures in program execution or in producing a valid final output. Examples include runtime exceptions, interface errors, incorrect return types, missing or unparsable fields, parsing failures, invalid data structures, incorrectly unpacked LLM outputs, incompatible intermediate object types, control-flow paths that fail to return an answer, and final outputs that violate the evaluator contract. These failures are diagnosed before higher-level behavioral problems when they are present, since they can prevent the intended agent design from being executed or evaluated reliably.
- **Mechanism Implementation.** This category is used when the executable code does not correctly implement the mechanism design declared in the mechanism IR. Examples include a declared trigger, rule, or control-flow structure that is not realized in the code, or declared mechanism inputs, outputs, and connections that do not match the actual information flow. This diagnosis therefore concerns a discrepancy between the represented mechanism design and its executable realization, rather than the intrinsic quality of the mechanism itself.
- **Mechanism Activation.** This category captures cases in which a mechanism is correctly implemented but is activated inappropriately. Examples include unsuitable trigger conditions, execution timing, iteration counts, stopping conditions, or early-return behavior, as well as mechanisms that execute when they are unnecessary or fail to execute when they are needed. If a mechanism is not activated on an observed execution path, its output quality is not diagnosed from that case; only evidence concerning its activation or routing can be inferred.
- **Mechanism Quality.** This category applies when a mechanism executes as intended but fails to produce sufficiently correct, useful, or beneficial signals. Such failures may include incorrect reasoning, judgment, or calculation; highly redundant candidate generation; unhelpful intermediate analysis; feedback that fails to identify the relevant error; verification that fails to detect an incorrect result; or revision that fails to improve, or even degrades, an existing result. More generally, this category concerns the quality of the computation performed within an individual mechanism, given that the mechanism was properly implemented and activated.
- **Mechanism Interaction.** This category captures failures that arise across mechanisms after a useful upstream signal has already been produced. Examples include useful information being lost during transfer, an available correct candidate not being selected, a better upstream result being overwritten by a worse downstream result, a downstream mechanism failing to use information it received, an aggregation stage excluding a useful candidate, or routing logic preventing a useful signal from reaching the final answer path. It also includes cases where information is incorrectly transformed or compressed during transfer, or where a downstream mechanism does not receive upstream context that is necessary for its intended function.

  A useful distinction is that Mechanism Quality asks whether an individual mechanism produced a sufficiently useful signal, whereas Mechanism Interaction asks whether an already useful signal was appropriately propagated, preserved, and utilized by the rest of the agent.
- **Insufficient Evidence.** This category is used when the available evaluation and execution evidence does not support a reliable and sufficiently specific mechanism-level attribution. Rather than forcing a diagnosis from ambiguous observations, the diagnoser explicitly records insufficient evidence. Mechanisms for which no supported problem is observed may also retain an empty problem set.

Attribution principles. To avoid conflating downstream consequences with the origin of a failure, diagnosis follows the actual execution path and prioritizes the earliest evidence-supported divergence from the desired behavior. When a case contains multiple failures, we distinguish the mechanism that first introduces an incorrect signal from later mechanisms that fail to detect, correct, preserve, select, or route that signal. For example, if candidate generation first produces an incorrect answer and a downstream verifier subsequently fails to identify the error, the original error remains attributable to candidate generation, while the verification failure may be diagnosed separately. Conversely, if a correct or otherwise useful signal has already been produced upstream but is subsequently lost or replaced, the downstream failure is treated as an interaction problem rather than

retroactively attributed to the upstream mechanism. This distinction allows diagnosis to separate error introduction from error propagation and failed recovery.

The executable code and observed execution behavior take precedence over the declared mechanism representation when the two disagree. If the mechanism IR declares an information connection that is absent from the actual code, the discrepancy is diagnosed as a mechanism implementation problem. If the information is correctly passed downstream but is subsequently ignored, overwritten, or misused, the failure is instead diagnosed as a mechanism interaction problem.

Diagnosis is also separated from redesign. The mere absence of an additional mechanism is not treated as a diagnosis in itself. For example, rather than diagnosing that an agent "lacks a verifier," the diagnosis identifies the concrete observed failure, such as an incorrect candidate remaining undetected. Determining whether the subsequent search should add a verifier, modify an existing mechanism, change its interaction structure, or adopt another solution is left to the search stage.

Diagnosis representation. The resulting diagnosis augments the mechanism IR with a set of problems associated with individual mechanisms. Each problem contains a problem type from the taxonomy above together with a natural-language description of the supporting evidence. Problems involving multiple mechanisms are stored separately as interaction diagnoses, each specifying a source mechanism, a target mechanism, and the corresponding diagnosed problems. A generation-level agent summary additionally provides a concise description of the principal performance bottlenecks observed for the evaluated design.

Accordingly, diagnosis preserves the distinction between the structure of the agent and the evidence obtained from its execution: the mechanism IR describes what mechanisms are instantiated, mechanism-level problems describe failures localized within those mechanisms, and interaction diagnoses describe failures arising from their composition. This structured diagnosis is then retained in Search History as evidence for subsequent mechanism-aware search.

A.3 Experimental Setup

Dataset Splits. E2H-Codeforces and Logic-Q each use 100 validation instances and 500 test instances. HotpotQA uses 200 validation instances and 800 test instances, while DROP uses 128 validation instances and 800 test instances. GPQA-Diamond contains 198 questions, with 32 used for validation and 166 for testing. Each GPQA-Diamond question is evaluated over five independent runs, resulting in 160 validation runs and 830 test runs, whereas all other benchmarks use a single run per instance.

Implementation Details. We use DeepSeek-V4-Pro (Preview) as the search model and DeepSeek-V4-Flash (Preview) as the execution model. The former runs with thinking enabled and `high` reasoning effort, while the latter runs with thinking disabled. MARS and ADAS run for at most 10 generations, AFlow for 10 optimization rounds, and MARS terminates early upon perfect validation performance. For MaAS, we use DeepSeek-V4-Pro (Preview) for TextGrad-based prompt optimization and DeepSeek-V4-Flash (Preview) for workflow execution, with the same reasoning configurations as above. Following its released implementation, MaAS runs for one optimization round with four training repetitions.

Dataset Construction. For Logic-Q, we construct the experimental splits from a held-out pool of 1,150 instances. Using deterministic stratified sampling with a fixed random seed of 42, we construct disjoint validation and test sets, leaving 550 instances unused. The sampling procedure is independent of the performance of any model or agent. We jointly stratify the data according to two dimensions: maximum reasoning depth and the number of accepted clarification answers. The maximum reasoning depth is grouped into four ranges: 1–2, 3–4, 5–6, and ≥ 7, which account for approximately 19.2%, 42.3%, 28.1%, and 10.3% of the original pool, respectively. The number of accepted clarification answers is grouped into 1, 2, 3–4, and ≥ 5, accounting for approximately 36.8%, 23.7%, 23.0%, and 16.6% of the original pool, respectively. The Cartesian product of these two dimensions yields 16 strata, and both the validation and test sets are sampled proportionally from these strata to preserve the composition of the original held-out pool.

Table 2: Search and optimization token consumption across five tasks

Task	ADAS	AFlow	MARS	MaAS (Sample)	MaAS (TextGrad)
E2H-Codeforces	776861	145299	1138021	5637857	3744
DROP	586300	66284	532040	1439718	3843
GPQA	477995	62014	502162	3090286	1308
HotpotQA	582090	51712	575296	7414973	1633
Logic-Q	566186	114291	395493	4870123	1325

For E2H-Codeforces, we construct the experimental splits from the 4,000 problems in the official evaluation split. We first remove problems that cannot be reliably evaluated through deterministic execution, including interactive problems, problems with non-unique outputs, constructive or witness-style outputs, and problems relying on floating-point or custom checkers. After filtering, 1,525 candidate problems remain.

We then stratify the remaining problems according to their rating quantiles. The four difficulty groups, Q1, Q2, Q3, and Q4, correspond to the intervals $[0, 0.25)$, $[0.25, 0.50)$, $[0.50, 0.75)$, and $[0.75, 1.00]$, respectively. Using a fixed random seed of 42, we construct mutually disjoint validation and test sets with the same difficulty composition: 10%, 20%, 50%, and 20% of the instances are sampled from Q1, Q2, Q3, and Q4, respectively. Accordingly, the validation set contains 10, 20, 50, and 20 problems from the four difficulty groups, while the test set contains 50, 100, 250, and 100 problems, respectively.

A.4 Search and Optimization Costs

Table 2 reports the cumulative token consumption during the search and optimization processes across five tasks. For ADAS, AFlow, and MARS, the reported values aggregate the recorded search-stage tokens across all available generations or rounds. For MaAS, we separately report the tokens consumed by training-time sampling and TextGrad optimization. All reported search and optimization components use DeepSeek-V4 Preview Pro with high thinking enabled, except for MaAS training-time sampling, which uses DeepSeek-V4 Preview Flash with thinking disabled. The reported values follow the token-accounting conventions of the respective implementations and do not represent identical end-to-end search costs.

A.5 Discovered Agent Designs

To complement the analysis of discovered agent designs in Section 4.3, we provide the concrete implementations of representative agents discovered by our search procedure. These implementations correspond directly to the architectures and evolutionary stages visualized in the main paper, and expose the discovered mechanisms, information flow, and control flow at the program level.

For E2H, we provide the complete implementation of the best discovered agent, which employs execution feedback to iteratively revise an initial candidate solution. For Logic-Q, we present representative agents from different generations along the search trajectory. The trajectory illustrates a transition from direct LLM-based reasoning to hybrid neuro-symbolic computation, and eventually to increasingly deterministic parsing and reasoning.

E2H Best Discovered Agent. The best discovered E2H agent first generates a candidate solution together with test cases, executes the generated program, and uses execution feedback to revise the solution when failures are detected. The revision process is repeated for a bounded number of attempts.

E2H Best Discovered Agent: Execution-Feedback Revision

```
def forward(self, taskInfo):
    import subprocess
    import sys
    import json
    import re
```

```
    # Step 1: Candidate generation
    solver = LLMAgentBase(
        output_fields=["analysis", "test_cases", "answer"],
        agent_name="Reasoning Solver",
        temperature=0.2,
    )
    analysis, test_cases, answer = solver(
        [taskInfo],
        "You are an expert programmer. Given a competitive "
        "programming problem, you must produce a correct Python 3 "
        "solution.\n\n"
        "Step 1: Restate the problem in your own words, clearly "
        "specifying the input, output, constraints, and any special "
        "conditions.\n"
        "Step 2: Identify and list all edge cases and tricky scenarios "
        "that could break a naive solution.\n"
        "Step 3: Design an efficient algorithm. Explain your reasoning "
        "step by step. Analyze the time and memory complexity and "
        "ensure it meets the constraints.\n"
        "Step 4: First, extract the sample test cases exactly as "
        "provided in the problem statement (input and expected output). "
        "Then, generate at least 3 additional diverse test cases, "
        "including edge cases. For each, provide the exact input and "
        "expected output. Output a JSON array of test case objects with "
        "'input' and 'output' keys.\n"
        "Step 5: For each test case, manually simulate your algorithm, "
        "showing the key steps and state changes, and verify that the "
        "output matches the expected output. If any discrepancy, revise "
        "the algorithm.\n"
        "Step 6: Write the complete Python program. The program must "
        "handle all input/output as specified.\n\n"
        "Output format:\n"
        "analysis: (your thorough analysis, including all steps above, "
        "the simulation trace, and complexity analysis)\n"
        "test_cases: (a JSON array of test case objects with 'input' "
        "and 'output' keys, starting with the sample test cases from "
        "the problem)\n"
        "answer: (the final Python code, without any extra explanation "
        "or markdown formatting)\n"
        "The answer field must contain only the executable Python code."
    )

    # Output processing and normalization
    def extract_code(text):
        pattern = r'```(?:python)?\s*\n?(.*?)```'
        match = re.search(pattern, text, re.DOTALL)
        if match:
            return match.group(1).strip()
        return text.strip()

    # Sample test execution
    def run_tests(code, test_cases):
        feedback = []
        all_passed = True

        for i, tc in enumerate(test_cases):
            inp = tc.get("input", "")
            expected = tc.get("output", "").strip()

            try:
                result = subprocess.run(
                    [sys.executable, "-c", code],
                    input=inp,
                    capture_output=True,
                    text=True,
                    timeout=10,
                )

                stdout = (
                    result.stdout.strip()
                    if result.stdout else ""
                )
                stderr = (
                    result.stderr.strip()
                    if result.stderr else ""
                )

                if result.returncode != 0 or stderr:
```

```
                    feedback.append(
                        f"Test {i+1}: FAILED with error\n"
                        f"input:\n{inp}\n"
                        f"stderr:\n{stderr}\n"
                    )
                    all_passed = False

                elif stdout != expected:
                    feedback.append(
                        f"Test {i+1}: FAILED\n"
                        f"input:\n{inp}\n"
                        f"expected:\n{expected}\n"
                        f"got:\n{stdout}\n"
                    )
                    all_passed = False

                else:
                    feedback.append(
                        f"Test {i+1}: PASSED"
                    )

            except subprocess.TimeoutExpired:
                feedback.append(
                    f"Test {i+1}: TIMEOUT\n"
                    f"input:\n{inp}\n"
                )
                all_passed = False

            except Exception as e:
                feedback.append(
                    f"Test {i+1}: EXECUTION ERROR: {e}\n"
                )
                all_passed = False

        return "\n".join(feedback), all_passed

    # Extract original code
    original_code = extract_code(answer.content)

    if not original_code:
        original_code = extract_code(analysis.content)

    # Parse test cases
    try:
        test_cases_list = json.loads(test_cases.content)

        if not isinstance(test_cases_list, list):
            test_cases_list = []

    except Exception:
        test_cases_list = []

    # If no valid test cases, return original code
    if not test_cases_list:
        return original_code

    # Step 2: Execution on sample tests
    exec_feedback, all_passed = run_tests(
        original_code,
        test_cases_list
    )

    # Step 3: Revision loop
    max_attempts = 2
    revision_attempt = 0

    current_code = original_code
    current_analysis = analysis.content

    revision_solver = LLMAgentBase(
        output_fields=["analysis", "answer"],
        agent_name="Revision Solver",
        temperature=0.2,
    )

    while revision_attempt < max_attempts and not all_passed:

        # Prepare execution feedback
        feedback_info = Info(
```

```
            "execution_feedback",
            "TestExecutor",
            exec_feedback,
            -1
        )

        code_info = Info(
            "original_code",
            "CodeExtractor",
            current_code,
            -1
        )

        analysis_info = Info(
            "original_analysis",
            "Analyzer",
            current_analysis,
            -1
        )

        # Code revision
        rev_analysis, rev_answer = revision_solver(
            [
                taskInfo,
                analysis_info,
                code_info,
                feedback_info
            ],
            "You are an expert programmer. A previous solution to a "
            "competitive programming problem failed on some sample "
            "test cases. Your task is to debug the code, identify the "
            "errors, and produce a corrected Python 3 solution. Below "
            "are the problem statement, the original analysis, the "
            "original code, and the execution feedback. Analyze the "
            "feedback to understand the failures, then fix the code. "
            "Ensure the revised code passes all given sample test cases "
            "and is correct for the problem.\n\n"
            "Output format:\n"
            "analysis: (your debugging analysis and explanation of fixes)\n"
            "answer: (the corrected Python code, without any extra "
            "explanation or markdown formatting)\n"
            "The answer field must contain only the executable Python code."
        )

        # Extract revised code
        revised_code = extract_code(rev_answer.content)

        if not revised_code:
            revised_code = extract_code(
                rev_analysis.content
            )

        if not revised_code:
            current_code = current_code
            break

        current_code = revised_code
        current_analysis = rev_analysis.content

        # Re-test revised code
        exec_feedback, all_passed = run_tests(
            current_code,
            test_cases_list
        )

        revision_attempt += 1

    return current_code
```

Logic-Q Agent Evolution. The Logic-Q search trajectory exhibits a qualitatively different form of evolution. The initial agent relies entirely on an LLM to perform problem interpretation, logical deduction, and clarification selection. Subsequent generations progressively externalize these operations into explicit programmatic mechanisms.

Logic-Q Generation 1: Reasoning Solver

```
def forward(self, taskInfo):
    # Mechanism 1: candidate generation
    solver = LLMAgentBase(
        output_fields=["analysis", "answer"],
        agent_name="Reasoning Solver",
        temperature=0.0,
    )

    analysis, answer = solver(
        [taskInfo],
        "You are given a logical reasoning problem about Alice's "
        "attributes. The input contains: Rules (implications), Known "
        "true facts, Known false facts, Cannot ask facts, and a Goal. "
        "Your task is to determine whether the Goal's truth value can "
        "be deduced from the given information. If it can be deduced, "
        "output exactly 'End questioning'. If it cannot be deduced, "
        "you need to find a clarification attribute (not already known "
        "true/false, not in the cannot-ask list, not the Goal itself) "
        "such that both: if the clarification attribute is true, the "
        "Goal becomes determinable; if the clarification attribute is "
        "false, the Goal also becomes determinable. Then output exactly: "
        "Question: Is Alice <attribute>? Follow these steps:\n"
        "1. List all attributes mentioned in rules and facts.\n"
        "2. Identify which attributes are known true, known false, "
        "cannot be asked, and the goal.\n"
        "3. Attempt to determine the goal using the rules and known "
        "facts. Use logical deduction (forward/backward chaining). "
        "If the goal's truth value is determined, output "
        "'End questioning'.\n"
        "4. If not determined, consider each attribute that is not "
        "known, not cannot-ask, and not the goal. For each candidate, "
        "assume it is true and see if the goal becomes determinable; "
        "then assume it is false and see if the goal becomes "
        "determinable. If both assumptions lead to determinability, "
        "that attribute is a valid clarification question.\n"
        "5. Choose one such attribute (if multiple, pick any). "
        "Output your reasoning in the analysis field, and the final "
        "answer line in the answer field. The answer field must "
        "contain only the required output line."
    )

    return answer
```

Generation 2 introduces a hybrid design. An LLM is retained for structured extraction of the natural-language problem, while logical deduction and clarification selection are moved into deterministic programmatic components.

Logic-Q Generation 2: Structured Extraction and Deterministic Deduction

```
def forward(self, taskInfo):
    import json

    # Mechanism 1: problem parsing
    parser = LLMAgentBase(
        output_fields=["json_output"],
        agent_name="Parser",
        temperature=0.0,
    )

    json_output, = parser(
        [taskInfo],
        "You will be given a logical reasoning problem about Alice. "
        "Extract the following information into a JSON object with "
        "keys: 'known_true' (list of attribute strings), "
        "'known_false' (list), 'cannot_ask' (list), 'goal' (string), "
        "'rules' (list of objects, each with 'antecedents' "
        "(list of strings) and 'consequent' (string)). "
        "For example, rule 'If Alice is curious and Alice is calm, "
        "then Alice is thoughtful.' becomes "
        "{'antecedents': ['curious', 'calm'], "
        "'consequent': 'thoughtful'}. "
        "Output only the JSON, no extra text."
    )
```

```
    # Parse structured output
    try:
        data = json.loads(json_output.content)
    except Exception:
        # Conservative fallback if parsing fails
        return "End questioning"

    known_true = set(data.get("known_true", []))
    known_false = set(data.get("known_false", []))
    cannot_ask = set(data.get("cannot_ask", []))
    goal = data.get("goal", "")
    rules = data.get("rules", [])

    # Build rule representations
    rule_list = []
    for r in rules:
        antecedents = r.get("antecedents", [])
        consequent = r.get("consequent", "")
        if antecedents and consequent:
            rule_list.append((antecedents, consequent))

    # Mechanism 2: deterministic solver
    def propagate(known_true, known_false):
        value = {}

        for attr in known_true:
            value[attr] = True

        for attr in known_false:
            value[attr] = False

        changed = True

        while changed:
            changed = False

            for antecedents, consequent in rule_list:
                # Forward deduction
                if all(
                    value.get(a) is True
                    for a in antecedents
                ):
                    if value.get(consequent) is not True:
                        value[consequent] = True
                        changed = True

                # Contrapositive deduction
                if value.get(consequent) is False:
                    unknown = [
                        a for a in antecedents
                        if value.get(a) is None
                    ]
                    true_ones = [
                        a for a in antecedents
                        if value.get(a) is True
                    ]

                    if (
                        len(unknown) == 1
                        and len(true_ones)
                        == len(antecedents) - 1
                    ):
                        attr = unknown[0]

                        if value.get(attr) is not False:
                            value[attr] = False
                            changed = True

        return value.get(goal)

    # Check whether goal is already determinable
    initial_result = propagate(
        known_true,
        known_false
    )

    if initial_result is not None:
        return "End questioning"
```

```
    # Collect candidate clarification attributes
    all_attrs = set()

    for antecedents, consequent in rule_list:
        all_attrs.update(antecedents)
        all_attrs.add(consequent)

    all_attrs.update(known_true)
    all_attrs.update(known_false)
    all_attrs.update(cannot_ask)
    all_attrs.add(goal)

    candidates = (
        all_attrs
        - known_true
        - known_false
        - cannot_ask
        - {goal}
    )

    # Search for a valid clarification question
    for q in candidates:
        # Assume q is true
        new_true = known_true | {q}
        new_false = known_false

        if propagate(new_true, new_false) is not None:
            # Assume q is false
            new_true = known_true
            new_false = known_false | {q}

            if propagate(new_true, new_false) is not None:
                return f"Question: Is Alice {q}?"

    # Fallback if no valid clarification is found
    return "End questioning"
```

By Generation 5, the search further removes the LLM-based parser. The structured task format is parsed deterministically, while the logical propagation and clarification search remain explicit programmatic procedures. This results in an agent whose reasoning pipeline is largely deterministic.

Logic-Q Generation 5: Deterministic Parsing and Propagation

```
def forward(self, taskInfo):
    import re

    content = taskInfo.content

    known_true = set()
    known_false = set()
    cannot_ask = set()
    goal = None
    rules = []

    # Mechanism 1: deterministic parser
    lines = content.split("\n")
    current_section = None

    for line in lines:
        line = line.strip()

        if not line:
            continue

        if line.startswith("Rules:"):
            current_section = "rules"
            continue

        elif line.startswith("Known true facts:"):
            current_section = "known_true"
            continue

        elif line.startswith("Known false facts:"):
            current_section = "known_false"
            continue
```

```
        elif line.startswith("Cannot ask facts:"):
            current_section = "cannot_ask"
            continue

        elif line.startswith("Goal:"):
            current_section = "goal"
            continue

        if current_section == "rules":
            attrs = re.findall(
                r"Alice is ([\w-]+)",
                line
            )

            if len(attrs) >= 2:
                antecedents = attrs[:-1]
                consequent = attrs[-1]
                rules.append(
                    (antecedents, consequent)
                )

        elif current_section == "known_true":
            if "None" in line:
                continue

            match = re.search(
                r"Alice is ([\w-]+)",
                line
            )

            if match:
                known_true.add(match.group(1))

        elif current_section == "known_false":
            if "None" in line:
                continue

            match = re.search(
                r"Alice is ([\w-]+)",
                line
            )

            if match:
                known_false.add(match.group(1))

        elif current_section == "cannot_ask":
            if "None" in line:
                continue

            match = re.search(
                r"Do not ask whether Alice is ([\w-]+)",
                line
            )

            if match:
                cannot_ask.add(match.group(1))

        elif current_section == "goal":
            match = re.search(
                r"Is Alice ([\w-]+)\?",
                line
            )

            if match:
                goal = match.group(1)

    if goal is None:
        return "End questioning"

    # Mechanism 2: deterministic propagation
    def propagate(
        known_true_set,
        known_false_set
    ):
        value = {}

        for attr in known_true_set:
            value[attr] = True
```

```
        for attr in known_false_set:
            value[attr] = False

        changed = True

        while changed:
            changed = False

            for antecedents, consequent in rules:
                # Forward deduction
                if all(
                    value.get(a) is True
                    for a in antecedents
                ):
                    if value.get(consequent) is not True:
                        value[consequent] = True
                        changed = True

                # Contrapositive deduction
                if value.get(consequent) is False:
                    unknown = [
                        a for a in antecedents
                        if value.get(a) is None
                    ]

                    true_ones = [
                        a for a in antecedents
                        if value.get(a) is True
                    ]

                    if (
                        len(unknown) == 1
                        and len(true_ones)
                        == len(antecedents) - 1
                    ):
                        attr = unknown[0]

                        if value.get(attr) is not False:
                            value[attr] = False
                            changed = True

        return value.get(goal)

    # Check initial determinability
    initial_result = propagate(
        known_true,
        known_false
    )

    if initial_result is not None:
        return "End questioning"

    # Construct clarification candidates
    all_attrs = set()

    for antecedents, consequent in rules:
        all_attrs.update(antecedents)
        all_attrs.add(consequent)

    all_attrs.update(known_true)
    all_attrs.update(known_false)
    all_attrs.update(cannot_ask)
    all_attrs.add(goal)

    candidates = (
        all_attrs
        - known_true
        - known_false
        - cannot_ask
        - {goal}
    )

    # Search for a valid clarification attribute
    for q in candidates:
        # Assume q is true
        new_true = known_true | {q}
        new_false = known_false
```

Table 3: Validation and test performance across five Logic-Q generations. Accuracy is reported as a percentage, and Tokens denotes the average number of LLM inference tokens per instance. Test accuracy is reported as the bootstrap median.

Generation	Validation		Test	
	Acc.	Tokens	Acc.	Tokens
Gen 1	51.0	6102	45.4	6339
Gen 2	89.0	2360	80.0	2455
Gen 3	56.0	0	61.8	0
Gen 4	84.0	0	91.2	0
Gen 5	**100.0**	**0**	**99.8**	**0**

```
        if propagate(new_true, new_false) is not None:
            # Assume q is false
            new_true = known_true
            new_false = known_false | {q}

            if propagate(new_true, new_false) is not None:
                return f"Question: Is Alice {q}?"

    return "End questioning"
```

Overall, the Logic-Q trajectory shows that the search procedure does not merely modify prompts within a fixed agent template. Instead, it changes the computational structure of the agent itself. In particular, the discovered designs progressively replace LLM-based reasoning with explicit deterministic components when the structure of the task makes such computation possible. This evolution contrasts with the E2H agent, where execution feedback and iterative LLM-based revision remain useful for handling open-ended program synthesis.

A.6 Case Studies

Logic-Q: Diagnosis-Guided Agent Evolution. Logic-Q, introduced in QuestBench (Li et al., 2025), evaluates an agent's ability to identify the missing information required to resolve an underspecified logical reasoning problem. Each task provides a set of logical rules, known facts, prohibited clarification attributes, and a target proposition. The agent must determine whether the target can already be inferred or identify a clarification attribute whose truth value, under either possible answer, makes the target determinable. The original QuestBench paper also investigates symbolic search methods, including breadth-first search (BFS), demonstrating the effectiveness of deterministic reasoning on this task.

We use Logic-Q to illustrate how MARS progressively modifies agent structure and mechanism implementation in response to failure diagnosis. Figure 5(a) presents the main evolution from Generation 1 to Generation 2 and Generation 5. Table 3 summarizes the performance and LLM inference token consumption of all five generations.

Identifying the reasoning bottleneck. Generation 1 employs a single LLM-based reasoning mechanism that produces an explicit analysis and a final answer. However, it achieves only 51% validation accuracy. Failure diagnosis reveals that the LLM frequently fails to evaluate clarification candidates correctly. It either prematurely concludes that no valid clarification attribute exists or selects an attribute that does not satisfy the required two-branch determinability condition. In some failure cases, the generated analysis even acknowledges that a selected candidate fails the condition, yet the agent still outputs it as the final answer.

To address this reasoning bottleneck, MARS decomposes the original reasoning mechanism into two specialized mechanisms in Generation 2. An LLM-based Problem Parsing mechanism extracts logical rules, known facts, prohibited clarification attributes, and the target into a structured representation. A Deterministic Solver then performs logical propagation and evaluates candidate clarification attributes under both possible truth assignments. By separating information extraction from logical deduction, this modification increases validation accuracy from 51% to 89% while substantially reducing LLM inference tokens.

Diagnosing the parsing bottleneck. Despite this improvement, the new architecture exposes another failure mode. Mechanism-level diagnosis of Generation 2 identifies an output reliability problem in the LLM-based parser. The parser sometimes generates malformed JSON or appends extraneous text, causing structured-output parsing to fail. In these cases, the fallback returns `End questioning`, even when a clarification question is required.

The following excerpt presents the diagnosed Mechanism IR of Generation 2. The diagnosis identifies a reliability problem in the Problem Parsing mechanism, while no problem is identified for the Deterministic Solver. This localizes the observed failure to a specific mechanism and provides a concrete target for subsequent modification.

Logic-Q Generation 2: Mechanism-Level Diagnosis


```
{
    "mechanism_ir": [
        {
            "mechanism_id": "m1",
            "name": "Problem Parsing",
            "family": "intermediate_analysis",
            "implementation_type": "llm",
            "inputs": ["taskInfo"],
            "outputs": ["parsed_info"],
            "problems": [
                {
                    "problem_type":
                        "output_or_runtime_reliability",
                    "problem":
                        "Malformed JSON or extraneous text causes parsing failures and an
     incorrect 'End questioning' fallback."
                }
            ]
        },
        {
            "mechanism_id": "m2",
            "name": "Deterministic Solver",
            "family": "candidate_generation",
            "implementation_type": "deterministic",
            "inputs": ["parsed_info"],
            "outputs": ["final_answer"],
            "problems": []
        }
    ],
    "interaction_diagnosis": []
}
```


Exploring deterministic implementations. Guided by this diagnosis, MARS explores replacing the LLM-based parser with deterministic code. Generations 3 and 4 represent alternative designs explored from Generation 2 rather than consecutive modifications of one another. Both eliminate LLM inference but introduce implementation-related failures.

In Generation 3, an incorrect substring offset causes the parser to extract prohibited clarification attributes incorrectly, allowing invalid candidates to enter the search. Generation 4 repairs parts of the parsing implementation but still mishandles hyphenated attribute names in logical rules. Diagnoses of both generations also report limitations in their logical propagation implementations. Their validation accuracies of 56% and 84%, respectively, illustrate that replacing an LLM mechanism with deterministic code does not automatically guarantee improved performance.

Discovering the fully deterministic agent. Generation 5 replaces the remaining LLM-based parser with a deterministic, section-based implementation that directly extracts logical rules, known true and false facts, prohibited attributes, and the target from the task input. Its outputs are passed to a deterministic Clarification Search mechanism, which performs logical propagation and evaluates candidate attributes under both possible truth assignments.

Importantly, the core clarification-search procedure follows the same basic approach introduced in Generation 2. The principal modification is therefore the replacement of the remaining LLM-based parsing mechanism and the rewiring of the downstream mechanism to consume its deterministic outputs, rather than the introduction of an entirely new search algorithm.

The resulting Generation 5 agent achieves 100% validation accuracy and a median test accuracy of 99.8%, with zero LLM inference tokens. This case illustrates how MARS uses successive mechanism-level diagnoses to identify different bottlenecks and guide targeted design modifications. The search first separates logical deduction from LLM-based reasoning and subsequently replaces the remaining LLM parser with deterministic computation, demonstrating its ability to modify both workflow structure and mechanism implementation types.

E2H-Codeforces: Discovering an Execution-Guided Revision Workflow. E2H-Codeforces provides a complementary case in which MARS discovers a feedback-driven workflow for competitive programming. As shown in Figure 5(b), the final agent integrates candidate generation, deterministic test execution, and iterative code revision, rather than relying solely on LLM-based self-verification.

Diagnosing the verification bottleneck. In the earlier generations, MARS progressively strengthens the candidate generation mechanism by introducing explicit algorithmic reasoning, edge-case analysis, and self-verification. However, failure diagnosis reveals that the LLM still produces incorrect algorithms and buggy implementations, even when its analysis appears plausible. For example, Generation 9 exhibits incorrect dynamic programming formulations, off-by-one errors, and misplaced implementation logic that remain undetected by prompt-based self-verification. These failures motivate the introduction of an external execution signal.

An alternative branch, Generation 8, further reveals a mechanism-interaction problem. Although it introduces test execution and iterative revision, its test cases are generated by the same LLM that produces the candidate solution. Consequently, the generated tests may share the candidate's flawed assumptions and fail to expose its errors. The execution mechanism can therefore report that all tests pass, prematurely terminating the revision loop and returning incorrect code. This diagnosis highlights the importance of the information passed between candidate generation, execution, and control mechanisms.

Discovering the execution-guided workflow. Building on Generation 9, MARS introduces an execution-feedback revision workflow in Generation 10. The candidate generation mechanism is instructed to extract the sample tests provided in the problem statement before generating additional edge cases. A deterministic execution mechanism then runs the extracted code against the resulting test cases and produces execution feedback, including incorrect outputs, execution errors, and timeouts.

When execution detects a failure, an LLM-based revision mechanism receives the original task, current analysis, current code, and execution feedback to generate a revised solution. The revised code is subsequently extracted and re-executed. A deterministic control mechanism repeats this process for at most two revision attempts, terminating early if all tests pass. The discovered workflow thus establishes an explicit dependency between execution feedback and subsequent code revision.

Generation 10 achieves 87% validation accuracy, compared with 83% for its parent, Generation 9. Nevertheless, passing the available tests does not guarantee correctness on hidden cases, and the final diagnosis still identifies limitations in test coverage and revision quality. This case illustrates how MARS can identify failures arising from both individual mechanisms and their interactions, and use these diagnoses to construct conditional, execution-guided agent workflows.