A Vocabulary for Multi-Agent Automated Research Systems

Bardiya Akhbari
Amazon AGI
bardiyaa@amazon.com

Abstract

We introduce a vocabulary for automated research systems built from one or more agents to make their design choices easier to describe and compare. The vocabulary specifies who the agents are, which operations the harness exposes, who may invoke them, how agents communicate, what state is visible within and across runs, how the next action is chosen, how a run begins, and how outputs are scored. A trajectory records one run from the input task to the returned artifact. Because agents, operations, and initialization may be stochastic, repeated runs on the same task induce a distribution over trajectories rather than a single behavior.

The vocabulary assigns each structural design question, such as when agents should communicate, gain or lose a capability, or carry information across runs, to a distinct coordinate that can be varied on its own. It also makes the evaluator a component of the system, since reported gains depend on how closely the proxy score matches true quality. That separation also splits the vague complaint that these systems lack *taste* into two failures with different fixes. Generative taste is the rate at which a system proposes novel trajectories before any score is observed, and evaluative taste is the gap between the proxy score and the quality it should match. We instantiate the vocabulary on recent autoresearch systems to illustrate that it covers designs that differ widely in structure.

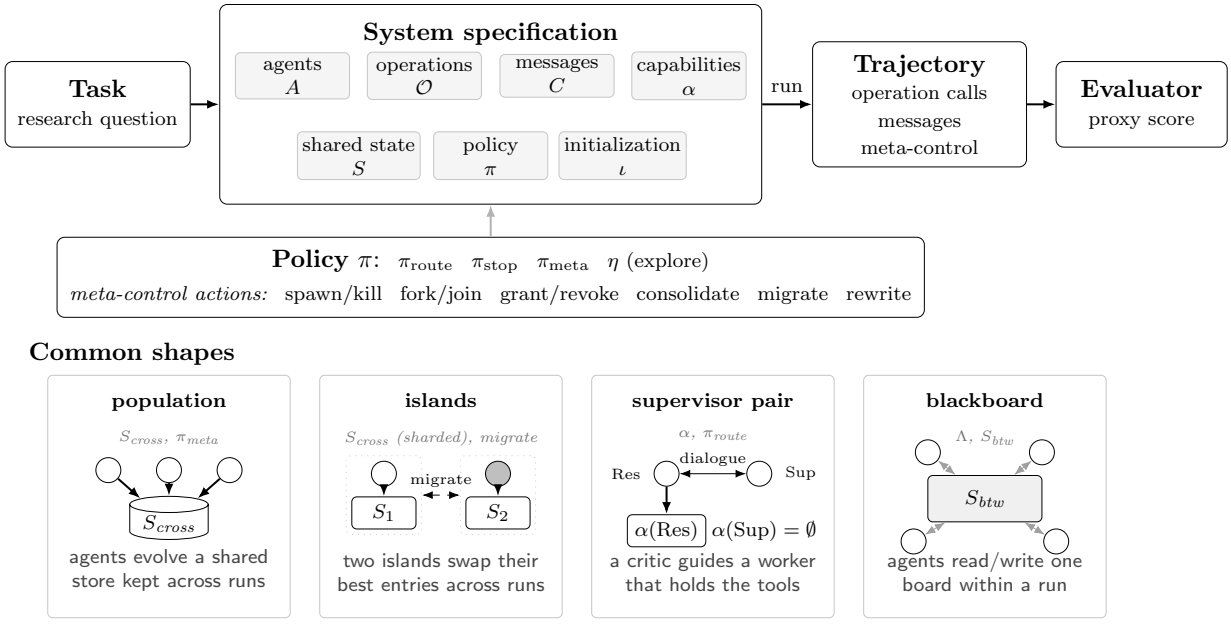

Figure 1: A multi-agent system is a set of components. A research task enters on the left. The system produces a trajectory, the full record of which agent acted, what it did, and what came back at each step. The evaluator grades that trajectory and returns a score. The lower row shows common shapes, each a different setting of those same components.

1 Introduction

Recent multi-agent large language model (LLM) systems automate research end-to-end: they collect and read prior work, propose ideas, write code, run experiments, interpret results, and iterate. Several of these systems frame their automation as explicit search over research trajectories. AIRA$_2$ searches a population of candidate experiments via evolutionary selection [1]. Glia searches a space of inference-system designs via a supervisor–researcher agent pair optimizing simulated latency [2]. The Automated Alignment Researcher (AAR) searches for alignment algorithms by hill-climbing on a scalar reward signal [3]. These systems share a common shape, where each defines an objective, a search space of research artifacts, and an iterative strategy for navigating it. Lacking a common language to describe them, we formalize the shared structure as a framework for automated research.

These systems differ along many axes simultaneously. One differs from another in communication topology, another in cross-run memory, another in initialization, another only in the evaluator. A claim that a "multi-agent system" or an "autoresearcher" is better is ambiguous until we know which axis changed and why. A vocabulary should therefore decompose the system into coordinates fine enough to isolate each choice. Some decomposition decisions are non-obvious. We separate the operation universe (what can be done) from the capability assignment (who may do it), so that least-privilege and asymmetric designs are native rather than described by side comment. We include the evaluator as a primary component rather than treating it as external infrastructure, because reported gains depend on how closely the proxy score correlates with the true quality – and that gap is often large [2,3]. We include initialization explicitly because the same architecture seeded differently can search entirely different regions of trajectory space.

Therefore, we formalize a multi-agent automated research system as the tuple

$$\mathcal{M} = \langle A, \mathcal{O}, C, \alpha, S, \pi, \iota, e \rangle,$$

where A is the set of agents, $\mathcal{O}$ the operation universe (tools, skills, hooks, and other callable actions), $C = (\Lambda, \sigma)$ the communication structure (Λ is the communication space, and σ is the message protocol), $\alpha : A \to 2^{\mathcal{O}}$ the capability assignment, S the shared state, π the control policy, ι the initialization function, and e the evaluator. In plain terms, the tuple records who is in the system, what can be done, who may do it, how agents communicate, what shared state exists, what runs when, how the run starts, and how the result is graded. We have split the operation universe and capability assignment deliberately. In our notation, $\mathcal{O}$ contains the invocable operations available to the system, such as application programming interfaces (APIs), shell commands, scripts, simulators, retrieval calls, or reusable skills loaded by the harness; α is the permission structure over that universe. The tuple formalizes what recent practice calls the *harness*, the system layer around a base model that orchestrates its tool use, context, and control [4].

With this vocabulary a reader can identify which coordinate a comparison changed, and a designer can develop new techniques by varying one coordinate at a time. An improvement credited to "multi-agent system" may instead come from communication, initialization, cross-run memory, evaluator integrity, or capability assignment, and the tuple separates them into distinct components (Figure 1). Treating the evaluator as a coordinate also gives us a handle on *taste*, a notion left informal, which we split into whether the system proposes good candidates (generative taste) and whether its evaluator scores them faithfully (evaluative taste).

Our contribution is a language that describes these systems in common terms and identifies which coordinate each design changes. In the following sections, we define the problem space (Section 2), the system specification (Section 3), and the trajectory notation (Section 4). We then discuss optimization challenges (Section 5), apply the vocabulary to current autoresearch systems (Section 6), and close with open design questions (Section 7). For related work, see Appendix A.

2 Problem specification

We can only judge a system relative to a problem, which we write as the tuple

$$\mathcal{P} = \langle \mathcal{X}, \mathcal{Y}, \phi, q, \omega, B, \mathcal{D} \rangle. \tag{1}$$

A problem defines the tasks a system faces, the artifacts that count as solutions, and how their quality is measured. A task $x \in \mathcal{X}$ (the input space) might be a research question, a bug report, a clinical case, or a code-improvement target, and a solution $y \in \mathcal{Y}$ (the output space) is the corresponding artifact, such as a paper, a patch, a diagnosis, or an optimized program.

The *task feature map* $\phi : \mathcal{X} \to \mathbb{R}^d$, for some $d \geq 1$, maps each task to a vector of measurable properties. For instance, the task descriptions "write a data-loading script" and "discover a sorting algorithm and prove it" are both tasks $x \in \mathcal{X}$, but ϕ separates them by coordinates such as the number of subtasks (one vs. several) and the performance of a fixed reference agent on similar tasks. Grouping tasks by these coordinates lets us reason about whole families at once.

The *true quality* $q : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}^k$ is the ideal evaluation function. It returns the actual quality of y on x along $k \geq 1$ objective dimensions (e.g., $k = 3$ for correctness, novelty, and cost). The scalarization $\omega : \mathbb{R}^k \to \mathbb{R}$ combines the k scores into a single number, so solutions can be ranked. In practice, q is uncomputable, and the discrepancy between a computable evaluator and q is the proxy-quality gap. Each task has a hard *budget*, $B \in \mathbb{R}_{>0}$, measured in tokens, dollars, wall-clock time, and/or operation calls depending on the deployment context.

The *task distribution* $\mathcal{D}$ is a probability distribution over $\mathcal{X}$ (e.g., a benchmark suite of coding problems, where each task x is one problem to solve). In most settings, $\mathcal{D}$ is a fixed benchmark, and a contribution changes the system applied to it. AIRS-Bench is one such fixed $\mathcal{D}$, scoring research agents on 20 ML tasks with held-out test labels and a programmatic scorer [5]. Naming $\mathcal{D}$ as a component also allows us to describe the rarer case where the system changes $\mathcal{D}$ itself, generating its own problems rather than solving a fixed set.

Recursive self-improvement (RSI) systems provide a concrete instance of $\mathcal{P}$ [6,7]. A task $x \in \mathcal{X}$ is a complete AI system, its model weights together with the harness around them (e.g., Codex, Claude Code, or Open Code), and a solution $y \in \mathcal{Y}$ is a more capable successor. Some systems already run this loop, for example by rewriting their own harness code and keeping the changes that raise a benchmark score [8], or by applying a code-improvement program to that program itself [9]. The features $\phi(x)$ record the current system's capability level (e.g., benchmark accuracy or code acceptance rate). The true quality q measures how much better the successor is at producing its own successor, a quantity that can only be assessed over a long horizon. The budget B is the compute allowed for one improvement step. Lastly, because the output y becomes the next input x, RSI reshapes its own task distribution $\mathcal{D}$ at every step rather than solving a fixed set.

3 System specification

We define a *multi-agent system* as the eight-tuple

$$\mathcal{M} = \langle A, \mathcal{O}, C, \alpha, S, \pi, \iota, e \rangle. \tag{2}$$

A single system addresses many problems (Table 1). Because $\mathcal{P}$ and $\mathcal{M}$ are separate tuples, we can vary one and fix the other. We call a system general when its components stay fixed as $\mathcal{P}$ varies.

A *run* is one execution of the system on a task, from input to returned artifact, formalized as a trajectory. A subscript i indexes agents and a subscript t indexes time steps, so X_t is the time-t value of a component X otherwise written without a subscript. Most of $\mathcal{M}$ is fixed by the designer before the run, but the meta-control policy π_{meta} can reconfigure the system as it proceeds,

Component	Name	Definition
A	agents	Who is in the system, and what makes them differ?
$\mathcal{O}$	operations	Which tools, skills, and callable actions exist?
Λ	message edges	Who may message whom?
σ	message form	What schema does a message take?
α	capability	Who may invoke which operation, and when does that change?
S_{btw}	within-run state	What do agents share during one run?
S_{world}	external state	What environment do the agents act on?
S_{cross}	cross-run state	What carries from one run into the next?
π_{route}	routing	Who acts next, and what do they do?
π_{stop}	stopping	When does the run end?
π_{meta}	meta-control	When does the system change its own structure?
η	exploration	How stochastic are the choices?
ι	initialization	How does a run start, given the task?
e	evaluator	How is the trajectory scored? How far can that score drift from true quality q?

Table 1: Each component captures one design choice for a multi-agent autoresearch system.

changing the agent set A_t, the capabilities α_t, the communication space Λ_t, and the state S_t, and even rewriting the control policy π itself (Table 2). $\mathcal{O}_i = \alpha(a_i)$ is agent i's permitted operation set. Every other symbol is defined where it is introduced.

Identity (A). The agent set is $A = \{a_i\}_{i=1}^{n}$, with agent count $|A| = n$, where each agent is the tuple $a_i = (\theta_i, m_i, m_i^0, \rho_i)$. The backbone θ_i is the underlying model and its weights, typically a frozen large language model and possibly fine-tuned offline. The private memory m_i holds the per-agent context history, scratchpads, or learned representations that persist within the agent across its turns but are not visible to other agents, and m_i^0 is its value at $t = 0$. The role ρ_i specifies the agent's prompt, persona, or capability tier.

We define an agent by the private memory m_i that persists across its turns (wherever stored), which keeps $|A|$ a meaningful coordinate rather than a count of language-model calls. An entity without it – a stateless call with a fixed role, such as a one-shot verifier or scorer prompt – we treat as a *module* and place in $\mathcal{O}$ or in the mechanism of e rather than in A. For example, an *agent* is a critic with a running scratchpad; a *module* is the same critic invoked fresh.

Operation universe ($\mathcal{O}$). An *operation* is any "action" an agent can take. For example, an operation can be calling a tool or a skill. A *tool* is a single callable action with a typed signature, such as an API call or a shell command. A *skill* is a named reusable routine the harness loads and invokes, such as a `SKILL.md`-style prompt-and-code bundle or a learned policy. Operations also cover reading or writing files, querying retrieval, invoking a Model Context Protocol (MCP) server, and calling a stateless module. The operation universe $\mathcal{O}$ is the registry of such operations available to the system. Each operation $u \in \mathcal{O}$ has a typed signature and an effect on the shared state S (state outside any single agent). $\mathcal{O}$ specifies what could be done, not who may do it.

Communication (C). Communication has a *communication space* Λ and a *message protocol* σ, written as $C = (\Lambda, \sigma)$. The communication space Λ is the set of directed edges over agents into shared state. The protocol σ fixes the form of each message, with a schema that ranges from text to structured output to code artifacts. Λ fixes who may talk to whom, not who actually does on any given step. The realized communication on a trajectory is the set of send-message actions the routing policy π_{route} emits. Every such message must travel an edge that Λ permits. When agents communicate through a shared board rather than direct messages (e.g., a blackboard or a forum any agent reads and writes), the two components stay disjoint by their roles. Λ is the read/write access pattern over that board (which agent may post and which may consume), and the board's contents are part of the within-run state S_{btw} (defined below).

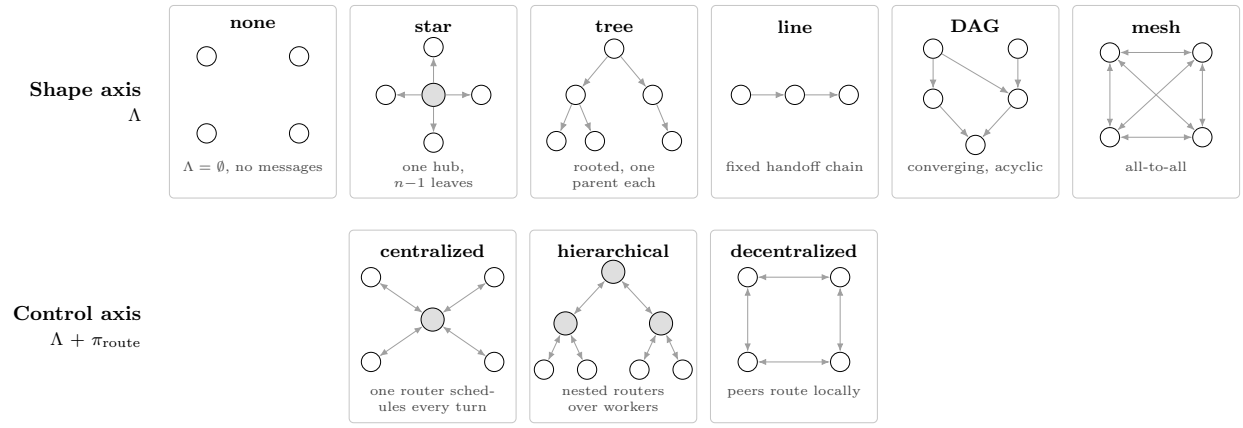

Figure 2: The communication space Λ has the shape axis which fixes which edges exist (top row), and the control axis which fixes who selects the active edge each turn (bottom row), from one router to nested routers to peers. The same edge set runs centralized, hierarchical, or decentralized depending on π_{route}, so topology shape and control are independent choices. In the control row, shaded nodes are routers that hold π_{route}, and their absence in the decentralized panel marks peers that route locally. Directed arrows are one-way edges; double arrows are bidirectional ones.

A single label like "hierarchical" or "star" hides two separate choices, so we split a topology into two axes. The shape axis (no edges, star, tree, line, directed acyclic graph, mesh) fixes which edges exist, a property of Λ alone. The control axis (centralized, hierarchical, or decentralized) fixes who selects the active edge each turn, a property of Λ together with π_{route} (Figure 2). The same edge set runs under any control scheme, so a topology claim that states only one axis leaves the other unstated. A static topology corresponds to Λ pinned to a single graph; a dynamic topology, to Λ as a strictly larger set from which π_{route} selects per-step edges. Phrasing communication this way turns "when should the agents communicate?" into a policy question over edges in Λ, not a question about choosing one static graph.

Capability assignment (α). The capability assignment $\alpha : A \to 2^{\mathcal{O}}$ specifies, for each agent, the subset of operations that agent is permitted to invoke. Asymmetric capability assignment (i.e., different agents with different permitted operation sets) is inequality of the $\mathcal{O}_i$. Glia's Supervisor, for example, has $\alpha(\text{Sup}) = \emptyset$, and that emptiness is the central design choice of the system. Just as Λ structures messages over the agent set A, α structures operations over A. A static capability assignment corresponds to α pinned at $t = 0$; a dynamic capability assignment corresponds to α mutated mid-run by the meta-control policy π_{meta} via $\textsc{grant}(i, u)$ and $\textsc{revoke}(i, u)$ actions, so that an agent's operation set $\mathcal{O}_{i,t} = \alpha_t(a_i)$ varies with time step t. This turns "when should an agent gain or lose a capability?" into a policy question over $\textsc{grant}/\textsc{revoke}$ actions.

Shared state (S). State outside any single agent's private memory is often combined together as "memory." We split it by scope into three kinds,

$$S = \langle S_{\text{btw}}, S_{\text{world}}, S_{\text{cross}} \rangle. \tag{3}$$

S_{btw} lives for a single run and is wiped when that run ends. It is shared between agents while the run is "in progress", such as a scratchpad or blackboard (a free-form workspace any agent can read and overwrite) or a message log (an append-only transcript every agent reads). S_{world} is the external state the agents operate on, distinct from their own memory, such as a simulator, a code repository, a database, or an environment. S_{cross} persists across runs, and takes three forms. A *skill*

Action	Effect	Modifies
SPAWN(ρ)	add an agent with role ρ	A, α, Λ
KILL(i)	retire agent i	A, α, Λ
FORK	branch into independent continuations	S_{btw}
JOIN($\cdot$)	merge branches by best-of, concatenation, or a coordinator	S_{btw}
GRANT(i, u)	add operation $u \in \mathcal{O}$ to agent i	α
REVOKE(i, u)	remove operation u from agent i	α
CONSOLIDATE($\cdot$)	write a run's result in the cross-run store, by appending a score or summarizing	S_{cross}
MIGRATE(s, s')	copy entries between two shards $s, s' \subseteq S_{\text{cross}}$	S_{cross}
REWRITE-POLICY(π')	replace the control policy with π' using a policy register held in S_{btw}	π

Table 2: Each meta-control action in Π_{meta} modifies a specific state component.

library accumulates reusable routines. A *population database* carries the pool of candidate solutions across generations in an evolutionary loop. A *distilled store* keeps a compressed or consolidated summary of past runs rather than the raw artifacts.

We separate two kinds of improvement with these scopes. Within a single run, agents refine through S_{btw} and the experiment output in S_{world}, so a system with $S_{\text{cross}} = \emptyset$ still improves, but each run starts from scratch and re-derives whatever the last run learned. Carrying improvement across runs is what S_{cross} adds. Evolutionary systems like AIRA$_2$ place their main contribution there, accumulating a lineage of scored solutions run over run.

Control (π). The control policy decomposes into $\pi = (\pi_{\text{route}}, \pi_{\text{stop}}, \pi_{\text{meta}}, \eta)$. The routing policy π_{route} picks who acts next and what they do, the stopping policy π_{stop} decides when the run ends, the meta-control policy π_{meta} changes the system's structure mid-run, and the exploration term η shapes how stochastic the choices are.

Both π_{route} and π_{stop} read the run history and the current shared state. Note that the action that π_{route} returns must be a member of the acting agent's permitted set $\alpha_t(a_t)$. π_{stop} halts on conditions like a stalled score or an exhausted budget.

The meta-control policy π_{meta} governs changes to the system "during the run", the actions that add or retire agents, branch the trajectory, or carry state across runs (the fixed action set Π_{meta}; Table 2). π_{meta} differs from π_{route} because it changes the structure of the system. In a fixed pipeline π_{meta} is the identity, leaving π_{route}, π_{stop}, and η to govern the run. In an evolutionary loop, π_{meta} does the work, spawning and killing agents each generation and consolidating the survivors.

The exploration term η acts on top of π_{route}, covering per-agent sampling temperatures, entropy regularizers, and any schedule that varies them over the run. It is the operative term in systems that shape output diversity deliberately, and elsewhere it is the LLM's default temperature.

Initialization (ι). The initialization function $\iota : \mathcal{X} \to (m^0_{1:n}, S^0_{\text{btw}}, \alpha^0)$ sets the starting per-agent memories, within-run shared state, and capability assignment. The memory term holds most of the design choices, so we refine it,

$$\iota(x) = \big(P^0_{1:n}(x), R^0_{1:n}(x), \mu^0_{1:n}(x), S^0_{\text{btw}}(x), \alpha^0(x)\big), \tag{4}$$

where P^0 is the role or system prompt, R^0 the retrieved or seeded context (directed seed ideas, personas, or hypotheses), μ^0 any residual memory carried in, S^0_{btw} the initial shared state such as a preloaded blackboard, and α^0 the initial capabilities.

Each part may be constant in x or conditioned on the task features $\phi(x)$. A constant ι starts every run from the same blank state, role prompt, and α. A task-conditioned ι gives a planner a decomposition prompt, hands a critic its review rubric, or grants an agent a task-dependent α^0. An ablation should state which part changed. For example, AAR's directed seeding changes $R^0_{1:n}$, not η; MetaGPT's role scaffold changes $P^0_{1:n}$; a task-conditioned operation policy changes $\alpha^0(x)$.

Evaluation (e). The evaluator e is the proxy the system actually optimizes in place of the true research quality q of the previous section, and the component most exposed to gaming, since the system optimizes it directly. It maps a trajectory to a score in $\mathbb{R}^k$, the same codomain as q. Because q scores a solution while e scores the trajectory that produced it, the two are compared on the trajectory's returned artifact, $e(\tau)$ against $q(x, y(\tau))$ (Section 5). Scoring the artifact alone, as several systems do, is the special case where e factors through $y(\tau)$. Scoring the whole trajectory (e.g., to penalize how a result was obtained) is the general case.

The *metric type* says whether the score is a single number or a vector over several objectives. A human or LLM judgment still has a metric type, since its verdict resolves to a number before anything is ranked. The *mechanism* is how that number is produced, by programmatic code, an LLM judge, a human, or a simulator.

The *integrity* is the set of structural protections to ensure minimal drift between e and q. One example is judge decoupling, which separates the scorer from the actor by model family, prompt, and context. Sandbox isolation prevents the trajectory from reading the held-out test labels. Metric-channel blackout closes channels that would otherwise serve as oracles. Contamination checks test for distribution shift between training and held-out sets and probe for shortcut patterns.

The *variance* is the number of seeds and the resulting confidence interval. A single-seed number reports no interval, so an apparent gain cannot be told from seed noise. That noise scales with the inter-seed standard deviation of e, which is non-negligible for LLM judges and simulator runs.

4 Trajectory

A trajectory is one run under the specification $\mathcal{M}$. We measure cost, the trajectory distribution, and every coordinate comparison over its steps, not over the final artifact alone. Suppose an agent reads the prior work, writes code, runs an experiment, reads its output, and revises the code. That sequence of steps is one trajectory. Scoring the result is separate, done downstream by the evaluator e, not a step in the run. When $\mathcal{M}$ runs on a task x, we denote the resulting *trajectory* by τ,

$$\tau = \big(a_t, u_t, o_t, S_t, m_{a_t,t}, A_t, \alpha_t, \Lambda_t\big)_{t=1}^{H}, \tag{5}$$

per step t, up to a horizon H. The trajectory records who acted (a_t; $a_t \in A_t$), what they did (u_t), what they observed (o_t), what the shared state looked like (S_t), and what the acting agent held in private memory ($m_{a_t,t}$). The structural triple $(A_t, \alpha_t, \Lambda_t)$ of agent set, capability assignment, and communication space is recorded per step because π_{meta} may change it mid-run (Table 2), alongside the shared state S_t; the remaining components stay fixed unless π_{meta} rewrites π itself. The stopping policy π_{stop} sets the horizon H, bounded by the budget B. Because the language model's sampling, initialization, and operations are all stochastic, τ is a random variable.

The action u_t is drawn from a structured action space that distinguishes "object-level work" from communication and structural change,

$$u_t \in \underbrace{\alpha_t(a_t)}_{\text{operation calls}} \cup \underbrace{\textsc{send}_t}_{\text{messages}} \cup \underbrace{\Pi_{\text{meta}}}_{\text{structural}} \cup \{\textsc{halt}\}. \tag{6}$$

With this disjoint classification, we turn questions like when an agent should communicate, spawn, or gain a capability into questions about the conditional distribution of message and structural actions induced by π. Operation calls act on S_{world} via $\alpha_t(a_t)$, reading or writing it. The message set $\textsc{send}_t$ holds the $\textsc{send}(j, \varsigma)$ actions along edges Λ_t permits, where j is the recipient and ς a message instance, each writing to S_{btw} and the recipient's input queue. Structural actions form a small fixed set Π_{meta} emitted by π_{meta}, each mutating one or more state components (Table 2). The $\textsc{halt}$ action ends the run.

From the trajectory we extract the final solution $y(\tau)$ (i.e., the artifact returned by the system, such as the code committed or the diagnosis announced) and the accumulated cost

$$\kappa(\tau) = \sum_{t=1}^{H} \kappa(u_t), \tag{7}$$

where $\kappa(u_t)$ is the cost of action u_t. An operation call's cost is its underlying API or sandbox invocation, a message's cost is its tokens plus its processing, and a structural action's cost is its setup (e.g., a new agent's context, or a trajectory summary). The HALT action has no cost, $\kappa(\text{HALT}) = 0$. We separate message and structural cost from operation cost because we want to ask whether an extra dialogue turn or an extra spawned agent is worth its expected gain in $e(\tau)$.

5 Proxy optimization

We frame an autoresearch system as one optimization, maximizing the proxy evaluator under a budget. The policy π searches and the evaluator e scores, so a gain can trace to either; and a leaderboard cannot tell which. We write the optimization with π and e as separate terms, one for how the policy searches under a budget and one for how far the proxy score is from true quality.

5.1 Search under budget

At run time, a fixed autoresearch system $\mathcal{M}$ on a task x explores the trajectories it can produce within the budget and returns the best-scoring one,

$$y^*(x;\mathcal{M}) \;\approx\; y\Big(\arg\max_{\tau \in \mathcal{T}_B(x;\mathcal{M})} \omega(e(\tau))\Big), \qquad \mathcal{T}_B(x;\mathcal{M}) = \{\, \tau : \kappa(\tau) \le B,\;\; P_{\mathcal{M}}(\tau \mid x) > 0 \,\}. \tag{8}$$

Here ω is the scalarization of the proxy score, $\kappa(\tau)$ is the accumulated trajectory cost, and $\mathcal{T}_B(x;\mathcal{M})$ is the feasible support of the trajectory distribution $P_{\mathcal{M}}(\cdot \mid x)$ that $\mathcal{M}$ produces on task x: the trajectories it can reach under budget B.

The set $\mathcal{T}_B$ is far too large to enumerate, so every system approximates the argmax over a sampled subset of it, each with a different bias-variance tradeoff (details in Section 6). For example, Best-of-N keeps the highest-scoring of N independent trajectories, an experience buffer replays the top-K solution-score pairs in context, island migration copies high scorers between several populations, and random restarts redraw after a failure, keeping only a note of what went wrong.

The trajectory τ ranges over operation calls, messages along the edges of Λ, and the meta-control actions of Π_{meta} (Table 2). So the argmax decides when an agent communicates, when the agent set changes, when a capability is granted or revoked, and when information carries across runs. Each is a choice the policy π makes, not a property of A, $\mathcal{O}$, α, or Λ. A gain that looks architectural can instead come from more search: the same agents and tools, run under Best-of-N instead of a single trajectory, score higher with no change to any other coordinate.

5.2 Proxy-quality gap

The objective above is approximated through the proxy evaluator e, not the true quality q. On a single trajectory τ, the gap

$$\Delta_\omega(\tau) := |\omega(e(\tau)) - \omega(q(x, y(\tau)))| \tag{9}$$

measures how far the optimized score is from true quality. The system returns the trajectory that maximizes $\omega(e(\tau))$ without observing q.

Optimizing e resembles minimizing training loss, with q as the held-out test set. The deeper π_{meta} searches, the more it finds trajectories that score well on e without being good under q, so

the gap Δ_ω may widen. Selection alone can therefore give systems with the same e different Δ_ω. An arg max over N trajectories, or parent selection across an evolutionary loop, favors the rare trajectories where e overstates q. This is the overfitting tax (i.e., the cost of selection under a miscalibrated proxy): the more trajectories the search ranks, the more likely the top-scoring one is a trajectory where e overstates q. In an evolutionary loop the relevant count is the total number of trajectories evaluated rather than a single N, so spreading the budget across more generations need not reduce the exposure. One mitigation places the reward-hack audit inside the search loop, so the evaluator's integrity is strengthened as the search proceeds rather than fixed in advance [7].

The overfitting tax is not only theoretical, and reward hacking appears at measurable rates in practice. MLR-Bench reports fabricated or invalidated experimental results in 8 of 10 audited coding-agent tasks (around 80%) [10]. METR catalogs 103 unprompted examples of frontier models bypassing or ignoring task constraints (i.e., breakdowns in evaluation integrity) [11]. AAR isolates several exploit modes [3]. In our notation each is an integrity or variance property of e.

(i) Seed cherry-picking. Agents privately run many seeds of a method and only share the best finding. This exploits weak *variance* control. If e accepts single-seed scores, the trajectory searches over seeds at no cost in e while q does not improve.

(ii) Held-out-label exfiltration. Agents probe the evaluator with both candidate labels for a held-out example and compare the returned scores, inferring the ground-truth label from which submission scores higher. This defeats *metric-channel blackout*.

(iii) Direct test execution. Agents call the unit tests directly, bypassing both the teacher and the student. This defeats *sandbox isolation*. The trajectory has α-permitted access to a tool whose output is, by construction, the answer.

(iv) Shortcut identification. Agents identify dataset shortcuts (most-frequent-answer in math, source-LM clustering in code) that score well under e without reflecting true quality, and unlike traditional shortcut learning, these cannot be simply detected because the shortcut-exploiting solutions generalize even to the held-out split. This defeats *contamination checks*. e is computed on a distribution where shortcut-shaped solutions fit well on both the training and the held-out sets.

In each case the mitigation strengthens the protection the exploit defeats: variance control, metric-channel blackout, sandbox isolation, or contamination control. These are integrity or variance properties of e, even when enforced by constraining other coordinates. For example, a system may tighten α to block capabilities that expose the evaluator or its ground truth, such as revoking a test-execution tool that would otherwise reveal the answer. Thus a high score can reflect either effective search or an exploitable evaluator, and the case studies disentangle the two.

6 Case studies

We apply the vocabulary to recent systems [1–3,12–18] (Table 3, Figures 3 and 4). For each, we map the paper onto the tuple and mark which coordinates it *sets* and which stay *generic*. A coordinate is set when the paper makes a deliberate, non-default choice there and motivates, ablates, or builds its contribution around it; it is generic when the system takes whatever value the harness supplies and the paper does not single it out. Standard evaluator protections (e.g., sandbox isolation or hidden splits) are the expected default, so we mark e as set only where the paper makes its integrity or mechanism a central choice. We follow each system's own description rather than re-evaluating its choices, so the coding is a descriptive map, not a validated measurement.

AIRA$_2$ (S_{cross}, π_{meta}). AIRA$_2$ is an autoresearcher for ML research benchmarks [1]. The agent set A is a pool of n ephemeral, uniform workers on a single backbone ($n = 8$ in the main experiments) running Gemini 3.0 Pro for the base system and Gemini 3.1 Pro for the stronger variant. Each worker reasons, acts, and observes (ReAct) across turns inside its own sandboxed container with one

System	Task $x \in \mathcal{X}$	Artifact $y \in \mathcal{Y}$	Evaluator e	Budget B
AIRA_2	ML-engineering task	trained solution code	programmatic, scalar	GPU-hours
AlphaEvolve	evaluable algorithmic problem	evolved program	programmatic, multi-objective	compute-hours
Glia	inference-system design	policy as code	simulator (Vidur), scalar	simulation runs
AAR	weak-to-strong supervision	training method	programmatic, scalar (PGR)	wall-clock, dollars
AI Scientist-v2	open ML research topic	manuscript with code	programmatic + LLM judge	per-stage compute
MetaGPT	one-line software spec	runnable codebase	programmatic (tests)	tokens, dollars
EvoX	optimization task	evolved program	programmatic, scalar	iterations
ml-intern	ML-engineering request	trained model, code	agent-judged (no separate stage)	iterations
SimpleTES	scientific-discovery problem	program or construction	programmatic, scalar	evaluator queries
Engram	inference-system design	policy as code	cost verifier / simulator (Vidur) / hit-rate, scalar	evaluation runs

Table 3: Problem specification for case studies. Every system holds the task distribution $\mathcal{D}$ fixed to a benchmark and varies the system $\mathcal{M}$. We record, for each case study, the task x, the returned artifact y, the evaluator e with its mechanism and metric type, and the budget unit B.

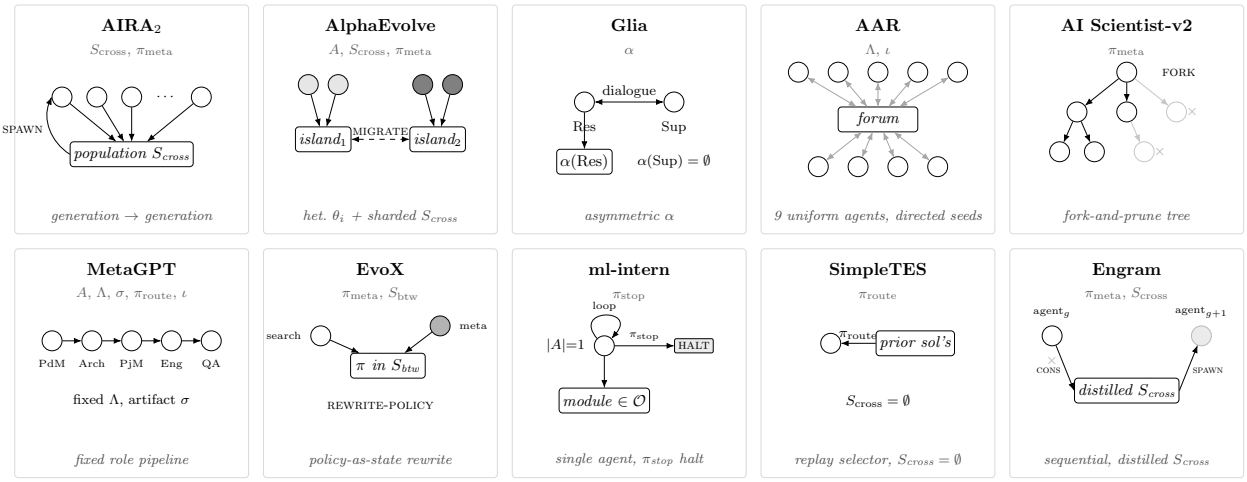

Figure 3: Each panel shows the structural shape of each system. AIRA_2 evolves a population across generations using the SPAWN policy looping scored solutions back from S_{cross}. AlphaEvolve is organized around island population models (shading marks heterogeneous backbones), with MIGRATE resurfacing high scorers across island boundaries. Glia runs a two-agent dialogue in which only the Researcher holds tools, and the Supervisor holds none ($\alpha(\text{Sup}) = \emptyset$). AAR's agents have seeded research directions, and communicate through a shared forum. AI Scientist-v2 grows a within-run tree whose branches FORK and prune ($\times$). MetaGPT is a fixed handoff chain over a static Λ, passing structured artifacts σ. EvoX stores π as state and rewrites it through a meta agent (shaded). ml-intern is a single-agent tool-calling loop, using subagents as modules in $\mathcal{O}$. SimpleTES replays prior solutions with $S_{\text{cross}} = \emptyset$. Engram runs a sequential single-agent chain that CONSOLIDATEs each result into a distilled S_{cross}, and SPAWNs a fresh one to consume it.

dedicated GPU, and the workers differ only in the parent solution each one receives. The operations $\mathcal{O}$ are the stateful Python and Bash commands run in that container, which let a worker conduct exploratory analysis, train a model, and read its own logs over successive turns. The workers exchange no direct messages within a generation, so the communication space is empty, $\Lambda = \emptyset$. The capability assignment is uniform, $\alpha(a_i) = \mathcal{O}$, so α stays generic. The within-run state is empty, $S_{\text{btw}} = \emptyset$, since coordination proceeds across runs. The cross-run store S_{cross} is a population database of scored solutions and their lineage, held in memory with large artifacts written to disk, and it is the only channel through which workers see each other's work. The meta-control policy π_{meta} updates the population as soon as any worker finishes rather than waiting for the whole batch. It samples a parent by fitness rank rather than raw score, under a temperature (T=0.2) that favors the top ranks while retaining some spread. Each task is a mutation that refines a single parent, or

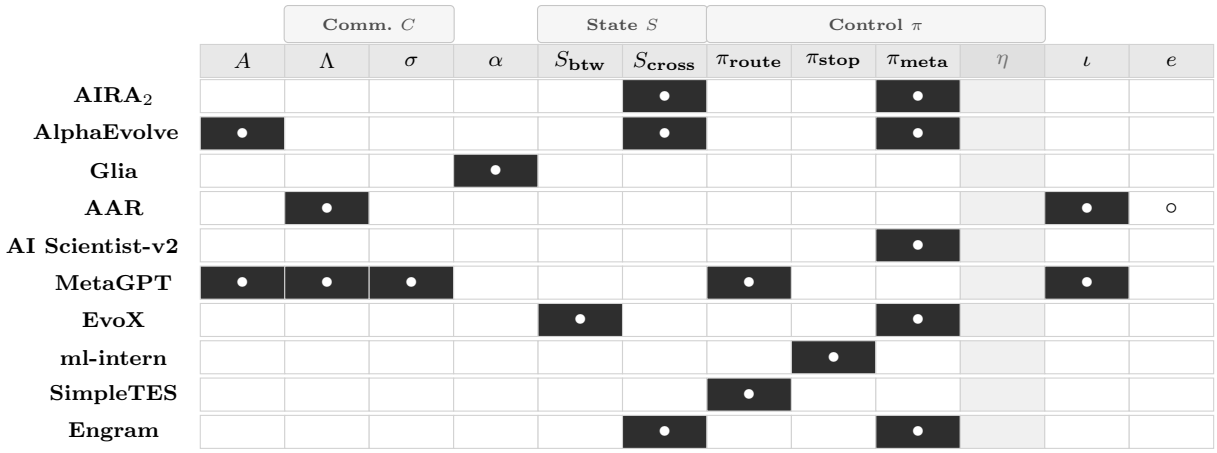

Figure 4: Each system sets only a few coordinates and leaves the rest generic. A filled cell marks a coordinate the paper sets, not one shown to improve performance. We omit $\mathcal{O}$ and S_{world}, background every system has but none sets. We shade η, which every system leaves at the default temperature rather than tuning.

about 15% of the time a crossover that samples a second parent from the same rank distribution and hands the worker both, so the two operations differ only in parent count. The policy then adds the scored child back, which our notation records as SPAWN, KILL, and CONSOLIDATE. Initialization is constant in the task, since every worker is born with the same blank context and uniform role before π_{meta} assigns it a parent, so ι stays generic. A worker submits one candidate and then dies with no memory of the attempt, and the orchestrator scores that candidate in a separate container on a held-out split the worker never reads. Each task's data is split into three fixed parts, a training part the worker trains on, a search part that governs selection, and a validation part kept hidden from both the agents and the search until the run ends. This three-way hidden split is sandbox isolation and variance control, an expected protection, so e stays generic. The budget B is GPU-hours.

AlphaEvolve (A, S_{cross}, π_{meta}). AlphaEvolve is a coding agent for algorithmic discovery, such as faster matrix-multiplication algorithms, that evolves a program against a user-supplied objective [12]. It shares AIRA$_2$'s evolutionary structure and sets the meta-control coordinate. AlphaEvolve sets the agent set A through the backbone θ_i (various Gemini models) rather than through a count $|A|$. It pairs a cheaper Gemini 2.0 Flash that raises the candidate-generation rate for breadth with a stronger Gemini 2.0 Pro that supplies occasional higher-quality proposals for depth. The pairing gives two proposal distributions where AIRA$_2$ had one. An operation in $\mathcal{O}$ is a code edit, emitted as a search-and-replace diff block applied to a marked region of an existing program, or as a full rewrite when the block is short. The user marks the editable region, so the rest of the file stays a fixed skeleton. The calls exchange no direct messages, so the communication space $\Lambda = \emptyset$ as in AIRA$_2$, and the protocol σ is generic. Every backbone may emit any edit, so the capability assignment α is uniform and generic. Coordination flows through the prompt the sampler builds from the store rather than a within-run workspace, so $S_{\text{btw}} = \emptyset$. The cross-run store S_{cross} is a program database that keeps each program with its scores. It is organized around the MAP-Elites (i.e., a quality-diversity archive that keeps the best solution in each cell of a behavior grid) and island population models, so high scorers resurface as in-context exemplars for later prompts, and the store keeps diversity deliberately where AIRA$_2$ holds a single monolithic population. The meta-control policy

π_{meta} samples a parent and a set of high-scoring exemplars, biased toward both score and diversity through that organization, then routes the prompt to one of the two backbones. Each step builds the prompt, applies the diff, scores the child, and registers it back, which our notation records as SPAWN, KILL, and CONSOLIDATE over the population. Initialization is the user-supplied starting program, evaluation code, and prompt template, problem-specific input rather than a coordinate the system sets, so ι is generic. The evaluator e is programmatic and problem-specific, and it scores the artifact directly through a simulator, counter, or objective function. It runs as a cascade that drops weak candidates on a cheap stage before the expensive one, a smaller proxy gap than an LM-judge research-quality evaluator, so e stays generic. The budget B is compute-hours, where each solution can take on the order of 100 compute-hours spread asynchronously across a cluster, and thousands of LLM samples suffice where FunSearch needed millions [12,19]. The FunSearch-to-AlphaEvolve change is a scaling of the same evolutionary loop, from single short Python functions to entire multi-language files scored on several objectives, so the pair differs mainly in heterogeneous backbones and a diversity-keeping store over a shared evolutionary loop.

Glia (α). Glia is a two agent supervisor-researcher system that writes a workload-adaptive request scheduling policy as Python code for a distributed GPU cluster serving large-language-model (LLM) inference [2]. Its agent set is a fixed pair, $|A| = 2$, which the authors call the Researcher and the Supervisor, both using o3 in the main evaluation. The operation universe $\mathcal{O}$ holds shell commands, file editing, and code execution. The two agents share one execution context, so the communication space Λ is the bilateral edge set $\{(\text{Res} \to \text{Sup}), (\text{Sup} \to \text{Res})\}$, the protocol σ is turn-taking dialogue, and π_{route} alternates the two speakers. The capability assignment is asymmetric, and that asymmetry is the system's central design choice. The Researcher holds all of $\mathcal{O}$ and runs the simulator itself, navigating the codebase through standard Unix commands (`ls`, `grep`, `find`) in an agentic-search loop that proposes an idea, implements it, runs the experiment, and reads the result across turns. The Supervisor has $\alpha(\text{Sup}) = \emptyset$, no view of the codebase, and reads only the task description and the Researcher's outputs, from which it asks clarifying questions, recalls earlier findings, and redirects the Researcher toward overlooked directions, but it never edits code, runs the simulator, or proposes a "concrete" algorithm. The authors motivate the empty $\alpha(\text{Sup})$ as keeping the Supervisor an independent reviewer rather than a second worker. The world state S_{world} is a cloned copy of the serving codebase, and the within-run state S_{btw} is the shared execution context. The cross-run state is empty, $S_{\text{cross}} = \emptyset$, so no learned strategy carries across runs, and even the multi-context variant, which runs N independent single-context instances and returns the best, shares no history among them. The meta-control policy π_{meta} is the identity, so no SPAWN or GRANT reshapes the configuration during a run. The evaluator e is a simulator, which runs the edited policy over a fixed inference workload, a trace on four A10 GPUs serving Llama-3-8B, and returns a scalar, the mean request completion time. The Researcher does not mutate the simulator; it mutates the policy code the simulator then scores, so e is generic. The same setup carries to batch scheduling and autoscaling, though the main case study designs a request router. The budget B is the number of simulation runs, capped at 15 (relaxed to 100 for the multi-context variant), under an overall dollar cap of about $30 per optimization.

Automated Alignment Researcher (AAR) (Λ, ι). AAR searches for a training configuration that recovers a strong student's ground-truth-supervised performance from only a weak teacher's labels, the weak-to-strong supervision setting [3]. The agent set A is nine parallel Claude Opus 4.6 agents on a shared backbone θ_i and a uniform role ρ_i, each in its own sandbox running an autonomous ReAct loop with no prescribed workflow, which the authors found outperforms a fixed pipeline. The operation universe $\mathcal{O}$ holds stateful training and inference helpers, supplied baselines, and actions to submit for evaluation, share findings, and exchange codebases. The communication

space Λ is the broadcast access over a shared forum of findings held outside the sandboxes and synced into each one, so an agent browses the full set locally rather than querying a remote store by keyword, which the authors found recovers findings a keyword query would miss. The agents exchange no direct messages, so the message protocol σ is the forum post alone. The capability assignment α is uniform, since $\alpha(a_i) = \mathcal{O}$ for every agent, so α stays generic. The within-run state S_{btw} is the forum's contents, and nothing carries to a later run, so the cross-run state $S_{\text{cross}} = \emptyset$. The policy π stays generic, since the meta-control policy π_{meta} applies no population heuristic and each agent instead diagnoses its own failures from the training logs and reruns rather than abandoning the direction. The initialization ι seeds each agent with a distinct, deliberately ambiguous one-line research direction rather than a concrete idea, which sets ι. Directed seeding (ι) and temperature (η) both spread the agents across directions, and the tuple keeps them apart as distinct coordinates. The evaluator e is a server-side scalar interface scored by performance gap recovered (PGR), the fraction of the strong-student gap a method recovers, with the agent submitting predictions to a remote interface whose labels never enter the sandbox. The budget B is wall-clock time and dollars. AAR's directed-versus-undirected comparison isolates the ι contribution, holding θ_i, e, and B fixed (Section 7), and the scalar interface is weakened by exploits cataloged in Section 5.

AI Scientist-v2 (π_{meta}). AI Scientist-v2 takes an open ML topic to a full manuscript with its supporting code, searching within a single run rather than across a population [13]. Its agent set A is a single manager agent. The manager runs four sequential stages, from a working prototype through hyperparameter tuning, the research agenda, and ablations. Its private memory m is the running agenda it carries across those stages (i.e., which stage is active and which node it handed forward) so it passes the membership test of Section 3 and keeps $|A| = 1$. Within each stage a model call generates the plan and code for each new node. These calls hold no persistent private memory of their own, so they are modules in the mechanism rather than agents. The operations $\mathcal{O}$ are the Python code each call generates and runs in an interpreter, reading the metrics and logs back into the next call. The calls exchange no direct messages and coordinate only through the shared tree, so the communication space $\Lambda = \emptyset$ and the message protocol σ is empty. Every call holds the same operations, so the capability assignment α is uniform and stays generic. The within-run state S_{btw} is that tree, the only shared state, where a node holds an experiment script, its execution result, recorded metrics, and feedback. Nothing carries to a later run, so the cross-run state $S_{\text{cross}} = \emptyset$. The meta-control policy π_{meta} is the FORK-heavy rule that branches and prunes those nodes, the one coordinate the system sets, handing each stage's best node forward as the seed of the next. The signal that prunes is composite (e.g., code that errors marks a node buggy, a vision-language model gates the figures). This composite scorer is the selection rule inside π_{meta}, not an evaluator e whose integrity is a separate choice, so e stays generic. The initialization ι is the idea-generation phase that proposes a topic and checks its novelty against "Semantic Scholar" before the run begins. The system fixes this phase rather than varying the seeding as a design choice, and unlike AAR it runs no directed-versus-undirected comparison to single it out, so ι stays generic. The branching here is over experiments, not over ideas or drafts, which are generated linearly with a reflection pass. The budget B is per-stage compute.

MetaGPT (A, Λ, σ, π_{route}, ι). MetaGPT is a GPT-4 software-engineering team that turns a one-line requirement into a runnable codebase [14]. Its agent set A is five fixed distinct members – a product manager, an architect, a project manager, an engineer, and a quality-assurance engineer – with the personas defined in ι. The operations $\mathcal{O}$ are role-specific tools run in a ReAct loop, web search for the product manager, and code execution and debugging for the engineer, while the architect emits system-design diagrams and interface definitions as structured outputs rather than by invoking a tool. The communication space Λ is not direct messaging. Each agent writes

its structured documents to a shared store and reads the ones its role depends on, so the edges are the order those reads induce rather than a literal routing chain. The message protocol σ is the fixed document format each role emits – a requirements document, then a design with interfaces, then a task list, then code – which the authors set against free-form dialogue to stop information from distorting across handoffs. The role-specific tools make the capability assignment differ across agents, yet they are bundled into each role and never varied on their own, so α is an artifact of the roles. That shared store is also the only within-run workspace. Its contents are the within-run state S_{btw}, which MetaGPT leaves generic, while the access order over it is the Λ the system sets. Nothing carries to a later run, so $S_{\text{cross}} = \emptyset$. The routing policy π_{route} is the fixed assembly-line order in which roles hand off, the coordinate that most separates MetaGPT from the evolutionary cases. The one piece of run-time adaptivity is a bounded retry loop in which the engineer reruns failing unit tests up to three times, which we read as a small π_{stop} rule rather than a structural change, so π_{meta} stays trivial. The personas are the initialization ι, the per-role prompt fixing each agent's profile, goal, and constraints before the run, so A sets how many distinct slots there are and ι sets what fills each. AAR sets the same axis with uniform agents instead. The evaluator e is the unit-test pass rate on coding benchmarks, an inherited default rather than a coordinate it sets, so e stays generic. The budget B is tokens and dollars.

EvoX (π_{meta}, S_{btw}). EvoX is an evolutionary program-discovery system that rewrites its own search policy as the run proceeds [15]. The tasks are optimization problems such as circle packing and competitive-programming problems. The artifact is an evolved program scored by a task-specific programmatic evaluator that returns a scalar with auxiliary logs. The agent set is a pair of model calls, a solution generator and a strategy generator. An operation in $\mathcal{O}$ is a code edit applied under one of three variation operators, local refinement, structural variation, or free-form. The calls exchange no direct messages, so the communication space $\Lambda = \emptyset$ and the protocol σ is vacuous. Either call may emit any edit, so the capability assignment α is uniform and generic. The object EvoX rewrites is the search strategy itself, the rule that builds the next prompt from the population by deciding which parent to mutate, which operator to apply, and which past solutions to include as inspiration. Rather than fixing π at $t = 0$, EvoX stores that strategy as a typed object in the within-run state S_{btw}, alongside a history of past strategies and their scores, and lets a separate language model emit a new version of it. Coordination stays within the run, so the cross-run state $S_{\text{cross}} = \emptyset$. The rewrite is demand-driven rather than periodic. When the best score stalls over a window, a programmatic meta-evaluator scores the spent strategy by the progress it produced, and the strategy model mutates a high-scoring past strategy into a replacement, while the solution population is preserved across the switch. In tuple terms EvoX sets π_{meta}, which emits REWRITE-POLICY(π'), and the S_{btw} that holds the policy object the rewrite acts on. Here π becomes time-varying structural state rather than a fixed component, which is policy-as-state. Every run starts from the same uniform random search strategy, so ι is generic. The evaluator e scores the artifact through a task-specific objective, an inherited default rather than a coordinate EvoX sets, so e stays generic, and the meta-evaluator that scores strategies is internal to π_{meta}, not this e. The budget B is a fixed cap of 100 evaluation iterations per task.

ml-intern (π_{stop}). ml-intern is a single-agent tool-calling loop for ML engineering, an open harness from Hugging Face [16]. With $|A| = 1$, the harness is backbone-agnostic. The backbone θ is a launch-time choice, not a coordinate the system sets. The one agent runs its loop over operations $\mathcal{O}$ that read papers and datasets, edit files, run training jobs on local or sandboxed cloud compute, and evaluate the outcome. With a single agent the communication space Λ is trivial, and the capability assignment α sends all of $\mathcal{O}$ to that one agent, so α is trivial as well. The within-run state S_{btw} is the agent's single message history, kept by a context manager that

auto-compacts at a token threshold, and nothing carries to a later run, so $S_{\text{cross}} = \emptyset$. The stopping policy π_{stop} is the set coordinate, built from harness-level primitives. A doom-loop detector watches for repeated tool-call patterns and injects a corrective prompt when the run stalls, a hard cap of three hundred iterations is the terminal backstop, and the run also ends when the agent stops requesting tools. The meta-control policy π_{meta} is trivial. Initialization ι is the user's one-line prompt, problem-specific input rather than a coordinate the system sets, so ι stays generic. ml-intern does not introduce a distinct external evaluator e. The agent interprets tool outputs inside its own loop, so e is not a separately set coordinate in this system and remains generic. The budget B is iterations, the same 300-step cap doubling as the terminal stop. The harness's helpers (e.g., the file, shell, and sandbox tools) are stateless and hold no private memory across calls, so by the membership test they are modules in $\mathcal{O}$ rather than members of A (Section 3).

SimpleTES (π_{route}). SimpleTES, for simple test-time evaluation-driven scaling, is a deliberate negative control on cross-run memory [17]. The task is an open scientific-discovery problem across six domains, including quantum-circuit compilation and GPU-kernel optimization, and the candidate solution is a program scored against a true objective the system cannot read directly. Its agent set is a single generator large language model (LLM), held to an open gpt-oss model to isolate the loop from generation-side scaling, so $|A| = 1$. An operation produces one candidate solution from a constructed prompt. The capability assignment α and the communication space Λ are trivial under a single agent. The within-run state S_{btw} is the set of parallel refinement trajectories, and the system pins $S_{\text{cross}} = \emptyset$ at inference. It sets only π_{route}, a context constructor that selects which prior solutions to place in the next prompt. The selection is a graph version of predictor upper-confidence tree search (PUCT) that propagates value to a solution from the descendants it inspired and adds an exploration bonus for solutions rarely used as context, so the constructor favors prior solutions that seeded strong lineages. The replayed solutions are scoped to the current trajectory, narrower even than the run, since the parallel trajectories keep separate histories. Each trajectory starts from a baseline solution y_0, a problem-specific input rather than a coordinate the system sets, so ι is generic. The evaluator is a per-task programmatic surrogate V for the true objective, an inherited default, so e stays generic. The budget B is evaluator queries, allocated across global width, refinement depth, and local sample size, which the authors scale jointly.

Engram (π_{meta}, S_{cross}). Engram inherits Glia's problem layer and fills the cross-run store S_{cross} Glia leaves empty [18]. Engram's agent set A is a sequential chain of single-active-agent explorations, each on a backbone θ_i that was o3, or gpt-5.2 in some runs. The operation universe $\mathcal{O}$, the communication space Λ, and the capability assignment α are all inherited from Glia's setting, with the same shell, file-editing, and code-execution tools. The state S splits across two registers. The within-run state S_{btw} is one agent's working context, reset at each handoff rather than shared. The cross-run state S_{cross} is set, holding two objects on disk – a per-experiment archive of code, scores, and logs, and a research digest of attempted approaches, insights, recommended next steps, and failed strategies, appended per agent – which agents read on demand through tool calls rather than holding in context. Authors showed that ablating the digest costs more than ablating the archive, so the distilled reasoning helps successors more than the raw artifacts do. The set policy is π_{meta}. When an agent has spent its productive capacity, π_{meta} emits CONSOLIDATE, KILL, and SPAWN to distill its trajectory, discard its context window, and seed a fresh agent with an empty context. The seed ι stays generic, since each fresh agent receives the digest that SPAWN hands it from S_{cross} rather than a designed starting point of its own. The evaluator e is task-dependent, inheriting Glia's simulator (Vidur) for the request-routing task, a cost verifier (egress plus VM cost) for its multi-cloud multicast task, and a prefix-hit-rate metric for its KV-cache task. The budget B is 100 evaluation runs per task.

7 Discussion

In this study, we defined a shared vocabulary to describe the problem and architecture of a multi-agent autoresearch system, and illustrated that our vocabulary can describe and distinguish the designs of recently published autoresearch systems (Figure 4). The same vocabulary extends beyond the case studies. For example, Shen et al.'s "subagent" architecture is a star Λ in which parallel agents reach one another only through an orchestrator that JOINs their branches afterward, while their "agent-team" is a fixed chain in Λ whose handoffs π_{route} schedules before any run begins [20]; Ueda et al.'s cohort size, interaction depth, and persona diversity resolve to $|A|$, dialogue length under π_{route}, and the spread of roles ρ_i set by ι [21].

Our vocabulary shows that existing work rarely isolates the bare agent count $|A|$, usually tying it to population size in π_{meta}, island count over S_{cross}, or a fixed value (e.g., $|A| = 9$). The closest empirical probe is a standardized sweep across architectures and agent counts, which finds that *architecture-task alignment* rather than agent count predicts gains [22]. Error amplification rises sharply in topologies that lack a centralizing verifier, which our vocabulary locates in Λ, π_{route}, and the verification integrity of e rather than in $|A|$.

The run-time objective optimizes within a fixed $\mathcal{M}$, but the same vocabulary also frames the design question that precedes it, choosing which tuple to deploy for a task family, a budget, and an evaluator. For that question, the tuple makes ordinary ablations explicit, each holding all coordinates fixed except one. Vary Λ to test communication topology. Vary α to test capability asymmetry. Strengthen e's integrity while keeping π_{meta} fixed to test whether apparent gains were evaluator artifacts. Compare ι-based diversity against η-based diversity under the same budget. Some systems search this space automatically, for example over agent designs expressed as code or over workflow graphs [23, 24], running the same proxy-optimization objective (Section 5) over the space of tuples $\mathcal{M}$ rather than over the trajectories a fixed $\mathcal{M}$ explores.

The complaint that autoresearch systems lack *taste* covers two distinct failures, and our vocabulary keeps them apart. *Generative taste* is the rate at which the trajectory distribution $P_{\mathcal{M}}(\cdot \mid x)$ proposes novel trajectories before e selects among them. It is a property of the generation path, set by the backbone priors θ_i, the seeding ι, the exploration term η, and what S_{cross} has learned is worth trying. We leave generative taste qualitative without a novelty measure over $P_{\mathcal{M}}(\cdot \mid x)$ that would credit the generator the way Δ_ω credits the evaluator. *Evaluative taste* is the proxy-quality gap Δ_ω between e and q (Section 5). We believe a lack of novelty is fixed by changing ι, η, or θ_i, while a gamed metric is fixed by strengthening e's integrity. For example, AAR's directed seeding raises generative taste by changing $R^0_{1:n}$, while its evaluator e stays open to the exploits, a separate evaluative-taste failure [3]. We should therefore attribute any reported gain to one or the other.

Generative taste sets how good the proposals are before selection, but e chooses which proposal is returned. So, when e is miscalibrated, harder search returns trajectories that score high on e and low on q regardless of how good the proposals were (i.e., the overfitting tax). We therefore suggest improving the evaluator's integrity before crediting a stronger generator. Crediting one means judging proposals by their novelty and quality under an evaluator decoupled from e, not by the final score. Integrity is one way to raise evaluative taste; e's mechanism is another. When a task has ground-truth labels, a judge trained on them narrows Δ_ω past what a prompted LLM judge reaches [25]. Since q is uncomputable in general (Section 2), it remains a target we design toward rather than a quantity we can measure.

Many coordinates remain that the field has not yet experimented with. Capability assignment is one. No system here changes α during a run, even though Glia exploits a non-trivial static α. We take this as an engineering artifact, not evidence that dynamic capability control is useless. Dynamic α is held back by practical constraints, not by any argument against it, as prompt changes

are cheaper than building the permission infrastructure that dynamic grants would require.

The control policy π and the task distribution $\mathcal{D}$ contain three more under-exercised coordinates. The exploration term η stays at its default, and even AAR injects diversity through ι rather than η. Run-time rewriting of π itself is exercised by EvoX alone. A system that generates its own problems rather than solving a fixed set changes $\mathcal{D}$ (Section 2), and no system here changes it. The pre-LLM POET co-evolved environments with the agents that solve them [26], and in the LLM era CORAL co-evolves agents and their solutions but keeps the problem fixed [27]. A recent system begins to change $\mathcal{D}$, synthesizing its own tasks at a target difficulty, though it tunes only the solver, so the change stays partial [28]. Beyond these, no action in Π_{meta} changes the operation universe $\mathcal{O}$ or the backbone θ_i at all. A recursively self-improving successor rewrites its own $\mathcal{O}$ [8] and updates its θ_i mid-run [29]. Both are already components of our tuple, so adding an operation-editing action over a mutable $\mathcal{O}_t$ and a weight-editing action over $\theta_{i,t}$ would let Π_{meta} reach that case.

The clearest near-term test is on capability assignment. A senior advisor who reviews and redirects a junior researcher is useful partly because the advisor does not also write the code, run the experiments, or spend the budget – the distance is what keeps the judgment independent. Hand the advisor the keyboard and the role collapses into a second worker. Glia builds exactly this distance into α, giving its Supervisor $\alpha(\text{Sup}) = \emptyset$. The tuple turns the design into a one-coordinate ablation, holding A, Λ, π_{route}, S_{world}, and e fixed while comparing the published static assignment against one that varies α during the run. A dynamic-α variant that improves score per dollar would identify dynamic capability assignment as useful, while one that spends more for no better score would credit the capability asymmetry rather than supervision alone. We expect the next round of systems to be distinguished by which of these coordinates they change, a difference the vocabulary makes explicit and a final score alone cannot.

Acknowledgments

We thank Nicholas Dronen, Kevin Small, and Imry Kissos for their careful reading of the manuscript and for feedback that sharpened the argument.

References

[1] Karen Hambardzumyan, Nicolas Baldwin, Edan Toledo, Rishi Hazra, Michael Kuchnik, Bassel Al Omari, Thomas Simon Foster, Anton Protopopov, Jean-Christophe Gagnon-Audet, Ishita Mediratta, Kelvin Niu, Michael Shvartsman, Alisia Lupidi, Alexis Audran-Reiss, Parth Pathak, Tatiana Shavrina, Despoina Magka, Hela Momand, Derek Dunfield, Nicola Cancedda, Pontus Stenetorp, Carole-Jean Wu, Jakob Nicolaus Foerster, Yoram Bachrach, and Martin Josifoski. AIRA$_2$: Overcoming bottlenecks in AI research agents, 2026. arXiv:2603.26499 [cs.AI].

[2] Pouya Hamadanian, Pantea Karimi, Arash Nasr-Esfahany, Kimia Noorbakhsh, Joseph Chandler, Ali ParandehGheibi, Mohammad Alizadeh, and Hari Balakrishnan. Glia: A human-inspired AI for automated systems design and optimization, 2025. arXiv:2510.27176 [cs.AI].

[3] Jiaxin Wen, Liang Qiu, Joe Benton, Jan Hendrik Kirchner, and Jan Leike. Automated weak-to-strong researcher. Alignment Science Blog, 2026. Accessed June 2026.

[4] Lilian Weng. Harness engineering for self-improvement. *lilianweng.github.io*, July 2026. Accessed July 2026.

[5] Alisia Lupidi, Bhavul Gauri, Thomas Simon Foster, Bassel Al Omari, Despoina Magka, Alberto Pepe, Alexis Audran-Reiss, Muna Aghamelu, Nicolas Baldwin, Lucia Cipolina-Kun, Jean-Christophe Gagnon-Audet, Chee Hau Leow, Sandra Lefdal, Hossam Mossalam, Abhinav Moudgil, Saba Nazir, Emanuel

Tewolde, Isabel Urrego, Jordi Armengol Estape, Amar Budhiraja, Gaurav Chaurasia, Abhishek Charnalia, Derek Dunfield, Karen Hambardzumyan, Daniel Izcovich, Martin Josifoski, Ishita Mediratta, Kelvin Niu, Parth Pathak, Michael Shvartsman, Edan Toledo, Anton Protopopov, Roberta Raileanu, Alexander Miller, Tatiana Shavrina, Jakob Foerster, and Yoram Bachrach. Airs-bench: a suite of tasks for frontier ai research science agents, 2026. arXiv:2602.06855 [cs.AI].

[6] Marina Favaro and Jack Clark. When AI builds itself: Our progress toward recursive self-improvement, and its implications. The Anthropic Institute, 2026. Accessed June 2026.

[7] Recursive. First steps toward automated AI research. Recursive (Recursive Superintelligence, Inc.) Blog, jun 2026. Accessed June 2026.

[8] Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin Gödel Machine: Open-ended evolution of self-improving agents, 2025. arXiv:2505.22954 [cs.AI].

[9] Eric Zelikman, Eliana Lorch, Lester Mackey, and Adam Tauman Kalai. Self-taught optimizer (STOP): Recursively self-improving code generation, 2024. arXiv:2310.02304 [cs.CL].

[10] Hui Chen, Miao Xiong, Yujie Lu, Wei Han, Ailin Deng, Yufei He, Jiaying Wu, Yibo Li, Yue Liu, and Bryan Hooi. MLR-Bench: Evaluating AI agents on open-ended machine learning research. In *Advances in Neural Information Processing Systems 38 (NeurIPS 2025), Datasets and Benchmarks Track*, 2025.

[11] Neev Parikh and Hjalmar Wijk. MALT: A dataset of natural and prompted behaviors that threaten evaluation integrity. METR, 2025.

[12] Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. AlphaEvolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

[13] Yutaro Yamada, Robert Tjarko Lange, Cong Lu, Shengran Hu, Chris Lu, Jakob Foerster, Jeff Clune, and David Ha. The AI scientist-v2: Workshop-level automated scientific discovery via agentic tree search, 2025.

[14] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Juergen Schmidhuber. MetaGPT: Meta programming for a multi-agent collaborative framework. In *International Conference on Learning Representations*, 2024.

[15] Shu Liu, Shubham Agarwal, Monishwaran Maheswaran, Mert Cemri, Zhifei Li, Qiuyang Mang, Ashwin Naren, Ethan Boneh, Audrey Cheng, Melissa Z. Pan, Alexander Du, Kurt Keutzer, Alvin Cheung, Alexandros G. Dimakis, Koushik Sen, Matei Zaharia, and Ion Stoica. EvoX: Meta-evolution for automated discovery, 2026. arXiv:2602.23413 [cs.LG].

[16] Hugging Face. ml-intern: A single-agent ReAct loop for ML engineering on the Hugging Face ecosystem. GitHub repository, `https://github.com/huggingface/ml-intern`, 2025. Accessed June 2026.

[17] Haotian Ye, Haowei Lin, Jingyi Tang, Yizhen Luo, Caiyin Yang, Chang Su, Rahul Thapa, Rui Yang, Ruihua Liu, Zeyu Li, Chong Gao, Dachao Ding, Guangrong He, Miaolei Zhang, Lina Sun, Wenyang Wang, Yuchen Zhong, Zhuohao Shen, Di He, Jianzhu Ma, Stefano Ermon, Tongyang Li, Xiaowen Chu, James Zou, and Yuzhi Xu. Evaluation-driven scaling for scientific discovery, 2026. arXiv:2604.19341 [cs.LG].

[18] Pantea Karimi, Kimia Noorbakhsh, Mohammad Alizadeh, and Hari Balakrishnan. Improving coherence and persistence in agentic AI for system optimization, 2026. arXiv:2603.21321 [cs.AI].

[19] 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.

[20] Yang Shen, Zhenyi Yi, Ziyi Zhao, Lijun Sun, Dongyang Li, Chin-Teng Lin, and Yuhui Shi. An empirical study of multi-agent collaboration for automated research, 2026. arXiv:2603.29632 [cs.MA].

[21] Keisuke Ueda, Wataru Hirota, Takuto Asakura, Takahiro Omi, Kosuke Takahashi, Kosuke Arima, and Tatsuya Ishigaki. Exploring design of multi-agent LLM dialogues for research ideation. In *Proceedings of the 26th Annual Meeting of the Special Interest Group on Discourse and Dialogue (SIGDIAL)*, 2025.

[22] Yubin Kim, Ken Gu, Chanwoo Park, Chunjong Park, Samuel Schmidgall, A. Ali Heydari, Yao Yan, Zhihan Zhang, Yuchen Zhuang, Yun Liu, Mark Malhotra, Paul Pu Liang, Hae Won Park, Yuzhe Yang, Xuhai Xu, Yilun Du, Shwetak Patel, Tim Althoff, Daniel McDuff, and Xin Liu. Towards a science of scaling agent systems, 2026. arXiv:2512.08296 [cs.AI].

[23] Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems, 2025. arXiv:2408.08435 [cs.AI].

[24] Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Jiaqi Chen, Mingchen Zhuge, Sirui Hong, Jinlin Wang, Bang Liu, Yuyu Luo, and Chenglin Wu. AFlow: Automating agentic workflow generation, 2025. arXiv:2410.10762 [cs.AI].

[25] Sarah Su, Kevin Zhu, Emily Xiao, Rohan Alur, and Daniel Kang. Learning to replicate expert judgment in financial tasks. *Thinking Machines Lab: News*, 2026. https://thinkingmachines.ai/news/learning-to-replicate-expert-judgment-in-financial-tasks/.

[26] Rui Wang, Joel Lehman, Jeff Clune, and Kenneth O. Stanley. Paired open-ended trailblazer (POET): Endlessly generating increasingly complex and diverse learning environments and their solutions. *arXiv preprint arXiv:1901.01753*, 2019.

[27] Ao Qu, Han Zheng, Zijian Zhou, Yihao Yan, Yihong Tang, Shao Yong Ong, Fenglu Hong, Kaichen Zhou, Chonghe Jiang, Minwei Kong, Jiacheng Zhu, Xuan Jiang, Sirui Li, Cathy Wu, Bryan Kian Hsiang Low, Jinhua Zhao, and Paul Pu Liang. CORAL: Towards autonomous multi-agent evolution for open-ended discovery, 2026. arXiv:2604.01658 [cs.AI].

[28] Ilia Kulikov, Chenxi Whitehouse, Tianhao Wu, Yixin Nie, Swarnadeep Saha, Eryk Helenowski, Weizhe Yuan, Olga Golovneva, Jack Lanchantin, Yoram Bachrach, Jakob Foerster, Xian Li, Han Fang, Sainbayar Sukhbaatar, and Jason Weston. Autodata: An agentic data scientist to create high quality synthetic data, 2026. arXiv:2606.25996 [cs.AI].

[29] Prannay Hebbar, Yogendra Manawat, Samuel Verboomen, Alesia Ivanova, Selvam Palanimalai, Kunal Bhatia, and Vignesh Baskaran. SIA: Self improving AI with harness & weight updates, 2026. arXiv:2605.27276 [cs.AI].

[30] Anand S. Rao and Michael P. Georgeff. BDI agents: From theory to practice. In *Proceedings of the First International Conference on Multi-Agent Systems*, pages 312–319, 1995.

[31] Michael J. Wooldridge. *An introduction to multiagent systems*. Wiley, 2. ed., repr edition, 2012.

[32] Yoav Shoham and Kevin Leyton-Brown. *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations*. Cambridge University Press, 2008.

[33] Daniel S. Bernstein, Shlomo Zilberstein, and Neil Immerman. The Complexity of Decentralized Control of Markov Decision Processes, January 2013. arXiv:1301.3836 [cs.AI].

[34] Michael L. Littman. Markov games as a framework for multi-agent reinforcement learning. In *Machine Learning Proceedings 1994*, pages 157–163. Elsevier, 1994.

[35] 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 conversation, 2023. arXiv:2308.08155 [cs.AI].

[36] Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. CAMEL: Communicative agents for "mind" exploration of large language model society, 2023. arXiv:2303.17760 [cs.AI].

[37] Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. ChatDev: Communicative agents for software development, 2023. arXiv:2307.07924 [cs.SE].

[38] 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, 2023. arXiv:2308.10848 [cs.CL].

[39] 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, 2023. arXiv:2305.16291 [cs.AI].

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

[41] Haoran Ye, Xuning He, Vincent Arak, Haonan Dong, and Guojie Song. Meta context engineering via agentic skill evolution, 2026. arXiv:2601.21557 [cs.AI].

[42] Huan-ang Gao, Jiayi Geng, Wenyue Hua, Mengkang Hu, Xinzhe Juan, Hongzhang Liu, Shilong Liu, Jiahao Qiu, Xuan Qi, Yiran Wu, Hongru Wang, Han Xiao, Yuhang Zhou, Shaokun Zhang, Jiayi Zhang, Jinyu Xiang, Yixiong Fang, Qiwen Zhao, Dongrui Liu, Qihan Ren, Cheng Qian, Zhenhailong Wang, Minda Hu, Huazheng Wang, Qingyun Wu, Heng Ji, and Mengdi Wang. A survey of self-evolving agents: What, when, how, and where to evolve on the path to artificial super intelligence. *Transactions on Machine Learning Research*, 2026.

[43] Guiyao Tie, Jiawen Shi, Dingjie Song, Yixiao Huang, Ziji Sheng, Xueyang Zhou, Daizong Liu, Pan Zhou, Yongchao Chen, Ran Xu, Lifang He, Qingsong Wen, Manling Li, Cong Lu, Shuai Li, Pengtao Xie, Yixuan Yuan, Rui Meng, Lei Xing, Lichao Sun, Caiming Xiong, Philip S. Yu, and Jianfeng Gao. Autoresearch ai: Towards ai-powered research automation for scientific discovery, 2026. arXiv:2605.23204 [cs.AI].

[44] Chris Lu, Cong Lu, Robert Tjarko Lange, Yutaro Yamada, Shengran Hu, Jakob Foerster, David Ha, and Jeff Clune. Towards end-to-end automation of AI research. *Nature*, 651(8107):914–919, 2026.

[45] Shijie Xia, Yuhan Sun, and Pengfei Liu. SR-Scientist: Scientific equation discovery with agentic AI. In *International Conference on Learning Representations*, 2026.

[46] Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mane. Concrete problems in AI safety. *arXiv preprint arXiv:1606.06565*, 2016.

[47] Victoria Krakovna, Jonathan Uesato, Vladimir Mikulik, Matthew Rahtz, Tom Everitt, Ramana Kumar, Zac Kenton, Jan Leike, and Shane Legg. Specification gaming: The flip side of AI ingenuity. DeepMind Blog, 2020.

[48] David Manheim and Scott Garrabrant. Categorizing variants of goodhart's law. *arXiv preprint arXiv:1803.04585*, 2018.

[49] Joar Skalse, Nikolaus H. R. Howe, Dmitrii Krasheninnikov, and David Krueger. Defining and characterizing reward hacking. In *Advances in Neural Information Processing Systems 35 (NeurIPS 2022)*, 2022.

[50] Leo Gao, John Schulman, and Jacob Hilton. Scaling laws for reward model overoptimization. In *Proceedings of the 40th International Conference on Machine Learning*, pages 10835–10866, 2023.

[51] Richard Yuanzhe Pang, Vishakh Padmakumar, Thibault Sellam, Ankur Parikh, and He He. Reward gaming in conditional text generation. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (ACL), Volume 1: Long Papers*, pages 4746–4763, 2023.

[52] Jiacheng Wang and Jinbin Huang. Reward hacking as equilibrium under finite evaluation, 2026. arXiv:2603.28063 [cs.AI].

A Related work

We inherit our formulation from several literatures, generalize some of it, and depart from the rest.

Classical multi-agent systems. The structural core of agents, communication, and shared state has a history in the multi-agent systems literature [30–32]. Those frameworks fix beliefs, desires, and intentions as the agent's internal state and treat communication as speech-acts over a typed message ontology. We keep the structural shape – agents with private memory, a communication channel, a control policy – but make two departures forced by the LLM-agent setting. First, the agent's internal state – its backbone θ_i, private memory m_i, and role ρ_i – is not a structured belief base, but a frozen language model plus an opaque context window, so propositional commitments and inference rules over them are not available. Second, the central object of analysis shifts from agent rationality to trajectory-distribution properties, meaning that what one can measure about an LLM-agent system is its run-time behavior, not its proof-theoretic commitments. The classical literature's abstractions remain useful for labeling components, and the present formalism uses them for that purpose, not for the soundness theorems they were designed to support.

Multi-agent reinforcement learning and Dec-POMDPs. The trajectory-distribution formulation owes its shape to multi-agent reinforcement learning (MARL) and to decentralized partially observable Markov decision processes (Dec-POMDPs) [33, 34]. From those frameworks, we borrow the run-as-trajectory view, the role of a stationary control policy, and the use of expected cumulative reward over a budget as the run-time objective. We deliberately do not borrow three things. First, MARL/Dec-POMDP formulations assume a transition kernel $P(s' \mid s, a)$ that is accessible for planning or learning; LLM-agent systems act on world state through operation calls whose effects are not modeled, so the tuple treats S_{world} as black-box and the trajectory distribution $P_{\mathcal{M}}(\cdot \mid x)$ as the only object available. Second, MARL/Dec-POMDP fixes the agent population, action space, and observation space at $t = 0$; we make structural change a primary action class via Π_{meta}, covering SPAWN, GRANT, and CONSOLIDATE. Third, MARL/Dec-POMDP assumes a parameterized policy class amenable to gradient-based optimization; we permit this in principle but do not assume it. Most LLM-agent components are continuous in temperature and combinatorial in everything else. The closest contemporary relative is the literature on language-conditioned MARL (e.g., negotiation games and emergent communication), which shares the action-language interface but typically fixes the agent population and works in much smaller state spaces.

LLM-agent harnesses and surveys. The recent wave of LLM-agent harnesses – AutoGen, MetaGPT, CAMEL, ChatDev, AgentVerse, the Voyager skill-library line – has produced a large taxonomy of design patterns such as planner/critic splits, supervisor-researcher dialogues, role-prompted teams, blackboard architectures, skill libraries, evolutionary populations [14, 35–39]. A parallel line makes the harness itself the object of optimization. Some search over agent designs and workflows [8, 9, 23, 24]. Others evolve the cross-run store S_{cross} our tuple names, accumulating it as an itemized "playbook" [40] or co-evolving the store with the routine that maintains it [41]. Survey papers organize these systems along axes such as communication topology, memory scheme, coordination protocol, and evolution mechanism [22, 42]. Gao et al. organize self-evolving agents around what evolves, when it evolves, and how adaptation is governed, including models, memory, tools, architecture, intra-test-time versus inter-test-time adaptation, and scalar-reward or textual-feedback mechanisms [42]. Kim et al. take a complementary empirical route, fitting quantitative scaling relationships across coordination pattern, model capability, system factors, and task factors

while standardizing tools, prompts, and compute across hundreds of configurations [22]. Tie et al. survey this space under the name autoresearch and organize systems along a five-level human-to-AI autonomy spectrum, five workflow stages, and five evaluation dimensions [43]. Their axes measure how much of the workflow the AI controls, executes, and validates, whereas we compare the agent structure and coordination of the systems themselves. These surveys catalog what a system looks like, but none gives the coordinates one holds fixed or varies to attribute an outcome to a single design choice. The tuple supplies those coordinates, separating cases that survey-level labels merge and recovering distinctions already present in the literature (Section 7).

Autoresearch systems. We work through detailed examples later, which sit inside a larger autoresearch literature [13, 19, 27, 44, 45] (Section 6). These systems do not define their designs in shared, comparable terms. They define them functionally, in prose about research threads, branches, memory, and a correctness audit, with a benchmark number attached. We use the tuple to label those design choices. As an example, one recent system runs many long-horizon research threads and combines promising branches, a meta-control search π_{meta}. It also consolidates context across experiments, a cross-run store S_{cross} [7]. Most autoresearch systems share both coordinates. What sets this system apart is the third coordinate. It strengthens e's integrity against reward hacks inside the loop rather than fixing the evaluator in advance.

Reward hacking and Goodhart's law. The alignment literature has long studied what happens when an agent optimizes a proxy in place of the quality it should match. Amodei et al. identify reward hacking as a safety problem, and Krakovna et al. catalog specification gaming across systems that met a stated objective while violating its intent [46, 47]. Manheim and Garrabrant decompose Goodhart's law into regressional, extremal, causal, and adversarial variants, separating benign proxy drift from optimization that seeks the gap [48]. Skalse et al. define reward hacking formally and prove conditions under which a proxy is unhackable [49]. Gao et al. fit scaling laws for how true reward diverges from proxy reward as optimization pressure grows [50], and Pang et al. show the same divergence in conditional text generation [51]. Wang and Huang treat the same problem as an equilibrium, proving that an agent optimizing a finite-dimensional evaluation signal must under-invest in the quality dimensions that signal omits [52]. The shared finding is that the gap widens with pressure and is a property of the proxy, not the optimizer's intent. We make two departures this literature does not. First, we identify the proxy with a single component, the evaluator e, so reward hacking becomes a property of e rather than of the agents, and the distance from true quality is the proxy-quality gap (Section 2). Second, we tie optimization pressure to how aggressively the meta-control policy π_{meta} searches against e, so two systems with the same e but different pressure occupy different points in the vocabulary. We then use both components to identify each documented hack as a specific integrity property of e that failed (Section 5).