CosmoEvolve

A Virtual Research Lab as a Hierarchical Multi-Agent System

Licong Xu[1] and Claude Code[2]

[1]Kavli Institute for Cosmology, University of Cambridge, Cambridge, UK
[2]Anthropic, San Francisco, CA, USA

April 11, 2026

Abstract

We present CosmoEvolve, an open, domain-general framework that instantiates a virtual research laboratory, consisting of one principal-investigator (PI) agent and a community of student scientist agents, inside a single Python process. Unlike fixed research pipelines, CosmoEvolve leaves the ordering of scientific actions emergent: at every round the PI observes a summarised lab state and selects an action from a finite discrete action space $\mathcal{A}_{\text{PI}}$ with six elements (group meeting, individual meeting, task assignment, paper request, symposium, and wrap-up), while students execute the selected action through a tool-calling LLM loop that delegates concrete work to read-only and write-enabled subagents. We describe each CosmoEvolve agent as a tuple $(\mathcal{L}, \mathcal{C}, \tau, \pi, \mathcal{A}, \mathcal{S})$ of LLM backbone, context, tool set, policy, action space, and internal state; describe the two main operating modes (the bounded lab session and the asynchronous lab-continuous mode with a PI thread and N parallel student threads synchronised by per-student locks); and present the collaboration primitives (a shared discussion thread acting as a blackboard, sequential group-meeting rollouts, parallel peer review, a shared artifact store, and cross-session memory and skill evolution) that turn the collection of agents into a laboratory that improves itself across runs. We further document the system's tool management, context engineering, and agent visibility architecture in sufficient detail to be reproduced.

1 Introduction

Large language models (LLMs) have matured into the backbone of general-purpose autonomous agents, driven by tool-use formulations such as ReAct [26], Toolformer [21], and the function-calling interfaces exposed by contemporary LLM providers [1, 15]. Building on these primitives, several multi-agent frameworks, including AutoGen [25], MetaGPT [7], CAMEL [10], and ChatDev [19], have shown that role-playing collections of LLMs can tackle substantially harder tasks than a single agent, and *Generative Agents* [16] demonstrated the value of persistent memory and emergent routines over long horizons.

Existing LLM-agent frameworks for scientific research typically fall into two categories. *Fixed pipelines*, such as AI Scientist [12], follow a predetermined sequence (ideation, investigation, writing) and are therefore predictable but cannot adapt the ordering of actions to what is learned at each step. *Flat multi-agent systems* preserve this adaptivity but incur a substantial cost: each agent holds its own LLM session and orchestration is carried out through external message buses whose reliability and cost are difficult to reason about.

CosmoEvolve is designed to recover the emergent structure of a research laboratory while retaining the cost profile of an embedded pipeline. It runs one PI and N student scientists inside a single Python process, with a shared budget, a shared artifact store, a shared memory tree, and

a shared discussion transcript. The PI is an LLM policy that, at every round, receives a compact summary of the laboratory state and emits a structured SupervisorDecision selected from a finite discrete action space, in the spirit of classical Markov decision processes [23]. Students are hierarchical tool-using agents that delegate real work to inexpensive read-only subagents (`explore` and `plan`) and to a single strong write-enabled subagent (`code`). The self-improvement loop, consisting of the promotion of recurring learnings into per-agent `MEMORY.md` files and the evolution of new skills from past failures, follows the principles of REFLEXION [22], SELF-REFINE [13], and the lifelong skill library of VOYAGER [24]. The framework has been used for cosmology and astronomy workloads, but nothing in the core implementation is astronomy-specific: skills, tools, and roles are plug-in Markdown and Python modules.

Contributions and organisation. This paper provides a formal design description of COSMOEVOLVE. Section 2 gives a high-level system view; Section 3 introduces the agent tuple and describes each agent class; Section 4 formulates the lab mode as a discrete-time Markov decision process; Section 5 describes the asynchronous LAB-CONTINUOUS mode; Section 6 details the collaboration primitives; and Section 7 documents the engineering substrate (tool management, context engineering, and visibility). Section 8 discusses known limitations and future directions. Supplementary algorithms, code listings, and per-agent input specifications are collected in the appendices.

2 System overview

A COSMOEVOLVE run is organised as a three-tier stack (Figure 1). A `RunOrchestrator` reads configuration files, wires scientists and the shared budget, loads skills, and creates a `VirtualLab` that owns the PI policy π_{PI} and the mutable LabState. The `VirtualLab` delegates scientific actions to a community of `Scientist` coordinators. Each coordinator dispatches concrete work to one of three subagents: `explore` (read-only data and literature inspection), `plan` (read-only design of an implementation), and `code` (write-enabled implementation and execution).

The PI never writes files and never executes code directly; students never execute code directly either, and instead delegate every side-effectful operation to the `code` subagent. This three-tier organisation concentrates write privileges at the leaves, following the principle of least privilege [20], and bounds the blast radius of any single incorrect LLM completion.

Three operating modes reuse the same `VirtualLab` implementation. The LAB mode runs a single bounded session of at most `max_rounds` PI decisions. The LAB-TICK mode advances the embedded lab by exactly one persisted PI decision per invocation and is intended for cron-triggered bursts. The LAB-CONTINUOUS mode runs an unbounded loop with the PI and the N students in separate threads of a single Python process, synchronised only through per-scientist locks. All three modes share one action executor, one memory-and-skill evolution pipeline, and one budget subsystem.

3 Agents as components

We model each agent as a tuple

$$A \;=\; (\,\mathcal{L},\ \mathcal{C},\ \tau,\ \pi,\ \mathcal{A},\ \mathcal{S}\,), \tag{1}$$

where $\mathcal{L}$ is the LLM backbone, $\mathcal{C}$ is the context constructed at each step, τ is the tool set (JSON schemas and a dispatcher $\tau : \text{name} \mapsto \text{callable}$), π is the policy that maps the LLM output to the next action, $\mathcal{A}$ is the action space, and $\mathcal{S}$ is the agent's persistent state. The atomic building block of the framework is `BaseAgent`, a tool-calling loop that implements the inner control of a single LLM equipped with a tool set; the pseudocode is given in Appendix A.

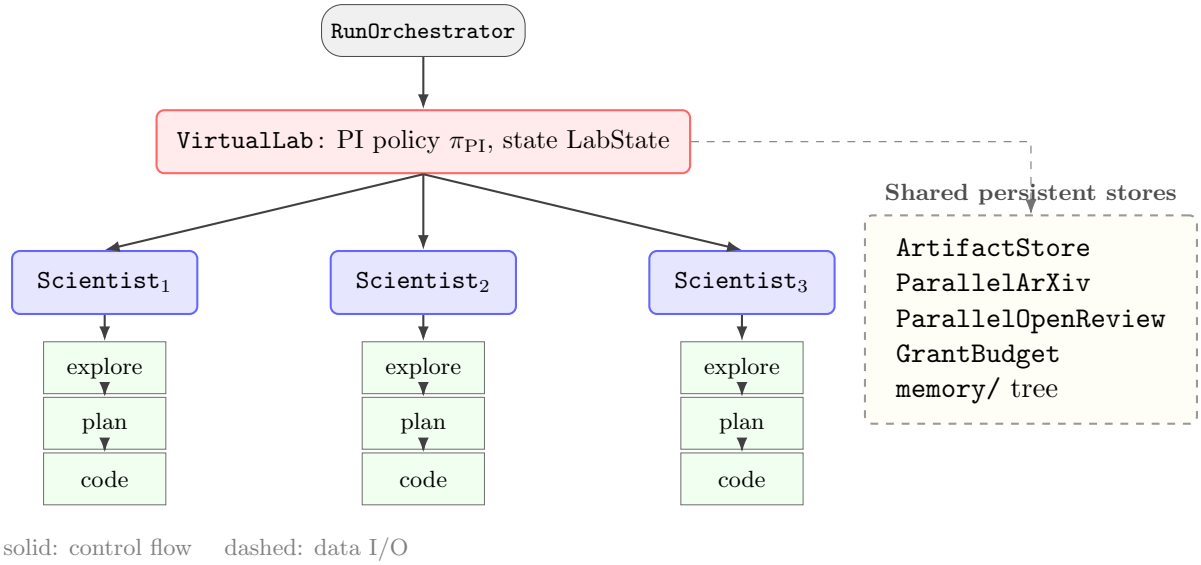

Figure 1: Three-tier architecture. The run orchestrator assembles configuration, budget, and skills into a virtual lab that owns the PI policy and the shared LabState. Each scientist coordinator dispatches work vertically to its own `explore`, `plan`, and `code` subagents. Shared persistent stores (right) are reached through explicit tool calls or the lab's publishing actions.

3.1 The student scientist

A `Scientist` wraps a `BaseAgent` with an identity, a skill set, a memory, and a `BudgetAccount`. Its LLM backbone is an OpenAI-compatible `LLMClient` instantiated from `cosmoevolve.yaml`.

Its context $\mathcal{C}_{\text{Sci}}$ is a system prompt assembled from five components: (i) a first-person identity statement; (ii) a workflow contract that mandates the `explore`→`plan`→`code` delegation pattern and forbids direct code execution; (iii) a skill index, produced by `build_skill_index`, that lists the names and one-line descriptions of loaded skills but not their full bodies; (iv) a persistent memory block from the student's own `MEMORY.md`, truncated to 4000 characters; and (v) a block containing the ten most recent anti-patterns from the student's `errors.jsonl`.

Its tool set τ_{Sci} is a filtered allowlist. The coordinator has read-only access to the workspace, literature search, logging, and note-taking tools, plus the `dispatch_subagent` tool. All side-effectful tools (`run_python`, `save_to_project`, `run_project_code`, `create_project`) are absent from this allowlist and reachable only by dispatching the `code` subagent. Its policy is the `BaseAgent` tool-calling loop with no fixed iteration cap; it terminates when the coordinator emits a plain assistant message or when the budget is exhausted. Its action space therefore comprises the read-and-log tools and the three dispatch variants shown in Table 1. Its internal state consists of the per-student `MEMORY.md`, `learnings.jsonl`, and `errors.jsonl`; a `BudgetAccount`; and a `threading.Lock` used to serialise concurrent `run_task` invocations.

Role	LLM tier	Writes?	Tool allowlist
`explore`	cheap / local	no	read and list workspace, fetch URL, search arXiv, read skill
`plan`	cheap / local	no	same as `explore`
`code`	strong	yes	`run_python`, `save_to_project`, `run_project_code`, `git_commit`

Table 1: The three subagent roles dispatched by a student coordinator.

Each subagent is itself a `BaseAgent` with a hard tool allowlist enforced by its factory, the parent

coordinator's skills inherited verbatim, and a fresh transcript per dispatch. Subagents cannot call one another; only the coordinator orchestrates them.

3.2 The PI supervisor

The PI is an LLM policy that decides which scientific action should be taken next. It lives in `VirtualLab` and holds a dedicated `LLMClient` tagged `_caller_name="PI"`. Its observation at round k is

$$o_k \;=\; \mathrm{summary}(s_k) \;\sqcup\; \mathrm{pi\_skills} \;\sqcup\; \mathrm{pi\_memory} \;\sqcup\; e_k, \tag{2}$$

where $\mathrm{summary}(s_k)$ is an $O(1)$-size summary of the state $s_k \in \mathcal{S}$ containing the topic, round number, per-type counts (ideas, tasks, papers, reviews), and the last ten messages of the shared discussion thread; `pi_skills` is the PI skill index; `pi_memory` is the PI's own `MEMORY.md` truncated to 4000 characters; and e_k is an optional block containing the accepted-paper counter (when a stop criterion is set) together with an optional research brief produced by a preceding tool-using research pass, truncated to 12000 characters. Each PI round therefore runs in two passes: a research pass, executed as a `BaseAgent.run` with a read-only tool set, and a decision pass, executed as a single structured call

$$\pi_{\mathrm{PI}}(o_k) \;=\; \mathrm{SupervisorDecision}\big(\mathrm{action},\ \mathrm{target},\ \mathrm{topic},\ \mathrm{reasoning}\big), \tag{3}$$

whose output schema is enforced by Pydantic [17]. When a structured call fails to parse, `VirtualLab` falls back to a default SupervisorDecision with action = GROUP_MEETING so that the loop does not terminate on a malformed completion. The PI action space is the finite set

$$\begin{aligned}\mathcal{A}_{\mathrm{PI}} = \{\ &\textsc{group\_meeting},\ \textsc{individual\_meeting},\ \textsc{assign\_task},\\ &\textsc{request\_paper},\ \textsc{call\_symposium},\ \textsc{wrap\_up}\ \}.\end{aligned} \tag{4}$$

Unlike a student, the PI does not modify the world directly. It produces a decision, and the action executor in `VirtualLab` implements the transition. The PI's persistent state consists of a `BudgetAccount`, a dedicated `AgentMemory`, and a view onto LabState mediated by $\mathrm{summary}(\cdot)$.

4 Lab mode as a discrete-time MDP

We model a single lab session as a tuple [18, 23] $(\mathcal{S}, \mathcal{A}_{\mathrm{PI}}, T, T_{\mathrm{term}}, s_0)$, where $\mathcal{S}$ is the state space, $\mathcal{A}_{\mathrm{PI}}$ is the PI action space from Equation 4, $T : \mathcal{S} \times \mathcal{A}_{\mathrm{PI}} \to \mathcal{S}$ is the transition function implemented by the action executor, and $T_{\mathrm{term}} : \mathcal{S} \to \{0, 1\}$ is the termination predicate. There is no explicit reward: the planner π_{PI} is an LLM conditioned on o_k, not the output of a value-iteration procedure.

State. The state $s \in \mathcal{S}$ is an instance of LabState whose fields include the topic; the round counters; the shared discussion thread, a list of LabMessage entries acting as a blackboard; the lists `ideas_proposed`, `tasks_assigned`, `papers_written`, and `reviews_done`; and the termination flags `finished` and `kickoff_done`. Each LabMessage carries a `speaker_id`, a `speaker_name`, a `content` field, and a `message_type` drawn from the set

$$\{\,\texttt{discussion},\ \texttt{finding},\ \texttt{question},\ \texttt{decision},\ \texttt{presentation}\,\}.$$

The full dataclass is given in Appendix B.

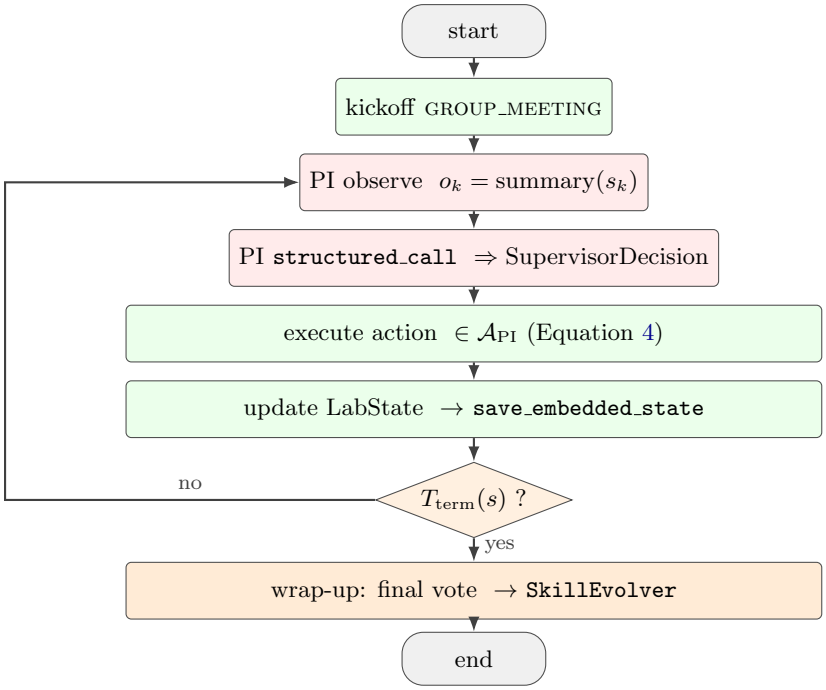

Figure 2: Control flow of `VirtualLab.run`. Each round the PI builds observation o_k, emits a structured SupervisorDecision, and dispatches to the action executor. After each executor mutates LabState and the state is checkpointed, the termination predicate T_{term} of Equation 6 decides whether control returns to the observation node (the outer loop on the left) or proceeds to wrap-up, final voting, and `SkillEvolver`.

Transition. On entry to `VirtualLab.run`, the state is initialised with an empty thread and a kickoff GROUP_MEETING is fired. The main loop then alternates between observation, decision, and execution:

$$s_{k+1} \;=\; T\big(s_k,\; \pi_{\text{PI}}(\text{summary}(s_k))\big), \qquad k = 0, 1, 2, \ldots \tag{5}$$

Each transition appends to the discussion thread, may mutate `ideas_proposed`, `tasks_assigned`, `papers_written`, or `reviews_done`, deducts tokens from the relevant `BudgetAccount`, and persists the updated state atomically to `embedded_lab_state.json`. The detailed per-action semantics are collected in Appendix C. Figure 2 summarises the control flow.

Termination. A session terminates when any of the following predicates hold:

$$T_{\text{term}}(s_k) = 1 \iff s_k.\text{finished} \vee k \geq s_k.\text{max\_rounds} \vee \#\{\text{accepted papers in } s_k\} \geq N_{\text{stop}}, \tag{6}$$

where N_{stop} is the optional `stop_after_accepted_papers` target. When the accepted count crosses the threshold, `VirtualLab` calls WRAP_UP automatically and returns a tick-info dictionary with `stop_criterion_met=True`.

The persisted single-step primitive. `RunOrchestrator.run_lab_tick` exposes a persisted single-step version of the transition that loads `embedded_lab_state.json`, advances the lab by exactly one unit (the kickoff meeting on a fresh state, or one PI decision on an existing state), and writes the state back atomically. The procedure, `advance_one_unit`, is given in Appendix D. The LAB-TICK mode is the primitive from which the asynchronous continuous mode of Section 5 is built.

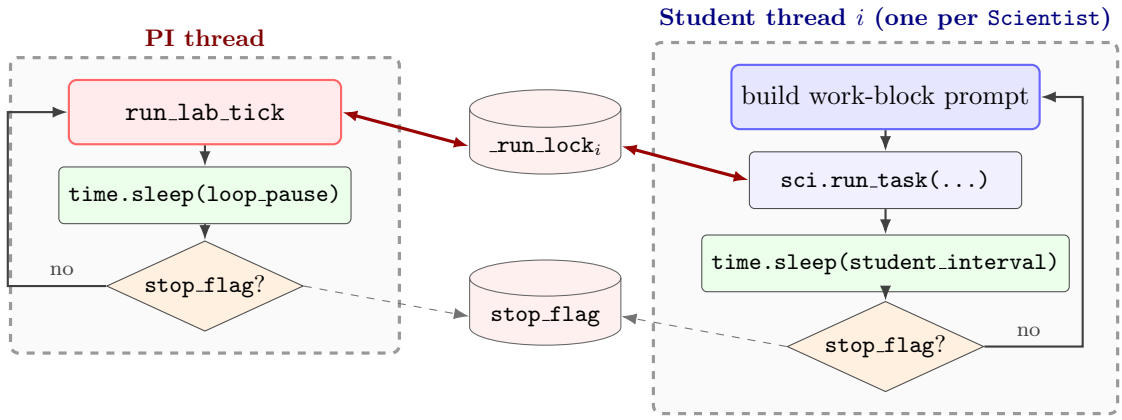

Figure 3: Asynchronous threading model of LAB-CONTINUOUS. The PI thread (left) and each student thread (right) run independent infinite loops. Synchronisation is provided only by a per-scientist `_run_lock`: when a PI decision touches scientist i via ASSIGN_TASK, REQUEST_PAPER, or a meeting action, the PI's call path acquires that scientist's lock and blocks until the student's current work block completes. The shared `stop_flag`, a `threading.Event`, is monitored at the end of every iteration. Red bidirectional edges denote lock acquisition; dashed edges denote non-blocking reads of `stop_flag`.

5 Lab-continuous mode

The LAB-CONTINUOUS mode is designed for unattended continuous operation. It is not implemented as a single monolithic loop; rather, it consists of two loosely coupled loops, one for the PI and one per student, running in separate threads of a single Python process. The PI loop invokes `run_lab_tick` at every iteration and then sleeps; each student loop runs independent work blocks with its own thread and its own per-block prompt. Figure 3 shows the resulting structure.

PI loop. The PI thread repeats the sequence (`run_lab_tick` → `sleep` → `check stop_flag`) until the flag is set. Because each tick goes through `run_lab_tick`, every PI decision is persisted to `embedded_lab_state.json`. When $\#\{\text{accepted papers}\} \geq N_{\text{stop}}$ or when `max_rounds` is reached, the embedded state file is unlinked and, unless `restart_after_stop_criterion` is true, `stop_flag` is set and the loop exits.

Student loop. Each student runs an independent infinite loop of work blocks. At round i the student is given a structured prompt (Appendix E) containing a lab-context statement, an optional shared-project block, a reminder that the skills are already present in the system prompt, and a six-step workflow that requires at most one `explore` dispatch, one `plan` dispatch, one `code` dispatch, and a final `submit_pi_update`. The prompt is passed to `sci.run_task`, which acquires the student's `_run_lock` before entering the `BaseAgent.run` loop. Exceptions raised inside a work block are logged and suppressed so that the failure of one student does not terminate the others.

Synchronisation. The only synchronisation primitive between the PI thread and the student threads is the per-student `_run_lock`. When a PI decision targets scientist i via ASSIGN_TASK, REQUEST_PAPER, or a meeting action, the executor calls `sci.run_task`, which attempts to acquire the same lock. If the student is inside a work block, the PI call blocks until the current block terminates. There is no preemption and no gather flag; work blocks always run to completion. For read-only introspection, a non-blocking `busy` property exposes `self._run_lock.locked()`.

Persistence and recovery. Two files form the persistent state of a continuous run. `workspace/embedded_lab_state.json` contains the serialised LabState, written atomically via the temporary-file, `fsync`, and `os.replace` pattern, with corrupted files quarantined to a `.corrupted` sibling. `workspace/budget.json` contains the `GrantBudget`, saved after every PI tick and after every student work block. Together these two files allow a continuous run to survive process restarts without losing either its place in the session or its budget history.

6 Collaboration mechanisms

CosmoEvolve mediates inter-agent communication through a small number of explicit channels; there is no direct agent-to-agent messaging.

Shared discussion thread. The primary channel is the `discussion_thread` field of LabState, a mutable list of LabMessage entries acting as a blackboard in the classical sense of Erman et al. [4] and Hayes-Roth [6]. Every PI action and every scientist response appends a message tagged with a `message_type`. On each observation, the PI reads only the final ten messages; older entries remain in s for checkpointing but are not exposed to π_{PI}.

Group meeting as a sequential rollout. The GROUP_MEETING action is implemented as a sequential rollout rather than a parallel broadcast: each scientist speaks in turn and observes the contributions of the scientists that preceded it. For $N = 3$,

$$m_1 = \pi_{\text{Sci}_1}(\text{thread}), \tag{7}$$
$$m_2 = \pi_{\text{Sci}_2}(\text{thread} \cup \{m_1\}), \tag{8}$$
$$m_3 = \pi_{\text{Sci}_3}(\text{thread} \cup \{m_1, m_2\}), \tag{9}$$

which preserves the incremental structure of a research discussion at the cost of exactly N LLM calls per meeting.

Individual meetings and tasks. The INDIVIDUAL_MEETING action is a two-turn exchange between the PI and a single student. The ASSIGN_TASK action triggers the full student `BaseAgent.run` over its coordinator tool set, which in turn dispatches to the `explore`, `plan`, and `code` subagents. The student reports back via `submit_pi_update`, which appends a LabMessage of type `finding` to the thread.

Paper submission, review, and voting. REQUEST_PAPER uses a structured LLM call, with a Pydantic `Paper` schema as the response model, to obtain a paper containing title, abstract, and sections; the submission is pushed to `ParallelArXiv` and mirrored in the shared store. CALL_SYMPOSIUM invokes `ParallelOpenReview` with $R = 2$ reviewers per paper, running reviews concurrently through a `ThreadPoolExecutor`. Each reviewer emits a Pydantic `ReviewScore` containing strengths, weaknesses, a numerical score in $[1, 10]$, and a recommendation. The mean score drives the `accept` / `reject` decision; this design mirrors the multi-agent debate and review protocols of Du et al. [3] and Liang et al. [11]. The decision is posted to the thread and logged in the author's `AgentMemory` as a structured learning, which then feeds the self-improvement pass at the end of the session. A final vote over accepted papers is conducted at wrap-up.

Memory and skill evolution. At the end of each session, `SkillEvolver` is run against the aggregated memory of all scientists and the PI. It gathers recurring failures (recurrence ≥ 2), proven patterns (recurrence ≥ 3 with successful outcome), and unresolved errors of severity at least `degrading`, deduplicates them, and produces up to three new `skills/learned-*/SKILL.md` files per session via a structured call. In the following session these files are loaded automatically

by `load_skills_for_role`, so that methodology mistakes made in session n become documented skills injected into every agent's system prompt in session $n+1$. Within a single session, a `MemoryExtractor` attached to each student `BaseAgent` fires whenever $\Delta_{\text{tokens}} \geq 5000$ and $\Delta_{\text{tool\_calls}} \geq 3$ since the last extraction; its entries are appended to `learnings.jsonl` and `errors.jsonl`, and recurrence counting promotes stable learnings into `MEMORY.md`. The extractor calls a dedicated cheap-tier `LLMClient`, so its cost does not accumulate in the paid budget.

Together, these two processes form the CosmoEvolve self-improvement loop, conceptually related to the verbal-feedback updates of Reflexion [22] and the lifelong skill library of Voyager [24], but applied at the laboratory level: one session's evolved skills are shared across every agent of the next session.

7 Engineering substrate

CosmoEvolve's behaviour depends as much on its engineering substrate as on its formal agent definitions. This section documents three aspects that are essential for correctness and cost control: tool management, context engineering, and the visibility architecture.

7.1 Tool management

Every agent in CosmoEvolve carries a tool set τ that is a strict allowlist: the dispatcher refuses any tool name not present in its dictionary, even when the tool is implemented elsewhere in the package. Four allowlists coexist in a running lab (the PI research pass, the student coordinator, the `explore` and `plan` subagents, and the `code` subagent), and the write-and-execute tools exist in exactly one location: the `code` subagent. A misclassified PI decision cannot overwrite a file, and a misbehaving `explore` subagent cannot modify the repository.

On top of the static allowlist, the framework enforces per-work-block quotas through mutable counters attached to the parent `LLMClient`:

```
sci.llm._explore_calls_this_run = 0
sci.llm._plan_calls_this_run    = 0
sci.llm._code_calls_this_run    = 0
```

The `dispatch_subagent` tool reads these counters on entry and refuses the dispatch once a role-specific cap is hit. In lab-continuous, `RunOrchestrator` resets the counters at the top of every student work block. The default cap $n_{\texttt{explore}} \leq 1$ per block was chosen after empirical observation of repeated `explore` dispatches in situations where the coordinator should have committed to `plan`→`code`.

Resources that cannot be recovered by the memory loop are capped at the tool itself. `create_project` enforces a workspace maximum of five projects; `run_python` and `run_project_code` run under a three-hour default wall-clock timeout (`COSMOEVOLVE_EXEC_TIMEOUT`); and every `llm.chat` call deducts `last_usage_tokens` from a `BudgetAccount`, so that budget exhaustion is a first-class termination condition.

Skills as tools. The skill library is exposed through a `read_skill` tool rather than as a fixed prompt prefix. Agents receive only a compact index in their system prompt (Section 7.2) and pay the full-body cost only for the specific skill required by the current task. This matches the retrieval-augmented prompting pattern of Wang et al. [24] applied inside a long-running agent.

7.2 Context engineering

Long-running agents face an inherent tension between the information that would be useful and the information that can be afforded by the context window. CosmoEvolve resolves this tension in three places.

Layered system prompt. The student coordinator's system prompt is assembled from the five layers described in Section 3.1. Layers (iii)–(v) replace an earlier design that embedded every skill body and every historical learning in the system prompt and produced a per-call prompt cost of several thousand tokens for material that most tasks did not consult. The skill index costs $O(\#\text{skills})$ tokens in names, while full bodies are loaded lazily and only on explicit `read_skill` calls. The persistent `MEMORY.md` is truncated to 4000 characters and the anti-patterns block is limited to the ten most recent entries, so that the system-prompt cost is bounded regardless of how large the underlying memory files become.

Bounded PI observation. The PI never sees the full discussion thread. Its observation is the output of summary_for_supervisor(s), which contains the current round number, per-type counts, and only the last ten messages. An optional research brief produced by a preceding research pass is further capped at 12000 characters. These caps convert a potential $O(k)$ growth in PI-observation tokens into a bounded $O(1)$.

Transcript compression preserves tool-call threading. Inside a single `BaseAgent.run`, the conversation grows with every tool call. The routine `_compress_old_messages` is invoked before the next `llm.chat` call whenever the accumulated prompt would cross the model's context window. The routine obeys one hard invariant: it must never break the pairing between an assistant tool-call message and its subsequent tool-result message, as required by OpenAI-style chat formats. The compressor therefore drops whole tool-call / tool-result blocks from the oldest turns, trims the tail of assistant messages rather than the middle, and preserves the most recent turns. Linear per-call token growth is treated as a correctness defect rather than as an optimisation target.

Tiered inference. Expensive inference (code writing, idea generation, paper drafting, and the PI's structured decision) is routed to a strong `LLMClient`. Low-stakes bookkeeping inference (auto-memory extraction, routing, summarisation, and the `SkillEvolver` call) is routed to a dedicated cheap-tier `LLMClient` so that it does not consume the paid budget. Cost tracking is unified across tiers by tagging each call with a `_caller_name` such as `memory-extract/student-1`, so that the ledger still attributes every token to its originating agent.

7.3 Visibility architecture

A defining property of CosmoEvolve is that each LLM call receives a deliberately restricted view of the running system. Agents do not share a single monolithic context; each call receives a constructed slice of the whole, and information flows between agents only through a small number of explicit channels. Each CosmoEvolve LLM call consists of exactly three inputs: (i) a system prompt built at call time from the agent's identity, skills, and memory; (ii) a user-turn message carrying the concrete task or observation for that one call; and (iii) the running chat transcript of the current `BaseAgent.run`. Nothing beyond these three inputs is passed to the model.

Table 2 summarises the visibility matrix. The per-agent enumeration of what is included in each of the three inputs, and what is not reachable at all, is given in Appendix F. The only inter-agent channels within a session are (i) task strings passed from the PI to a student and from

a coordinator to a subagent, (ii) return messages flowing back up the same call stack, (iii) the shared `ArtifactStore`, and (iv) the shared paper archive and review system. Between sessions, two additional channels open: the promotion of `learnings.jsonl` and `errors.jsonl` into `MEMORY.md`, and the session-level `SkillEvolver` pass that writes `skills/learned-*/SKILL.md` files.

Input channel	PI	Student	explore / plan	code
Own `MEMORY.md` (capped)	✓	✓	inherited	inherited
Own `errors.jsonl` (last 10)		✓		
Own skill index	✓	✓	inherited	inherited
Full skill bodies	on demand	on demand	on demand	on demand
LabState summary (last 10 msgs)	✓			
LabState full thread				
Other students' `MEMORY.md`				
Running work-block transcript	own	own	own	own
Parent or sibling transcripts				
Artifact store content		tool-gated	tool-gated	tool-gated
Read-only tools	✓	✓	✓	✓
Write-and-execute tools				✓
`dispatch_subagent`		✓		

Table 2: Visibility matrix. "✓" indicates that the information is included directly in the agent's LLM input; "tool-gated" indicates that the agent can retrieve the information only by issuing an explicit read tool call; "inherited" indicates that the parent student's value is copied into the subagent's system prompt at dispatch time; a blank cell indicates that the information is not reachable from that agent.

This strict separation, in which task strings are self-contained and LabState is invisible to the workers, is what permits the entire laboratory to run in a single Python process without agents accidentally leaking information into one another's contexts.

8 Discussion

8.1 Limitations

Absence of a learning signal at the policy level. Because π_{PI} is an LLM conditioned on o_k with no explicit reward, the system's long-horizon behaviour is sensitive to the prompt and to the underlying model. Three recurring failure modes have been observed: over-reliance on GROUP_MEETING in place of ASSIGN_TASK; delayed invocation of CALL_SYMPOSIUM, leaving papers unreviewed for many rounds; and premature WRAP_UP under tight budgets. Mitigation is currently indirect and relies on the session-level `SkillEvolver` emitting corrective meta-skills after recurrent stalls are observed.

Lossy memory extraction. Running the `MemoryExtractor` on the cheap tier keeps cost low but produces noisier learnings, occasionally including generic advice that subsequently pollutes the next session's system prompt. The deduplication layer limits duplication but the framework currently lacks a mechanism for evicting a previously promoted learning that was later found to be incorrect.

High-variance peer review. With $R = 2$ reviewers per paper and single-float scores, the accept / reject decision is sensitive to the stochasticity of individual LLM calls. Repeated invocations of CALL_SYMPOSIUM on the same papers can yield different decisions. Formal

multi-agent debate protocols in the style of Du et al. [3] or Liang et al. [11] would reduce this variance but have not yet been integrated into the review cycle.

Single-process failure domain. The embedded `VirtualLab` runs in a single Python process. Although `embedded_lab_state.json` and `budget.json` persist a session across process restarts, in-flight student work blocks do not: a crash that occurs mid-`BaseAgent.run` loses the tool-call transcript of that block. Durable-execution designs would remove this failure mode at the cost of abandoning the single-process simplicity that motivates the current implementation.

Sequential bottleneck within group meetings. Although the student threads parallelise ASSIGN_TASK and REQUEST_PAPER, GROUP_MEETING is sequential by construction. For large communities this becomes the dominant per-round cost. Current deployments use $N = 3$ students, for which the sequential rollout remains inexpensive.

Heuristic context compression. The transcript compressor drops whole tool-call blocks from the oldest turns but does not prioritise content by semantic relevance. A task whose critical context lies many turns in the past can lose it to compression even when budget and context window could, in principle, have afforded to retain it.

Skill accumulation. Because `SkillEvolver` may emit up to three new skills per session, and skills are never pruned, long deployments accumulate a large number of `learned-*` skills. The lazy skill-index mechanism bounds the system-prompt cost, but a nontrivial fraction of learned skills end up unused, and skill names can collide with earlier entries.

Static tool allowlists. Tool allowlists are hard-coded in Python factory functions. Adding a new subagent role requires editing the package, which limits the degree to which a user can customise a lab without modifying the source tree.

8.2 Future directions

Several directions follow naturally from the limitations above.

Reward-signal-based PI training. A natural next step is to treat the accept / reject signal from CALL_SYMPOSIUM and the final vote at wrap-up as a sparse reward and to fine-tune a small fast model as the PI policy via offline reinforcement learning [23] or reward-model distillation. This would complement the existing skill-evolution loop with an explicit gradient and would convert the current prompt-level learning into parameter-level learning.

Debate-based peer review. Replacing the current independent two-reviewer protocol with a multi-round multi-agent debate in the style of Du et al. [3] and Liang et al. [11] should reduce the variance of the accept / reject decision and produce richer feedback to `SkillEvolver`.

Structured memory. Upgrading `MEMORY.md` from a linear document to a structured memory, such as a knowledge graph of entities and concepts or a vector store of transcript chunks indexed by task type, would allow the PI to retrieve only the portion of history relevant to the current decision, and would support explicit forgetting of stale learnings.

Skill pruning and merging. A counterpart to `SkillEvolver` that periodically scores learned skills by the frequency with which they are loaded via `read_skill`, and merges overlapping ones, would keep the skill library bounded under indefinite operation.

Heterogeneous communities. The current default of three general-purpose students could be generalised to specialised roles (theorist, experimentalist, data engineer, referee) with role-specific tool allowlists and skill bundles, enabling a division of labour closer to that of a real laboratory, in the spirit of MetaGPT [7] and ChatDev [19].

Grounded external review. The local `ParallelArXiv` and `ParallelOpenReview` provide a self-contained review loop but no grounding in real literature. Integration with real arXiv search and a reproducible notebook-execution reviewer in the spirit of Lu et al. [12] would render accept / reject decisions less circular.

Distributed and durable execution. Scaling beyond one Python process, for instance by running students across machines with a durable event log, would allow lab-continuous to survive hardware failures and to support much larger communities. The central design question is how to retain the ergonomics of a single embedded process while introducing this durability.

Cost- and progress-aware scheduling. At present the only termination controls are `max_rounds` and `stop_after_accepted_papers`. A richer scheduler could trade off the marginal cost of another round against the expected progress, measured for instance by the change in mean review score, and could promote a student from the cheap tier to the strong tier during a productive sequence.

Formal correctness gates on generated code. The `code` subagent writes and executes code, but `run_project_code` currently reports only pass or fail with a traceback. Mandatory typed tests, property-based checks, or static analysis gates before a task is marked successful would improve the quality of artifacts promoted to the shared store.

9 Conclusion

CosmoEvolve implements a research laboratory as a hierarchical multi-agent system in a single Python process. A PI policy π_{PI} selects one of six discrete scientific actions per round over a shared LabState; student coordinators execute those actions by delegating to inexpensive read-only subagents and a single strong write-enabled subagent; collaboration is mediated by a shared discussion thread, a shared artifact store, a paper archive, and a parallel peer-review system. The same building blocks support a bounded lab session, a persisted lab-tick primitive suitable for cron bursts, and an asynchronous lab-continuous mode in which the PI and each student run in their own thread and are synchronised only by per-student locks. A memory extractor and a session-level skill evolver close the loop: each session's successes and failures are compiled into skills that appear in the following session's system prompts. The resulting framework admits a clean formal description as a collection of tool-using agents, while the overall system behaves as a laboratory that improves itself across runs.

Acknowledgements

The authors thank the open-source communities behind `pydantic`, `FastMCP`, and the many OpenAI-compatible LLM providers used by CosmoEvolve. L.X. acknowledges support from the Kavli Institute for Cosmology, Cambridge. Large portions of the package code, including early drafts of this manuscript, were co-developed with Anthropic's Claude family of models.

A `BaseAgent` tool-calling loop

Algorithm 1 `BaseAgent.run`: the inner tool-calling loop.

$c \leftarrow$ system prompt $\cup$ user task
$k \leftarrow 0$
while $k < K_{\max}$ **do**
  $m \leftarrow \mathcal{L}$.chat(c, tools $= \tau$)
  budget.deduct($\mathcal{L}$.last_usage_tokens)
  if m.tool_calls is non-empty **then**
    for all call $\in m$.tool_calls **do**
      $r \leftarrow \tau$[call.name](**call.args**)
      append ToolResult(r) to c
    end for
    $k \leftarrow k + 1$
  else
    return AgentResponse(success, m.content, c)
  end if
end while
return AgentResponse(success = False, error = max_iterations)

Students use $K_{\max} = 64$; the PI coordinator pass is uncapped. At five iterations remaining, a system message is inserted instructing the agent to conclude its reasoning and submit findings.

B `LabState` dataclass

```python
@dataclass
class LabState:
    topic: str
    cycle_id: str
    discussion_thread: list[LabMessage]    # shared blackboard
    ideas_proposed: list[dict]              # {author, content, round}
    tasks_assigned: list[dict]              # {scientist_id, task, round}
    papers_written: list[str]               # paper ids
    reviews_done:  list[str]
    round_number:  int
    max_rounds:    int
    finished:      bool
    kickoff_done:  bool
```

Each LabMessage carries `speaker_id`, `speaker_name`, `content`, `message_type`, and a UTC timestamp. The state is mutated only by the PI thread and persisted to disk after every transition.

C Per-action transition semantics

The action executor `VirtualLab._execute_decision` dispatches over d.action:

group_meeting(*topic*)**.** For each scientist $i \in 1{:}N$ in order, the scientist is prompted with the current topic and the final five messages of the thread, and produces a first-person response m_i. Every m_i is appended to the thread as a LabMessage of type `discussion`. Messages matching proposal heuristics are promoted to `ideas_proposed`.

individual_meeting$(i, \textbf{topic})$**.** A two-turn exchange. The PI question is logged as type `question` and the scientist's answer as type `finding`.

assign_task$(i, \textbf{task})$**.** The task is appended to `tasks_assigned`, and `run_task(task)` is invoked on $\texttt{Scientist}_i$. This acquires the scientist's `_run_lock`, runs the full coordinator `BaseAgent.run` with its subagent tree, stores the result as an `Artifact` in the shared store on success, and appends the closing summary to the thread.

request_paper$(i, \textbf{topic})$**.** $\texttt{Scientist}_i$`.write_paper` is called, which runs `llm.structured_call(response_model=Paper)`. The produced `Paper` is wrapped in a `PaperSubmission` and pushed to `ParallelArXiv`; the paper identifier is appended to `papers_written` and the paper is stored as an artifact.

call_symposium$(\textit{topic})$**.** Every paper in `papers_written` is reviewed by $R = 2$ reviewers drawn from the community, excluding the author. Reviews are produced concurrently through a `ThreadPoolExecutor`. Each reviewer emits a Pydantic `ReviewScore`, and the mean score drives the accept / reject decision

$$\text{decision}(p) = \begin{cases} \texttt{accept} & \text{if } \frac{1}{R}\sum_{r=1}^{R} \text{score}_r(p) \geq \theta, \\ \texttt{reject} & \text{otherwise.} \end{cases}$$

Decisions are appended to the thread and logged to the author's memory.

wrap_up$(\textit{reason})$**.** Sets s.finished $\leftarrow$ `True`. After the main loop exits, the final-vote procedure runs over accepted papers and `SkillEvolver.evolve_from_all_students` is invoked.

D Persisted single-step primitive

Algorithm 2 `advance_one_unit`: persisted single step.

if not s.kickoff_done **then**
  GROUP_MEETING$(s, s.\text{topic})$
  s.kickoff_done $\leftarrow$ `True`
else if not s.finished **and** $k < s$.max_rounds **then**
  $k \leftarrow k + 1$
  $d \leftarrow \pi_{\text{PI}}(\text{summary}(s))$
  $s \leftarrow T(s, d)$
  if $\#\{\text{accepted papers}\} \geq N_{\text{stop}}$ **then**
    WRAP_UP(s)
    stop_met $\leftarrow$ `True`
  end if
end if
save_embedded_state(s) ▷ atomic write + `fsync`
return {phase, action, round, finished, stop_met}

E Continuous-mode student prompt

Every student work block in LAB-CONTINUOUS is driven by a stereotyped prompt containing the following components:

1. a lab-context line stating the topic the student must stay on;

2. an optional shared-project block enabled by the environment variable `COSMOEVOLVE_SHARED_PROJECT`, in which case `create_project` is disabled for the code subagent;

3. a reminder that the skills are already embedded in the system prompt and that `read_workspace_file` may not be used on anything under `skills/`;

4. a six-step workflow requiring the student to pick a single concrete goal, issue at most one `explore` dispatch, issue one `plan` dispatch, issue one `code` dispatch, call `submit_pi_update`, and finish with a one-paragraph closing summary;

5. explicit hard rules limiting the work block to a single `explore` dispatch (enforced by the tool via per-run counters) and forbidding re-exploration of information already present in the inherited skills.

F Full agent input specifications

This appendix enumerates the complete set of inputs supplied to each agent on every LLM call, for reproducibility.

PI (decision pass). System prompt: PI identity and supervisor instructions, the PI skill index, and the PI's own `store/memory/pi/MEMORY.md` truncated to 4000 characters. User message: the output of summary_for_supervisor(s), consisting of the topic, round counters, the scalar counts $|s.\text{ideas\_proposed}|$, $|s.\text{tasks\_assigned}|$, $|s.\text{papers\_written}|$, $|s.\text{reviews\_done}|$, the scientist roster, and the last ten LabMessage entries of the discussion thread; optionally an accepted-papers counter and a research brief from the preceding research pass truncated to 12000 characters. Running transcript: empty. Output: a single `llm.structured_call(response_model=SupervisorDecision)`. Not reachable: any discussion-thread entry older than the last ten; any student's private `MEMORY.md`, `learnings.jsonl`, or `errors.jsonl`; the content of papers; individual reviewer scores; the contents of the `ArtifactStore`; `budget.json`; the raw tool-call transcript of any student; any subagent transcript; full skill bodies.

PI (research pass). Identical system prompt and user message to the decision pass, but executed as a `BaseAgent.run` with a read-only tool set (`list_workspace`, `read_workspace_file`, `search_arxiv`, `read_skill`, and related). Findings are flattened into the research-brief string that is injected into the user message of the subsequent decision pass and then discarded.

Student coordinator. System prompt: identity and workflow contract; the coordinator tool-set description; the student's own skill index; the student's own `MEMORY.md` truncated to 4000 characters; the ten most recent anti-patterns from `errors.jsonl`; in shared-project deployments, a block describing the shared project name. User message: the single task string passed to `run_task`, either the PI task for PI-initiated calls or the stereotyped continuous-mode work-block prompt of Appendix E; in all cases the message is self-contained. Running transcript: tool-call and tool-result messages from the current `BaseAgent.run`, including the final output of each `dispatch_subagent` call, but not the subagent's internal conversation. Not reachable: other students' system prompts or memory; other students' in-flight work; the PI's research brief or structured decision object; LabState; previous rounds' transcripts (each `run_task` starts fresh); the contents of the `ArtifactStore` except via explicit read-tool calls; `budget.json`; the subagents' tool-call transcripts; full skill bodies except via explicit `read_skill` calls.

Subagents (`explore`, `plan`, `code`). Each dispatch is a fresh `BaseAgent.run`. System prompt: the role-specific identity, the role tool-allowlist description, the parent student's skills inherited verbatim, and the parent student's `MEMORY.md` truncated to 4000 characters. User message: the task string passed by the coordinator to `dispatch_subagent`. Because the subagent cannot see the coordinator's conversation, every task string must be self-contained; the coordinator is responsible for splicing prior intermediate results into subsequent dispatches. Running transcript: the subagent's own tool-call and tool-result messages only. Not reachable: the coordinator's conversation; any sibling subagent's conversation; other students' state; the PI; LabState; `budget.json`.

References

[1] Anthropic. The Claude 3 Model Family: Opus, Sonnet, Haiku. *Technical Report*, 2024.

[2] DeepSeek-AI. DeepSeek-V2: A Strong, Economical, and Efficient Mixture-of-Experts Language Model. *arXiv:2405.04434*, 2024.

[3] Y. Du, S. Li, A. Torralba, J. B. Tenenbaum, and I. Mordatch. Improving Factuality and Reasoning in Language Models through Multiagent Debate. *arXiv:2305.14325*, 2023.

[4] L. D. Erman, F. Hayes-Roth, V. R. Lesser, and D. R. Reddy. The Hearsay-II Speech-Understanding System: Integrating Knowledge to Resolve Uncertainty. *ACM Computing Surveys*, 12(2):213–253, 1980.

[5] Google. Gemini: A Family of Highly Capable Multimodal Models. *Technical Report*, 2024.

[6] B. Hayes-Roth. A Blackboard Architecture for Control. *Artificial Intelligence*, 26(3):251–321, 1985.

[7] S. Hong et al. MetaGPT: Meta Programming for Multi-Agent Collaborative Framework. In *International Conference on Learning Representations (ICLR)*, 2024.

[8] Kimi Team. Kimi k2: Open Agentic Intelligence. *Technical Report*, 2024.

[9] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*, 2023.

[10] G. Li, H. A. A. K. Hammoud, H. Itani, D. Khizbullin, and B. Ghanem. CAMEL: Communicative Agents for "Mind" Exploration of Large Language Model Society. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[11] T. Liang, Z. He, W. Jian, X. Wang, Y. Yang, T. Zhang, and S. Shi. Encouraging Divergent Thinking in Large Language Models through Multi-Agent Debate. *arXiv:2305.19118*, 2024.

[12] C. Lu, C. Lu, R. T. Lange, J. Foerster, J. Clune, and D. Ha. The AI Scientist: Towards Fully Automated Open-Ended Scientific Discovery. *arXiv:2408.06292*, 2024.

[13] A. Madaan et al. Self-Refine: Iterative Refinement with Self-Feedback. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[14] MiniMax. MiniMax-M2: A Production-Ready Agentic MoE Model. *Technical Report*, 2025.

[15] OpenAI. GPT-4 Technical Report. *arXiv:2303.08774*, 2023.

[16] J. S. Park, J. C. O'Brien, C. J. Cai, M. R. Morris, P. Liang, and M. S. Bernstein. Generative Agents: Interactive Simulacra of Human Behavior. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology (UIST)*, 2023.

[17] Pydantic. Pydantic: Data Validation using Python Type Hints. https://docs.pydantic.dev, 2024.

[18] M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, 2014.

[19] C. Qian et al. ChatDev: Communicative Agents for Software Development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL)*, 2024.

[20] Y. Ruan, H. Dong, A. Wang, S. Pitis, Y. Zhou, J. Ba, Y. Dubois, C. J. Maddison, and T. Hashimoto. Identifying the Risks of LM Agents with an LM-Emulated Sandbox. In *International Conference on Learning Representations (ICLR)*, 2024.

[21] T. Schick et al. Toolformer: Language Models Can Teach Themselves to Use Tools. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[22] N. Shinn, F. Cassano, E. Berman, A. Gopinath, K. Narasimhan, and S. Yao. Reflexion: Language Agents with Verbal Reinforcement Learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[23] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2nd edition, 2018.

[24] G. Wang, Y. Xie, Y. Jiang, A. Mandlekar, C. Xiao, Y. Zhu, L. Fan, and A. Anandkumar. Voyager: An Open-Ended Embodied Agent with Large Language Models. *arXiv:2305.16291*, 2023.

[25] Q. Wu et al. AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation. *arXiv:2308.08155*, 2023.

[26] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao. ReAct: Synergizing Reasoning and Acting in Language Models. In *International Conference on Learning Representations (ICLR)*, 2023.