LEARNING COMPACT STATE ABSTRACTIONS FOR LLM AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

LLM agents commonly use the full interaction history as a proxy for state, retaining all past observations and actions regardless of whether they remain relevant. We instead ask what information an agent actually needs to carry forward. We define agent state as a compact abstraction of interaction history that preserves the competence of a fixed reference agent, with the minimal such abstraction as the ideal. Because matching the reference agent's behavior is unnecessarily restrictive and difficult for black-box policies, we define sufficiency through task performance rather than behavior matching. We represent state abstractions as deterministic recursive programs and search over them using an LLM-guided genetic algorithm. Across ALFWorld and ScienceWorld, the learned abstractions preserve near-reference task success while reducing mean policy input by up to 71% and 57%, respectively. The resulting programs are directly inspectable, providing an explicit account of what information is retained, updated, and discarded during interaction.

1 INTRODUCTION

LLM agents typically select an action by conditioning on the full interaction history. Observations, actions, and intermediate reasoning are appended to a growing context and passed back to the model at every step (Yao et al., 2023). This is a natural default because the history is sufficient by construction: no information from the interaction is intentionally discarded. But it retains all information indiscriminately, regardless of whether it remains relevant. As context grows, it accumulates observations and action outcomes whose long-term value may be much smaller than the space they occupy. The model must repeatedly recover the useful information from an increasingly large history, which can eventually exceed the model's finite context window. Most existing approaches treat this as a context-management problem. They summarize the history, retrieve selected memories, or compress the context once it becomes too large (Jiang et al., 2023; Wu et al., 2021; Kang et al., 2025).

We take a different view. The more fundamental question is not how to shorten the context, but what information the agent actually needs to carry forward. The notion of state is foundational to sequential decision making, from Markov decision processes and partially observable MDPs (Bellman, 1957; Sondik, 1971) to reinforcement learning (Sutton, 1988; Watkins & Dayan, 1992; Williams, 1992). Sequential decision making provides a natural framing for this question through state abstraction (Li et al., 2006): rather than assuming the full interaction history is the agent's state, we ask whether a smaller representation can preserve the competence of the same agent on the task.

A natural answer is to define agent state in terms of what the agent needs to remain competent. We call a compact abstraction of the interaction history an agent state if the reference agent preserves its performance when conditioned on that abstraction rather than the full history; the ideal agent state is the minimal such abstraction. This separates two requirements. Sufficiency asks whether the abstraction preserves competence, while minimality asks how much information can be removed without sacrificing it. Because many tasks admit multiple successful trajectories, we measure sufficiency through preserved task performance rather than exact behavioral matching. The full history trivially satisfies sufficiency, so the problem becomes meaningful only when both sufficiency and minimality are considered together. We instantiate the abstraction as a deterministic recursive program that updates the state throughout the interaction and search over such programs with a genetic

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

algorithm, yielding explicit state-update rules that can be inspected and executed without additional model calls at deployment.

Our contributions are threefold:

- **Agent state abstraction.** We formulate agent state for black-box LLM agents as a compact abstraction of interaction history that preserves the competence of a fixed reference agent. This connects the agentic setting to classical state abstraction (Li et al., 2006), with sufficiency defined through task performance.
- **Programmatic state discovery.** We represent agent states as deterministic recursive programs and develop an LLM-guided genetic search procedure (Romera-Paredes et al., 2024; Guo et al., 2024) that discovers such abstractions from interaction alone.
- **Empirical evaluation and interpretability.** We evaluate the approach on ALFWorld (Shridhar et al., 2021) and ScienceWorld (Wang et al., 2022). Across ten task families, the learned abstractions reduce mean policy input by up to 71% in ALFWorld and 57% in ScienceWorld. They exceed full-history task success on four task families and remain within 8 percentage points on the remaining six. Because each state is an executable program, it also provides an interpretable description of the information carried forward during interaction.

2 Background and Related Work

2.1 LLM Agents and the Context-as-State Convention

LLM agents typically choose each action by conditioning on the accumulated interaction history. ReAct (Yao et al., 2023) interleaves reasoning traces with actions and appends both to a growing context, and subsequent agent scaffolds, including reflection over failed attempts (Shinn et al., 2023), planner–executor decompositions (Prasad et al., 2024), and skill libraries (Wang et al., 2023), elaborate the control flow while retaining this convention. In effect, the context serves as the agent's state.

Retaining the full history is sufficient by construction, since no information from the interaction is intentionally discarded. It is also maximally non-selective. As an episode progresses, the context accumulates observations and action outcomes whose durable information may be much smaller than their textual representation, forcing the model to repeatedly recover the relevant information from an increasingly large history.

Existing agent-memory systems impose structure on this history through external storage, retrieval, or hand-designed memory components. MemGPT (Packer et al., 2023) manages a bounded working context against external storage, while generative agents maintain a retrievable memory stream with recency and relevance scoring (Park et al., 2023). These approaches improve how information is stored or accessed, but the structure and contents of the state are still largely specified by the system designer. Our focus is instead on identifying what information the agent actually needs to carry forward.

2.2 State Abstraction in Sequential Decision Making

A central question in sequential decision making is what information from the past must be retained for future control. In partially observable environments, the full interaction history is always a valid information state, but it grows with the horizon and may contain substantial redundancy. Classical work therefore seeks smaller representations that preserve only the distinctions that matter for decision making.

One line of work formalizes this through state abstraction. Li et al. (2006) distinguish several abstraction schemes according to what they preserve. Model-irrelevance abstractions preserve immediate rewards and transition dynamics over abstract states. Q-irrelevance abstractions preserve action values, while π-irrelevance abstractions preserve the action selected by a policy. These form increasingly weaker notions of equivalence, allowing progressively coarser abstractions as fewer properties of the original state space must be retained.

Bisimulation provides a closely related notion of state equivalence. Ferns et al. (2012) characterize states as similar when they induce similar immediate rewards and transition behavior over equivalent

future states, and extend this binary relation to a quantitative metric. In this sense, bisimulation is closely aligned with model-preserving abstractions, but provides a recursive notion of behavioral similarity between states. Abel et al. (2016) further study approximate abstractions, showing how violations of exact abstraction conditions translate into bounded degradation in control performance.

Policy-relative notions are especially relevant to our setting. Castro (2020) introduces a policy-dependent variant of bisimulation in which state similarity is defined relative to a fixed behavior policy. This weakens the requirement that the abstraction preserve distinctions that are irrelevant under the particular policy being executed. Our setting is similarly policy-relative: the LLM agent is fixed, and we seek an abstraction sufficient for that specific agent rather than for all possible policies. However, unlike policy-relative bisimulation, we do not assume access to transition dynamics or state-level reward structure, and we do not require the abstracted process to preserve the environment model.

A related line of work studies compact representations of history under partial observability. Littman & Sutton (2001) represent state through predictions of future observable events rather than latent environment variables, while Singh et al. (2003) develop methods for learning such predictive summaries from experience. Subramanian et al. (2022) study recursively updateable summaries of history that approximately preserve the information required for planning and control. Our state representation shares this recursive structure, but our sufficiency criterion is defined relative to the realized performance of a fixed black-box agent rather than preservation of model-based planning quantities.

These approaches differ in what an abstraction is required to preserve, including transition dynamics, rewards, values, policies, or predictive distributions. Our setting differs in both what is available and what is fixed. The environment and LLM policy are treated as black boxes, the policy is not relearned after abstraction, and the abstraction itself is the object of optimization. We therefore seek a policy-relative notion of sufficiency that can be evaluated directly from interaction, based on whether the fixed agent preserves its task performance under the abstract state.

2.3 Context Compression for Agents

A parallel line of work reduces the amount of context presented to language models. Prompt-compression methods such as LLMLingua (Jiang et al., 2023) remove low-importance tokens under a compression budget, while learned compressors such as Chevalier et al. (2023) encode long contexts into compact summary vectors that are provided to the model as soft prompts. Recursive summarization methods (Wu et al., 2021) have also been used to process inputs beyond the model's context window by hierarchically summarizing segments and their summaries. More directly related to our setting, ACON (Kang et al., 2025) learns natural-language compression guidelines from successful and failed agent trajectories and distills the resulting compressor into a smaller model.

These approaches primarily treat history reduction as a compression or memory-management problem: given a context budget or compression mechanism, the goal is to retain enough information for the agent to continue performing well. Our perspective is different. We treat the problem as one of state identification: rather than asking how much of the history can be compressed, we ask what information must be carried forward for a fixed agent to preserve its competence. Context reduction is therefore a consequence of identifying a sufficient state, rather than the primary objective.

A second distinction is when abstraction is applied. Many context-compression methods operate only after the accumulated history reaches a budget or threshold, compressing an already-grown context. In our setting, the agent operates on the abstract state from the first step onward: each new observation is incorporated into the recursively maintained state, and the discarded history is never carried forward. The agent therefore acts throughout the episode in the learned state space rather than switching from full-history conditioning to compression once the context becomes large. Because the state update is a deterministic program, this recurrence also requires no additional language-model inference at deployment.

2.4 Program Search with Language Models

Recent work has used language models as proposal operators for search over discrete objects. FunSearch (Romera-Paredes et al., 2024) evolves executable programs by repeatedly generating candi-

date modifications and evaluating them with an external objective. EvoPrompt (Guo et al., 2024) similarly combines evolutionary search with LLM-generated mutations and recombinations, but over prompt space rather than programs. Other methods, including TextGrad (Yuksekgonul et al., 2024) and GEPA (Agrawal et al., 2026), use natural-language feedback to iteratively revise prompts or other textual artifacts based on execution outcomes.

Our optimization procedure follows this general pattern but applies it to a different object: a recursive state-update program embedded inside an agent-environment interaction loop. Candidate abstractions are evaluated through the downstream return of the fixed agent, and LLM-based mutation, crossover, and reflection are used to search over programs that preserve performance while reducing state complexity.

3 METHOD

We consider a fixed black-box LLM agent π interacting with an environment $\mathcal{E}$. At time step t, the agent has received observations $o_{1:t}$ and produced actions $a_{1:t-1}$. Standard LLM agents condition on the full interaction history,

$$c_t = (o_1, a_1, \ldots, o_{t-1}, a_{t-1}, o_t),$$

which is sufficient by construction but becomes increasingly redundant as the interaction proceeds.

Rather than treating c_t as the agent's state, we seek a compact state s_t that preserves the agent's task performance. The policy π remains fixed throughout; the only learned object is an abstraction function ϕ that recursively updates the state as

$$s_t = \phi(s_{t-1}, o_t, a_{t-1}),$$

with $s_0 = \varnothing$. The task instruction is provided separately to the agent as part of its prompt and is not stored in the recurrent state. The recursive form is necessary because computing an abstraction directly from c_t would itself require retaining the full history, defeating the purpose of the abstraction.

Our goal is therefore to identify a compact, recursively maintainable representation under which the fixed agent remains competent. We next formalize this objective, then describe the program class used to represent ϕ and the search procedure used to optimize it.

3.1 LEARNING A SUFFICIENT COMPACT STATE

> An agent state is a compact abstraction of the interaction history that preserves the performance of a reference agent acting on the full history. An ideal agent state is the minimal such abstraction.

A natural notion of sufficiency would require the abstracted agent to reproduce the behavior of the reference agent acting on the full history. In our setting, however, behavioral equivalence is unnecessarily strong. Many tasks admit multiple successful trajectories, so two agents may take different actions while achieving the same outcome. Moreover, matching trajectory distributions is impractical for a black-box language model, since an abstraction can alter the agent's subsequent actions and thereby induce a different state distribution from that of the full-history agent.

We therefore define sufficiency through outcome preservation: an abstraction is useful if the fixed agent retains its task performance when conditioned on the abstract state. We write $\pi \circ \phi$ for the induced agent that recursively maintains $s_t = \phi(s_{t-1}, o_t, a_{t-1})$ and acts according to $a_t \sim \pi(\cdot \mid s_t)$.

Let $J(\pi \circ \phi)$ denote the expected task return of this induced agent, and let $T(\phi)$ denote its mean policy input token usage. We optimize

$$\begin{aligned} \max_{\phi} \quad & J(\pi \circ \phi) - \lambda T(\phi) \\ \text{s.t.} \quad & J(\pi \circ \phi) \geq (1-\epsilon) J(\pi_{\text{full}}), \\ & T(\phi) < T(\phi_{\text{naive}}), \end{aligned} \tag{1}$$

where π_{full} denotes the same fixed policy conditioned on the full interaction history and ϕ_{naive} is the initial naive aggregation function. The success constraint prevents state-size reductions from being

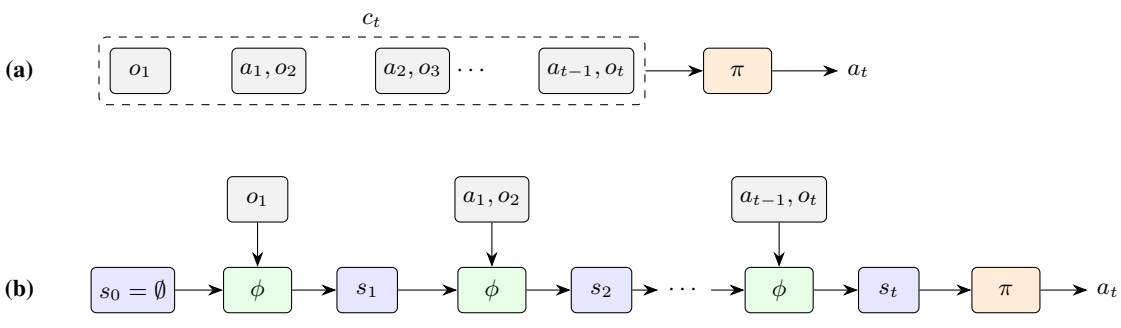

Figure 1: **(a)** A standard LLM agent conditions on the full context c_t, the concatenation of all observations and actions, which grows with the interaction. **(b)** The abstracted agent maintains a recursive state. At each step, ϕ combines the previous state s_{t-1} with the preceding action a_{t-1} and current observation o_t to produce s_t, which is then provided to the policy. The raw interaction history is not retained.

achieved through arbitrary degradation in task performance, while λ provides additional pressure toward compact states among candidate abstractions.

Optimizing Equation 1 over ϕ has the structure of a reinforcement learning problem: the abstraction is applied inside the interaction loop, its effect on behavior compounds over time, and its quality is observed only through trajectory-level return. Unlike standard reinforcement learning, however, the policy π is fixed; the learned object is the state abstraction ϕ itself.

We view this objective as a practical surrogate for the ideal of a minimal sufficient state: our method searches for compact sufficient abstractions within a restricted function class, rather than certifying global minimality.

3.2 Recursive Program States

Optimizing Equation 1 requires choosing a function class for ϕ. We represent ϕ as a deterministic Python program that maps the previous state, current observation, and preceding action to the next state,

$$s_t = \phi(s_{t-1}, o_t, a_{t-1}). \tag{2}$$

Equation 2 has the form of a recurrent hidden-state update. At each step, the current observation and preceding action are absorbed into s_t, after which the raw interaction history need not be retained. Figure 1 contrasts this recursive state representation with the standard full-context agent. Unlike a learned recurrent neural network, however, the update function ϕ is an explicit deterministic program whose state variables and update rules are directly inspectable.

Programs are well suited to this setting because the abstraction primarily requires structured state updates: parsing observations, maintaining task-relevant records, updating counters or flags, and discarding text once its relevant information has been extracted. They also allow the abstraction to be executed without an additional model call at each environment step.

We restrict ϕ to deterministic programs. Since the abstraction is applied recursively throughout the interaction, stochastic errors introduced at one step could propagate through all subsequent states after the original history has been discarded. Deterministic execution avoids this source of compounding uncertainty.

Finally, the resulting abstraction is directly inspectable. Rather than representing state as an opaque latent vector, the learned program explicitly specifies what information is retained, updated, and discarded during the interaction. This makes the learned abstraction itself a readable hypothesis about the information required by the agent.

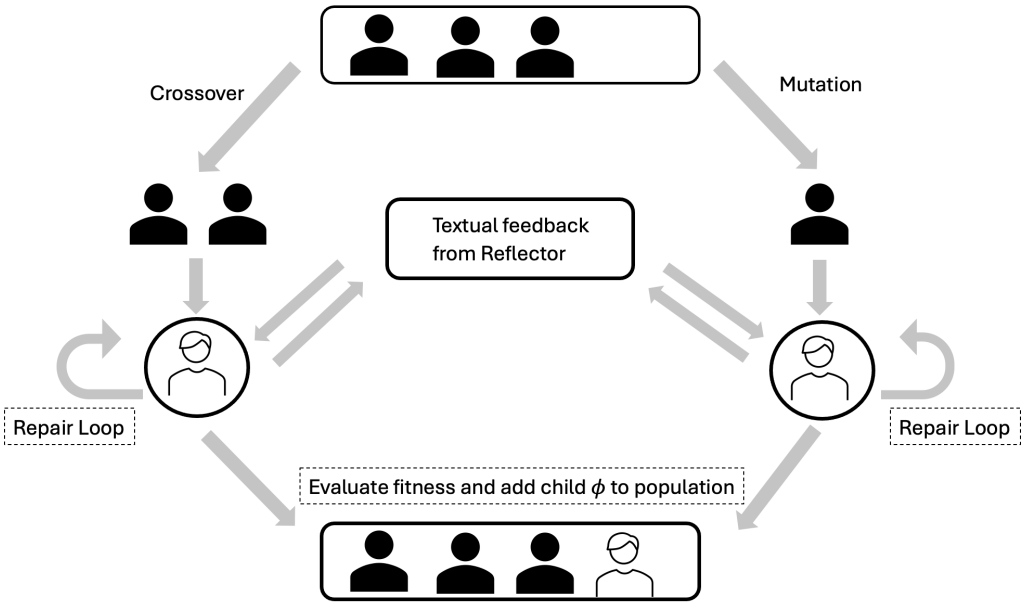

Figure 2: Overview of the genetic search procedure. Parent programs are selected for mutation or crossover and an LLM proposer generates a child. The child is evaluated and, when infeasible, iteratively revised using feedback from a reflector for at most three repair attempts. Only the highest-fitness candidate from the resulting repair trajectory is added to the population.

3.3 Searching Over State-Update Programs

The objective in Equation 1 is non-differentiable and can only be evaluated through interaction with the environment. We therefore optimize over abstraction programs using a genetic algorithm (Holland, 1975) with black-box fitness evaluation, following recent work that combines evolutionary search with LLM-generated program proposals (Romera-Paredes et al., 2024; Guo et al., 2024).

We maintain a population of candidate state-update programs. Each candidate is evaluated by rolling out the fixed agent while conditioning on its recursively maintained state. Parent programs are selected by tournament selection, and children are generated through mutation or crossover. For mutation, the proposer, implemented as GPT-5.6 Luna (OpenAI, 2026), receives a single parent program; for crossover, it receives two parents and is asked to combine useful state components from both. The proposer is additionally given the parent rollouts, corresponding full-history reference rollouts, and textual feedback from a reflector model, also implemented as GPT-5.6 Luna. Comparing the abstracted and reference rollouts helps identify performance-relevant information that may have been lost by the current abstraction.

Feasibility. We treat preservation of task performance as a constraint rather than allowing arbitrary trade-offs between success and state size. Let $J(\pi \circ \phi)$ denote the task success of the fixed policy under abstraction ϕ, and let $T(\phi)$ denote its mean policy input token usage. A candidate is feasible when

$$J(\pi \circ \phi) \geq (1-\epsilon)J(\pi_{\text{full}}) \qquad \text{and} \qquad T(\phi) < T(\phi_{\text{naive}}), \tag{3}$$

where π_{full} is the full-history reference and ϕ_{naive} is the initial naive aggregation program. Among feasible candidates, smaller states are preferred. We use $\epsilon = 0.10$ in all experiments.

Reflection and repair. Because small edits to a program can induce large changes in agent behavior, a newly proposed child is evaluated before being admitted to the population. If the child is infeasible, the reflector compares its rollout with the relevant parent and full-history reference rollouts and identifies information that may have been lost or tracked incorrectly. The proposer then revises the child using this feedback. We repeat this reflect-and-repair procedure until a feasible candidate is found or a maximum of three repair attempts is reached. The original proposal

and all repaired variants are retained during this process, but are not individually inserted into the population. Only the highest-fitness candidate from the repair process is added.

Each candidate is evaluated only when it is first produced. Its rollout, task success, token usage, and fitness are cached and reused for subsequent selection, reflection, mutation, crossover, and repair. After a fixed number of search iterations, we return the highest-fitness abstraction found during the search. Search hyperparameters and the reflector and proposer prompts are provided in Appendix B.2 and C respectively. Algorithm 1 provides the complete search procedure.

4 EXPERIMENTS

4.1 SETUP

We evaluate whether the learned abstractions preserve agent performance while reducing the number of tokens ingested by the policy across two interactive environments: ALFWorld (Shridhar et al., 2021) and ScienceWorld (Wang et al., 2022). In each domain, we learn a separate abstraction for each predefined task family using training instances from that family and evaluate it on held-out instances. The reference agent uses GPT-5.6 Luna (OpenAI, 2026) in a ReAct (Yao et al., 2023) scaffold and conditions on the full interaction history throughout the episode. We use GPT-5.6 Luna because our goal is to study state abstraction for a fixed black-box LLM policy rather than compare language models, and Luna provides a strong agent policy at a tractable cost for the repeated environment rollouts required by genetic search.

ALFWorld. ALFWorld (Shridhar et al., 2021) is a text-based household environment in which an agent must complete high-level instructions through navigation and object interaction. We evaluate on its six task families: `pick_and_place_simple`, `look_at_obj_in_light`, `pick_clean_then_place`, `pick_heat_then_place`, `pick_cool_then_place`, and `pick_two_obj_and_place`. The benchmark provides held-out environments, allowing us to test whether an abstraction learned for a task family transfers to new instances rather than memorizing individual trajectories. We use 12 training instances per task family (drawn from the seen split) and evaluate on all remaining held-out instances—the unused seen environments together with all unseen environments—which range from 19 to 47 per family (47, 19, 46, 27, 34, and 29 for the six families in the order listed above).

ScienceWorld. ScienceWorld (Wang et al., 2022) contains interactive science tasks requiring an agent to manipulate objects, inspect environments, and execute multi-step procedures. We evaluate on four task families: `lifespan`, `forces`, `classification`, and `genetics`. These tasks provide a qualitatively different setting from ALFWorld, with state requirements that include experimental progress, object properties, and intermediate scientific observations. We use 24, 24, 32, and 6 training instances for `lifespan`, `forces`, `classification`, and `genetics`, respectively (spread-sampled across each family's variation range), and evaluate on 12, 12, 16, and 12 held-out instances.

Baselines. We compare against the full-history reference agent, an LLM Summarizer baseline, and ACON (Kang et al., 2025). The LLM Summarizer uses the same runtime compression mechanism as our ACON implementation but with a fixed summarization guideline: once the retained context exceeds a token threshold, an LLM summarizes the older interaction history while preserving the most recent turn verbatim. ACON additionally optimizes the summarization guideline using its utility-maximization and compression-maximization stages before held-out evaluation. Because the original ACON implementation targets different domains, we faithfully re-implement its UT→CO optimization procedure for ALFWorld and ScienceWorld.

Evaluation. We report task success on held-out instances and the mean number of policy input tokens per step, averaged across evaluation episodes. For the LLM Summarizer and ACON, this token count reflects only the compressed context provided to the policy and does not include the additional tokens consumed by the runtime compressor.

4.2 RESULTS

Tables 1 and 2 compare our learned state abstractions against the full-history reference agent, a fixed LLM Summarizer, and ACON.

	Full		**Ours**		LLM-S		ACON	
ALFWorld task	Succ.	Tok.	Succ.	Tok.	Succ.	Tok.	Succ.	Tok.
Pick & Place	**1.00**	281	0.96	**82** (71%)	0.94	211 (25%)	**1.00**	183 (35%)
Look in Light	0.95	296	**1.00**	247 (17%)	0.95	207 (30%)	0.83	**200** (32%)
Clean & Place	**0.74**	519	0.72	245 (53%)	0.67	245 (53%)	0.67	**207** (60%)
Heat & Place	0.74	555	**0.85**	**172** (69%)	0.63	256 (54%)	0.25	201 (64%)
Cool & Place	0.76	561	0.74	**167** (70%)	0.59	252 (55%)	**0.83**	216 (61%)
Two Objects	**0.90**	430	0.86	402 (7%)	0.83	245 (43%)	0.42	**203** (53%)

Table 1: Held-out ALFWorld performance and mean policy input tokens. Numbers in parentheses report the reduction in mean policy input tokens relative to the full-history agent. LLM-S denotes the fixed LLM Summarizer, while ACON includes learned compression-guideline optimization. Bold indicates the best result and underline the second-best result within each metric.

	Full		**Ours**		LLM-S		ACON	
ScienceWorld task	Succ.	Tok.	Succ.	Tok.	Succ.	Tok.	Succ.	Tok.
Lifespan	**1.00**	260	0.92	**136** (48%)	0.92	236 (9%)	0.83	229 (12%)
Forces	0.42	394	**0.75**	412 (+5%)	0.33	265 (33%)	0.50	**208** (47%)
Classification	**0.50**	395	0.44	**169** (57%)	**0.50**	246 (38%)	0.38	239 (39%)
Genetics	0.67	401	**0.83**	247 (38%)	0.58	237 (41%)	0.58	**209** (48%)

Table 2: Held-out ScienceWorld performance and mean policy input tokens. Numbers in parentheses report the reduction in mean policy input tokens relative to the full-history agent; positive values with a + indicate an increase. LLM-S denotes the fixed LLM Summarizer, while ACON includes learned compression-guideline optimization. Bold indicates the best result and underline the second-best result within each metric.

ALFWorld. Our learned abstractions remain close to the full-history reference across all six task families while substantially reducing policy input for most tasks. The largest reductions occur for Pick & Place, Heat & Place, and Cool & Place, where mean policy input falls by approximately 69–71%. Clean & Place is reduced by 53%, while Look in Light and Two Objects admit much smaller reductions. This variation suggests that the amount of information that must be retained is strongly task dependent.

ScienceWorld. The same pattern appears in ScienceWorld. Our abstractions reduce mean policy input by 48% for Lifespan, 57% for Classification, and 38% for Genetics, while maintaining comparable or higher task success. Forces is the exception: the learned representation is slightly larger than full history. Together with the ALFWorld results, this indicates that our search does not find comparably compact abstractions across all task families.

Comparison to compression baselines. Relative to ACON, our method attains higher success on four of six ALFWorld task families and uses fewer policy input tokens on three of six. Relative to the fixed LLM Summarizer, it attains higher success on all six task families, uses fewer tokens on three, and ties on one. On ScienceWorld, our method attains higher success than ACON on all four task families and uses fewer policy input tokens on Lifespan and Classification. Compared with the fixed LLM Summarizer, it attains higher success on two tasks, ties on Lifespan, and uses fewer tokens on Lifespan and Classification.

Policy input token usage and task success should not be interpreted independently: unsuccessful trajectories may terminate earlier or make less progress and therefore consume fewer tokens. In addition, *both compression baselines require online LLM inference to generate summaries during execution, whereas our learned abstraction is a deterministic Python program that does not require LLM inference*.

4.3 What Do the Learned State Abstractions Contain?

To understand what the learned abstractions represent, we inspect the state variables and update rules of the final programs. Across task families, the programs typically convert raw interaction text into structured persistent variables and discard observations once their task-relevant information has been incorporated. Table 3 in Appendix A summarizes the information retained across all task families, while Table 4 reports the structural complexity of each learned program. Additional representative ϕ programs for both ALFWorld and ScienceWorld are provided in Appendix A.

For example, the ALFWorld Pick & Place abstraction maintains a structured search state containing the current location, searched receptacles, target location, inventory status, and task phase:

ALFWorld Pick & Place

```
m = re.search(r"\byou arrive at\s+(.+?)(?:\.|$)", ol)
if m:
    loc = norm(m.group(1))
    focus = loc
    add_receptacle(loc)
    if held is None:
        phase = "search"

m = re.search(
    r"\b(?:on|in)\s+the\s+([a-z][a-z0-9]*(?:\s+\d+)?)"
    r"\s*,?\s*you see\s+(.*?)(?:\.|$)", ol,
)
if m:
    view = norm(m.group(1))
    mark_searched(view)
    found = matching_target(m.group(2))

    if found and held is None:
        objloc = view
        phase = "pickup"
        frontier = []
```

Other ALFWorld abstractions retain task-specific persistent properties such as cleanliness, temperature, cooling status, receptacle contents, and task completion. ScienceWorld yields qualitatively different representations. The learned states track developmental stages, device configurations, object relations, entity locations, and experimental progress. For Classification, for example, the abstraction resolves entities and their locations from raw observations and maintains a bounded symbolic world model; its learned program is shown in Appendix A.

In contrast, the abstractions learned for ALFWorld Two Objects and ScienceWorld Forces remain close to bounded recent interaction histories, consistent with the limited token reductions observed for these tasks. Overall, the learned programs repeatedly replace raw trajectory text with task-specific persistent state, while the amount and form of retained information vary substantially across task families.

5 Conclusion

We asked what information an LLM agent actually needs to carry forward during interaction, rather than treating its full history as state by default. We formulated agent state as a compact abstraction of history that preserves the competence of a fixed reference agent, and instantiated this abstraction as a deterministic recursive program learned through black-box search. Across ALFWorld and ScienceWorld, the resulting states often preserve reference-level task performance while substantially reducing mean policy input tokens, although the achievable reduction varies considerably across task families. These results suggest that agent state is neither uniformly small nor identical across tasks, but can often be represented by a much more selective recurrent summary than the raw interaction history.

AI USE STATEMENT

Generative AI tools were used to assist with writing and editing the manuscript, including improving clarity, readability, and organization. They were also used to assist with software development, including implementation of the genetic search framework for learning state-abstraction functions and adapting baseline methods, such as ACON, to environments not supported by their original implementations.

Generative language models are also an explicit component of the methodology and experimental setup in this work. We evaluate LLM-based agents, and our search procedure uses LLMs as proposer and reflector models to generate and revise candidate state-update programs. The ACON baseline similarly uses language models as part of its compression and optimization procedure, as described in the paper.

All AI-assisted manuscript text, code, experimental configurations, and implementations were reviewed by the authors. The authors verified the correctness of the final implementations and experimental results and take responsibility for all content, claims, analyses, and artifacts presented in this work.

REFERENCES

David Abel, David Hershkowitz, and Michael Littman. Near optimal behavior via approximate state abstraction. In *International Conference on Machine Learning*, pp. 2915–2923. PMLR, 2016.

Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, et al. Gepa: Reflective prompt evolution can outperform reinforcement learning. In *International Conference on Learning Representations*, volume 2026, pp. 8479–8565, 2026.

Richard Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, 1957.

Pablo Samuel Castro. Scalable methods for computing state similarity in deterministic markov decision processes. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pp. 10069–10076, 2020.

Alexis Chevalier, Alexander Wettig, Anirudh Ajith, and Danqi Chen. Adapting language models to compress contexts. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 3829–3846, 2023.

Norman Ferns, Prakash Panangaden, and Doina Precup. Metrics for finite markov decision processes. *arXiv preprint arXiv:1207.4114*, 2012.

Qingyan Guo, Rui Wang, Junliang Guo, Bei Li, Kaitao Song, Xu Tan, Guoqing Liu, Jiang Bian, and Yujiu Yang. Connecting large language models with evolutionary algorithms yields powerful prompt optimizers. In *International Conference on Learning Representations (ICLR)*, 2024.

John H. Holland. Adaptation in Natural and Artificial Systems. University of Michigan Press, 1975. URL https://mlanthology.org/misc/1975/holland1975misc-adaptation/.

Huiqiang Jiang, Qianhui Wu, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. Llmlingua: Compressing prompts for accelerated inference of large language models. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pp. 13358–13376, 2023.

Minki Kang, Wei-Ning Chen, Dongge Han, Huseyin A Inan, Lukas Wutschitz, Yanzhi Chen, Robert Sim, and Saravan Rajmohan. Acon: Optimizing context compression for long-horizon llm agents. *arXiv preprint arXiv:2510.00615*, 2025.

Lihong Li, Thomas J Walsh, and Michael L Littman. Towards a unified theory of state abstraction for mdps. *AI&M*, 1(2):3, 2006.

Michael Littman and Richard S Sutton. Predictive representations of state. *Advances in neural information processing systems*, 14, 2001.

OpenAI. GPT-5.6 Luna. https://developers.openai.com/api/docs/models/gpt-5.6-luna, 2026. Accessed: 2026-09-24.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G Patil, Ion Stoica, and Joseph E Gonzalez. Memgpt: Towards llms as operating systems. *arXiv preprint arXiv:2310.08560*, 2023.

Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th annual acm symposium on user interface software and technology*, pp. 1–22, 2023.

Archiki Prasad, Alexander Koller, Mareike Hartmann, Peter Clark, Ashish Sabharwal, Mohit Bansal, and Tushar Khot. Adapt: As-needed decomposition and planning with language models. In *Findings of the Association for Computational Linguistics: NAACL 2024*, pp. 4226–4252, 2024.

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*, 625:468–475, 2024.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. ALFWorld: Aligning text and embodied environments for interactive learning. In *International Conference on Learning Representations (ICLR)*, 2021.

Satinder P Singh, Michael L Littman, Nicholas K Jong, David Pardoe, and Peter Stone. Learning predictive state representations. In *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, pp. 712–719, 2003.

Edward Sondik. *The Optimal Control of Partially Observable Markov Decision Processes*. PhD thesis, Stanford University, 1971.

Jayakumar Subramanian, Amit Sinha, Raihan Seraj, and Aditya Mahajan. Approximate information state for approximate planning and reinforcement learning in partially observed systems. *Journal of Machine Learning Research*, 23(12):1–83, 2022.

Richard Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3: 9–44, 1988.

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

Ruoyao Wang, Peter Jansen, Marc-Alexandre Côté, and Prithviraj Ammanabrolu. Scienceworld: Is your agent smarter than a 5th grader? In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pp. 11279–11298, 2022.

Christopher Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3):279–292, 1992.

Ronald Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3-4):229–256, 1992.

Jeff Wu, Long Ouyang, Daniel M Ziegler, Nisan Stiennon, Ryan Lowe, Jan Leike, and Paul Christiano. Recursively summarizing books with human feedback. *arXiv preprint arXiv:2109.10862*, 2021.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. Textgrad: Automatic" differentiation" via text. *arXiv preprint arXiv:2406.07496*, 2024.

Task family	Persistent information retained
ALFWorld	
Pick & Place	current location, target and destination, held object, search frontier, searched receptacles, task phase
Look in Light	object locations, receptacle contents, open/closed state, device state, examination results, recent observations
Clean & Place	location, held object, target locations, cleanliness, receptacle contents, searched locations, task completion
Heat & Place	target identity and location, required condition, object property, container state, task phase
Cool & Place	target identity and location, required transformation, container state, searched locations, task phase
Two Objects	bounded recent interaction history
ScienceWorld	
Lifespan	entity identity, location, developmental stage, inspected stage, inventory and task progress
Forces	bounded recent interaction history
Classification	entities, aliases, flower-pot contents, inventory, object locations, facts, milestones
Genetics	object locations, inventory, device states, connections, plant-development stages, failed actions

Table 3: Representative persistent information retained by the learned state-update programs.

Task family	Program LOC	State fields
ALFWorld		
Pick & Place	278	11
Look in Light	174	11
Clean & Place	297	15
Heat & Place	428	18
Cool & Place	358	18
Two Objects	10	1
ScienceWorld		
Lifespan	283	10
Forces	18	1
Classification	210	12
Genetics	165	15

Table 4: Structural complexity of the final learned state-update programs. Program Lines Of Code (LOC) counts non-empty, non-comment source lines in the final ϕ. State fields reports the number of distinct top-level keys that may appear in the returned state dictionary; individual states may contain fewer fields because many keys are conditionally omitted.

A Learned State-Update Programs

This appendix provides representative excerpts from the learned state-update programs for each task family. We omit normalization, defensive copying, and other implementation boilerplate where possible, and instead show the portions of each program that best illustrate how raw interaction text is converted into persistent state.

A.1 ALFWorld

Pick & Place. The learned abstraction maintains a search frontier and converts observations about receptacles into persistent search state. Finding the target changes the task phase and removes exploratory state that is no longer required.

Pick & Place: Search State

```
m = re.search(r"\byou arrive at\s+(.+?)(?:\.|$)", ol)
if m:
    loc = norm(m.group(1))
    focus = loc
    add_receptacle(loc)
    if held is None:
        phase = "search"

m = re.search(
    r"\b(?:on|in)\s+the\s+([a-z][a-z0-9]*(?:\s+\d+)?)"
    r"\s*,?\s*you see\s+(.*?)(?:\.|$)", ol,
)
if m:
    view = norm(m.group(1))
    contents = m.group(2)
    mark_searched(view)

    found = matching_target(contents)
    if found and held is None:
        objloc = view
        phase = "pickup"
        frontier = []

frontier = [
    x for x in dict.fromkeys(frontier)
    if x not in searched and not is_destination(x)
]
```

Look at Object in Light. Here the abstraction maintains a small world model containing object locations, receptacle contents, container state, and device properties. It explicitly bounds these structures and preferentially retains goal-relevant objects.

Look in Light: Structured World State

```
loc = dict(new.get("loc", {}))
cont = {k: list(v) for k, v in new.get("cont", {}).items()}
opn = dict(new.get("open", {}))
prop = dict(new.get("prop", {}))
exam = dict(new.get("exam", {}))

for x in ents(lo):
    if re.search(r"\b" + re.escape(x) + r"\s+is closed\b", lo):
        opn[x] = "closed"
    elif re.search(r"\b" + re.escape(x) + r"\s+is open\b", lo):
        opn[x] = "open"

eff = re.search(
    r"\b(?:turn|switch)\s+(on|off)\s+(?:the\s+)?("
    + ent_re + r")", lo,
)
if eff:
    prop[eff.group(2)] = eff.group(1)

if len(loc) > 28:
    keep = [
        k for k in loc
        if k == held or
        (goal_obj and k.startswith(goal_obj + " "))
    ]
    keep += [k for k in loc if k not in keep]
    loc = {k: loc[k] for k in keep[:28]}
```

Clean & Place. The cleaning abstraction tracks whether the target object has acquired the required property, alongside its location and the agent's search history. Once the task is complete, it aggressively prunes search-related state.

Clean & Place: Property and Completion State

```
m = re.search(r"\bclean\s+([a-z]+\s+\d+)\b", am)
if m and not failure:
    obj = m.group(1).lower()
    if "you clean" in low_obs or "cleaned" in low_obs:
        out["clean"][obj] = True

m = re.search(
    r"\b(?:move|put|place)\s+([a-z]+\s+\d+)\s+"
    r"(?:to|in|on)\s+([a-z]+\s+\d+)", am,
)
if m and not failure:
    obj, dst = m.group(1).lower(), m.group(2).lower()
    out["held"] = None
    out["items"][obj] = dst

    if (
        move_confirmed
        and is_target
        and is_destination
        and (not requires_clean or is_clean)
    ):
        out["done"] = True

if out.get("done"):
    out["searched"] = [
        p for p in out["searched"]
        if p in keep_places
    ]
```

Heat & Place. The learned program first parses the task specification to determine the target object and required condition, and then converts successful transformation events into persistent object properties.

Heat & Place: Goal and Property Parsing

```
m = re.search(
    r"\b(?:heat|warm)\s+(?:some|a|an)\s+([a-z]+)",
    gt,
)
if m:
    target, cond = m.group(1), "hot"

effects = (
    ("hot",
     r"\byou heat the ([a-z]+\s+\d+)\b",
     r"\bheat\s+([a-z]+\s+\d+)\s+with\b"),
    ("cold",
     r"\byou (?:cool|chill) the ([a-z]+\s+\d+)\b",
     r"\b(?:cool|chill)\s+([a-z]+\s+\d+)\s+with\b"),
)

for value, op, ap in effects:
    m = re.search(op, lo)
    if not m and not failed():
        m = re.search(ap, la)
```

```
    if m and is_target(m.group(1)):
        oid = norm(m.group(1))
        out["obj"] = oid
        out["prop"][oid] = value
        out["phase"] = "conditioned"

out["prop"] = {
    k: v for k, v in out["prop"].items()
    if is_target(k)
}
```

Cool & Place. The cooling abstraction parses both the target transformation and destination from the task specification. As execution progresses, it transitions among search, acquisition, transformation, and placement phases and prunes exploratory state after the target is acquired.

Cool & Place: Phase-Dependent State

```
if not need:
    for w, n in (
        ("cool", "cool"), ("chill", "cool"),
        ("clean", "clean"), ("wash", "clean"),
        ("heat", "heat"), ("warm", "heat"),
    ):
        if re.search(r"\b" + w + r"\b", g):
            need = n
            break

if target_id:
    target_loc = seen.get(
        target_id, out.get("target_loc", "")
    )
    status = statuses.get(
        target_id, out.get("target_status", "")
    )

    if target_loc == "held" or held == target_id:
        if need and status != need:
            phase = "transform"
        elif dest:
            phase = "place"
        else:
            phase = "done"
    elif target_loc:
        phase = "acquire"
    else:
        phase = "search"

if acquired or phase == "done":
    out.pop("target_candidates", None)
    out.pop("searched", None)
    out.pop("source_map", None)
```

Two Objects. In contrast to the preceding task families, the selected abstraction remains close to recent interaction history. This is consistent with the much smaller token reduction observed for this task family.

Two Objects: Bounded Recent History

```
new_state = dict(state)
history = list(state.get("history", []))
```

```
entry = {
    "a": event.get("last_action") or "",
    "o": event.get("last_observation") or "",
}

history.append(entry)
new_state["history"] = history[-40:]

return new_state
```

A.2 SCIENCEWORLD

Lifespan. The learned lifespan abstraction canonicalizes entities across textual descriptions and maintains their current developmental stage, location, and whether the relevant stage has been explicitly inspected.

Lifespan: Entity and Developmental State

```
def remember_entity(surface, inspected=False):
    surface = clean(surface).lower()
    key = canonical(surface)
    stage = lifecycle_stage(surface)

    old_entity = out["entities"].get(key, {})
    previous_stage = old_entity.get("s", "")

    record = {
        "s": stage,
        "n": surface,
        "l": out["loc"] or old_entity.get("l", "unknown"),
        "is": old_entity.get("is", ""),
    }

    if previous_stage and previous_stage != stage:
        record["is"] = ""

    if inspected:
        record["is"] = stage

    out["entities"][key] = record
```

Forces. As with ALFWorld two-object placement, the selected abstraction for forces remains a bounded interaction history rather than reducing the trajectory to a small structured state.

Forces: Bounded Recent History

```
new_state = dict(state)
history = list(state.get("history", []))

entry = {
    "a": event.get("last_action") or "",
    "o": event.get("last_observation") or "",
}

history.append(entry)
new_state["history"] = history[-40:]

return new_state
```

Classification. The classification abstraction constructs a compact entity-relation representation. It parses flower-pot contents, resolves generic references to concrete entities, tracks locations and inventory, and bounds the number of retained facts and milestones.

Classification: Entity Resolution

```
for match in re.finditer(
    r"flower pot\s+(\d+)\s*\(containing\s+([^)]*)\)",
    obs, flags=re.I
):
    pot = "pot" + match.group(1)
    contents = clean(match.group(2))
    new["pots"][pot] = contents

    if contents and contents.lower() not in {
        "nothing", "empty"
    }:
        entity = resolve(contents)
        new["ents"][entity.lower()] = {"loc": pot}

if (
    action_obj
    and generic(action_obj)
    and not generic(observed_obj)
):
    new["aliases"][action_obj.lower()] = observed_obj

new["facts"] = new["facts"][-6:]
new["milestones"] = new["milestones"][-10:]
```

Genetics. The genetics abstraction retains relational and process state, including connections between objects, device configurations, object locations, and plant-development stages.

Genetics: Relational and Process State

```
m = re.search(r"connect (.+?) to (.+)", action, re.I)
if m and (
    "connect" in ol or
    "connected" in ol or
    "now" in ol
):
    edge = (
        clean(m.group(1), 55).lower()
        + ">"
        + clean(m.group(2), 55).lower()
    )
    if edge not in con:
        con.append(edge)

pot_match = re.search(
    r"(flower pot\s*\d+)", obs, re.I
)
if pot_match:
    pot = clean(pot_match.group(1), 30).lower()

    if "reproducing stage" in ol:
        stage = "reproducing"
    elif "flowering stage" in ol:
        stage = "flowering"
    elif "seedling" in ol:
        stage = "seedling"
    elif "dead" in ol:
        stage = "dead"
```

Hyperparameter	Value
Initial population size	1
Population size	8
Search iterations	10 (ALFWorld), 5 (ScienceWorld)
Tournament size	3
Crossover probability	0.5
Proposer model	GPT-5.6 Luna
Reflector model	GPT-5.6 Luna
Rollouts per candidate	1
Maximum repair attempts	3
Maximum success drop ϵ	10%
State-size coefficient λ	5×10^{-4}
Schema S	Frozen
Search seed	0

Table 5: Hyperparameters used for genetic search over state-update programs.

```
    else:
        stage = None

    if stage:
        plants[pot] = stage
```

Summary. Across task families, the learned programs repeatedly convert raw observations into persistent structured variables and remove textual evidence once its relevant information has been incorporated. The particular representation is strongly task dependent: ALFWorld abstractions primarily recover navigation, search, object, and transformation state, while ScienceWorld additionally requires entity resolution, relational structure, device configuration, and process variables. For the two task families with little or no context reduction, the selected representations instead remain close to bounded interaction histories.

B Additional Search Details

B.1 Genetic Search Procedure

Algorithm 1 gives the complete genetic search procedure described in Section 3.3.

B.2 Search Hyperparameters

Table 5 reports the configuration used for genetic search. Unless otherwise stated, the same search procedure is used across task families. We initialize the population with a single naive aggregation function ϕ, use a state-size coefficient of $\lambda = 5 \times 10^{-4}$, and define a candidate as feasible if its task success is within 10% of the full-history reference while using fewer policy input tokens than the naive aggregation baseline. We run 10 search iterations for ALFWorld and 5 for ScienceWorld.

C Prompt Templates

Our genetic search uses an LLM reflector and proposer, both instantiated with GPT-5.6 Luna. In all reported runs, the schema S is frozen and only the state-update program ϕ is modified. The reflector receives the environment description, optimization objective, paired full-history and compact-state rollouts, and an instruction to diagnose missing state information or compression opportunities. The proposer then receives the current ϕ, the reflector feedback, example episodes, the frozen schema fields, and the program contract, and proposes a targeted revision to ϕ.

Algorithm 1 Genetic search for recursive state abstractions

Require: Initial population $\mathcal{P}_0$, fixed policy π, naive full-history reference agent, generations G, maximum allowed success drop ϵ

Obtain and cache reference rollouts $\mathcal{D}_{\text{ref}}$
for all $\phi \in \mathcal{P}_0$ **do**
  Evaluate ϕ once and cache its rollouts $\mathcal{D}_\phi$ and fitness $F(\phi)$
end for
for $g = 1, \ldots, G$ **do**
  Select genetic operator $o \in \{\textsc{Mutation}, \textsc{Crossover}\}$
  if $o = \textsc{Mutation}$ **then**
    Select parent ϕ_p from $\mathcal{P}_{g-1}$ using tournament selection
    Retrieve cached rollout $\mathcal{D}_{\phi_p}$
    Generate reflector feedback using $\mathcal{D}_{\phi_p}$ and $\mathcal{D}_{\text{ref}}$
    $\phi^{(0)} \leftarrow \textsc{Mutate}\big(\phi_p, \mathcal{D}_{\phi_p}, \mathcal{D}_{\text{ref}}, \text{feedback}\big)$
  else
    Select parents ϕ_p, ϕ_q from $\mathcal{P}_{g-1}$ using tournament selection
    Retrieve cached rollouts $\mathcal{D}_{\phi_p}, \mathcal{D}_{\phi_q}$
    Generate reflector feedback using $\mathcal{D}_{\phi_p}$, $\mathcal{D}_{\phi_q}$, and $\mathcal{D}_{\text{ref}}$
    $\phi^{(0)} \leftarrow \textsc{Crossover}\big(\phi_p, \phi_q, \mathcal{D}_{\phi_p}, \mathcal{D}_{\phi_q}, \mathcal{D}_{\text{ref}}, \text{feedback}\big)$
  end if
  Evaluate $\phi^{(0)}$ once and cache $\mathcal{D}_{\phi^{(0)}}$ and $F(\phi^{(0)})$
  $\mathcal{A} \leftarrow \{\phi^{(0)}\}$
  for $r = 0, \ldots, 2$ **do**
    if $\phi^{(r)}$ satisfies Eq. 3 **then**
      break
    end if
    Generate reflector feedback using cached $\mathcal{D}_{\phi^{(r)}}$, parent rollout(s), and $\mathcal{D}_{\text{ref}}$
    $\phi^{(r+1)} \leftarrow \textsc{Repair}(\phi^{(r)}, \text{feedback})$
    Evaluate $\phi^{(r+1)}$ once and cache $\mathcal{D}_{\phi^{(r+1)}}$ and $F(\phi^{(r+1)})$
    $\mathcal{A} \leftarrow \mathcal{A} \cup \{\phi^{(r+1)}\}$
  end for
  $\phi^\star \leftarrow \arg\max_{\phi \in \mathcal{A}} F(\phi)$
  Add only $\phi^\star$ to the population
  $\mathcal{P}_g \leftarrow \textsc{UpdatePopulation}(\mathcal{P}_{g-1}, \phi^\star)$
end for
return $\arg \max_{\phi \in \bigcup_{g=0}^{G} \mathcal{P}_g} F(\phi)$

C.1 Reflector Prompt

The reflector receives the environment description, optimization objective, and paired full-history and compact-state trajectories. Its role is to diagnose which information is missing from the current abstraction when the policy fails, and to identify compression opportunities when the policy succeeds. The prompt used in our experiments is reproduced below, with $\{\ldots\}$ denoting task- or candidate-specific content.

> {`environment_description`}
>
> **Objective.**
>
> What is being optimized: TASK SUCCESS RATE. A candidate is a pair (S, ϕ): S declares fields and a `parse_step(observation, action)` function that extracts one step's fields from that observation/action pair alone, while ϕ(`state`, `event`, S, `goal`) folds one new parsed event into a persistent state carried forward across steps.
>
> The fixed policy reads ϕ's compact state instead of the full raw interaction history available to the teacher reference. Success is the primary objective, while state size provides secondary pressure.
>
> The teacher's specific trajectory is only one example of a successful path. Different valid trajectories may solve the same task. Do NOT recommend matching the teacher's step-by-

step actions. Focus on whether the candidate solves the task and, when it fails, why: what information was missing that the policy needed to choose a better action. Trajectory-shape matching is not the objective; task completion is.

Reference and candidate trajectories.

For each evaluation episode, the prompt contains a full-history teacher trajectory and the corresponding trajectory produced when the policy uses the candidate abstraction. Each trajectory is rendered step by step as

```
--- {label} ---
step i: action={a} -> observation={o}
```

with available actions and execution errors included when present.

Reflector instruction.

Your task: analyze the candidate's mixed performance ({n_succ}/{n_total} succeeded, {n_fail}/{n_total} failed).

FAILURES (priority — fix these first):

For each FAILED episode:

- What specific step did the policy fail at and why?
- What information did the policy need that ϕ's state did not contain?
- Was this a ϕ tracking failure or a missing capability?

COMPRESSION OPPORTUNITIES (secondary):

For the SUCCESSFUL episodes, identify any state fields that are redundant or oversized and safe to remove without breaking correctness.

Be concrete and specific in both sections.

S is FROZEN. Do not suggest changes to `parse_step` or S fields. Focus only on ϕ.

Do NOT recommend matching the teacher's specific action sequence.

When all shown episodes fail, the reflector is instead instructed to diagnose why the candidate fails across the set. When all episodes succeed, it is instructed to focus on compression opportunities in ϕ's state.

C.2 Proposer Prompt

For mutation, the proposer receives the current state-update program ϕ, the reflector feedback, example trajectories, and the frozen schema S. Here, S is a fixed interface schema containing only the current observation and preceding action; `parse_step` simply populates these fields and does not encode task-specific state knowledge. The prompt used in our experiments is reproduced below, with $\{\ldots\}$ denoting task- or candidate-specific content.

Contract.

Implement a single Python function `def phi(state, event, S, goal):` that folds ONE new parsed event into a persistent state, carried forward across many calls, one per time step.

`state` is whatever was returned on the previous call, or an initial empty dictionary on the first step. `event` contains the parsed fields for the current step. `S` is a dictionary containing `S["fields"]`, which maps field names to descriptions. You are not required to return exactly these keys: you may track additional derived fields of your own, such as a running count or whether some event has ever occurred, and you may omit unused fields from S.

`goal` is a plain string describing the task. It may be used to decide what information is worth tracking, but it does not change within an episode.

Do not mutate the input arguments.

{environment_description}

Objective.

What is being optimized: TASK SUCCESS RATE.

The fixed policy reads ϕ's compact state instead of the full raw interaction history available to the teacher reference. Success is the primary objective, while state size provides secondary pressure.

The teacher's specific trajectory is only one example of a successful path. Different valid trajectories may solve the same task. Do NOT recommend matching the teacher's step-by-step actions. Focus on whether the candidate solves the task and, when it fails, why: what information was missing that the policy needed to choose a better action. Trajectory-shape matching is not the objective; task completion is.

Example raw episodes (format grounding):
{demo_episodes}

S fields (frozen; ϕ receives these keys in `event`):
{s_fields}

Current ϕ:
{phi_code}

Reflection on this pairing's behavior:
{reflection_text}

Size pressure on ϕ's output.

Keep ϕ's returned state SMALL.

If the reflector has identified specific information that was missing and caused task failure, preserve that information even if it increases state size. Correctness first, compaction second.

The state returned by ϕ is JSON-serialized and fed to the frozen policy at every step; its per-step token count directly penalizes fitness.

- Do NOT store derived or redundant values that can be recomputed from primitives.
- Do NOT keep accumulating stale entries.
- Prune fields whose usefulness has passed.
- Do NOT duplicate task information across multiple fields.
- Prefer compact keys and values because the state is fed to the policy as text.

Proposer instruction.

Propose an improved ϕ based on the reflection. Make targeted changes that directly address the issues identified. Preserve the overall structure and logic of the current ϕ where the reflection does not indicate a problem. Do not restructure or rewrite parts that are working correctly.

Return your answer as a single Python code block containing just the complete `phi` function, with no other text.

For crossover, the same prompt structure is used with two parent programs provided instead of one.