

000 001 002 003 004 005 006 007 008 009 010 011 012 013 014 015 016 017 018 019 020 021 022 023 024 025 026 027 028 029 030 031 032 033 034 035 036 037 038 039 040 041 042 043 044 045 046 047 048 049 050 051 052 053 EvoTest: EVOLUTIONARY TEST-TIME LEARNING FOR SELF-IMPROVING AGENTIC SYSTEMS

Anonymous authors

Paper under double-blind review

ABSTRACT

A fundamental limitation of current AI agents is their inability to learn complex skills on the fly at test time, often behaving like “clever but clueless interns” in novel environments. This severely limits their practical utility. To systematically measure and drive progress on this challenge, we first introduce the **Jericho Test-Time Learning (J-TTL)** benchmark. J-TTL is a new evaluation setup where an agent must play the same game for several consecutive episodes, attempting to improve its performance from one episode to the next. On J-TTL, we find that existing adaptation methods like reflection, memory, or reinforcement learning struggle. To address the challenges posed by our benchmark, we present **EvoTest**¹, an evolutionary test-time learning framework that improves an agent without any fine-tuning or gradients—by evolving the entire agentic system after every episode. EvoTest has two roles: the **Actor Agent**, which plays the game, and the **Evolver Agent**, which analyzes the episode transcript to propose a revised configuration for the next run. This configuration rewrites the prompt, updates memory by logging effective state-action choices, tunes hyperparameters, and learns the tool-use routines. On our J-TTL benchmark, EvoTest consistently increases performance, outperforming not only reflection and memory-only baselines but also more complex online fine-tuning methods. Notably, our method is the only one capable of winning two games (Detective and Library), while all baselines fail to win any.

1 INTRODUCTION

The pursuit of truly autonomous agents hinges on a critical human capability: the ability to learn “on the fly” (Maes, 1993; Franklin & Graesser, 1996). When faced with a new task, humans can attempt it, reflect on their successes and failures, formulate a better strategy, and try again. By contrast, most AI agents arrive at deployment with a fixed policy, behaving like “clever but clueless interns” that can execute instructions but cannot reform their own process from experience (Huang et al., 2024; Talebirad & Nadiri, 2023; Wang et al., 2024). This gap severely limits their reliability in dynamic settings. While the field acknowledges this problem, progress has been hampered by a lack of standardized testbeds designed specifically to measure an agent’s capacity for rapid, in-session improvement (Zhou et al., 2023; Mialon et al., 2023; He et al., 2025).

This gap defines our core scientific problem: **Test-Time Learning (TTL)** in the context of long-horizon tasks. To address this, we first introduce the **Jericho Test-Time Learning (J-TTL)** benchmark, a new evaluation framework designed to systematically measure and drive progress in on-the-fly agent learning. The benchmark’s core task is straightforward: an agent must play the same complex, text-based adventure game (Hausknecht et al., 2020) for a series of consecutive attempts (“episodes”). In each episode, the agent interacts with the environment through a standard loop: it receives a textual observation of its surroundings (state), submits a natural-language command (action), and receives a numerical score change (reward). These games are difficult for LLM agents because they feature complex puzzles, long-range planning, sparse rewards (many critical actions yield no points), and irreversible consequences (a single wrong move can make the game unwinnable). The agent’s goal is structured at two levels: 1) The *Episodic Goal*: Maximize the final score within

¹The code is available at <https://anonymous.4open.science/status/evotest-38C1>

054 a single playthrough. 2) The *Learning Goal*: Play the same game repeatedly and progressively in-
 055 crease its final score from one episode to the next, using only the experience gathered within that
 056 single session.

057 The J-TTL benchmark starkly reveals the inadequacies of existing adaptation paradigms. Consider
 058 a simple but critical failure in the game *Detective*: an agent gets stuck in a navigation loop by repeat-
 059 ingly attempting an invalid action, such as GO WEST, which the game rejects with "You can't
 060 go that way." This seemingly simple failure reveals deep flaws in current adaptation methods:
 061 A **Static** agent has no learning mechanism and will likely repeat this error in every episode, leading
 062 to a flat, low-scoring performance. An **SFT (online)** agent will have no good data to learn from in
 063 this failed episode. It is trapped because it cannot generate the very data it needs to improve. An
 064 **Reinforcement Learning (RL)(online)** agent receives a `reward=0` for the invalid move, which
 065 is a weak signal in a sparse-reward environment. A single update based on this noisy signal is
 066 insufficient to correct the policy, demonstrating a failure of credit assignment. Methods based on
 067 **reflection**, such as *Reflexion* (Shinn et al., 2023), modify the agent's prompt with summaries of
 068 past failures. While useful, it does not alter the agent's core decision-making logic or its use of
 069 tools. Similarly, advanced **memory systems** (Packer et al., 2023; Zhong et al., 2024) improve an
 070 agent's ability to recall information but do not teach it how to act differently. On the other end of the
 071 spectrum, RL and online fine-tuning are fundamentally ill-suited for the test-time learning setting.
 072 These methods are too slow and data-inefficient for the rapid learning J-TTL demands. To meet
 073 the challenge posed by our benchmark, we introduce **EvoTest**, an **evolutionary test-time learn-**
 074 **ing framework designed for rapid, holistic adaptation without fine-tuning**. EvoTest decouples
 075 acting from adaptation using two distinct roles: an **Actor Agent** that plays a full episode and an
 076 **Evolver Agent** that improves the system between independent episodes. After each episode, the
 077 Evolver Agent analyzes the full transcript and proposes a revised configuration for the entire agentic
 078 system. This process of whole-system evolution involves:

- 078 1. Rewriting the guiding prompt to encode new strategies;
- 079 2. Updating a structured deployment-time memory with records of successful and failure actions;
- 080 3. Tuning decision-making hyperparameters like temperature and exploration strength;
- 081 4. Refining the tool-use routines that govern how and when memory or python code is accessed.

082 By evolving the agent configuration, EvoTest transforms the narrative of one episode into multi-
 083 faceted improvements for the next attempt, enabling a deeper form of learning than prior methods.
 084 We summarize our contributions as follows:

- 085 • **A Benchmark for Test-Time Learning:** We propose J-TTL, a benchmark using Jericho games
 086 to measure an agent's on-the-fly learning ability across a series of playthroughs of the same game.
- 087 • **A Test-time Learning Algorithm:** We propose *EvoTest*, an evolutionary agent learning frame-
 088 work that evolves the entire agentic system (policy, memory, tool-use routines, and hyperparame-
 089 ters) via transcript-level analysis without gradients or fine-tuning.
- 090 • **State-of-the-Art Empirical Results:** We demonstrate on the J-TTL benchmark that *EvoTest*
 091 shows a 38% improvement over the strongest prompt-evolution baseline and a 57% improvement
 092 over online RL, outperforming all strong reflection-based, memory-based, and gradient-based
 093 baselines on every game.

094 2 RELATED WORK

095 **From Static Agents to Test-Time Learning.** The majority of current AI agents, while capable, op-
 096 erate with static configurations that are manually designed and fixed after deployment (Wang et al.,
 097 2024; Xi et al., 2025). This limits their ability to adapt to novel situations, a key challenge motivating
 098 the development of "self-improving AI agents" (Gao et al., 2025; Fang et al., 2025). A prominent
 099 line of work enables agents to learn from past mistakes without updating weights. *Reflexion* (Shinn
 100 et al., 2023), a key baseline for our work, allows an agent to verbally reflect on trajectory failures and
 101 append these reflections to its prompt for subsequent episodes. Other approaches focus on enhanc-
 102 ing agent memory. For instance, *MemGPT* (Packer et al., 2023) provides agents with a structured
 103 memory system to manage long contexts. Beyond reflection/memory, *Uncertainty of Thoughts* (Hu
 104 et al., 2024) adds test-time *uncertainty-aware planning*, deciding when to ask, verify, or revise with-
 105 106 107

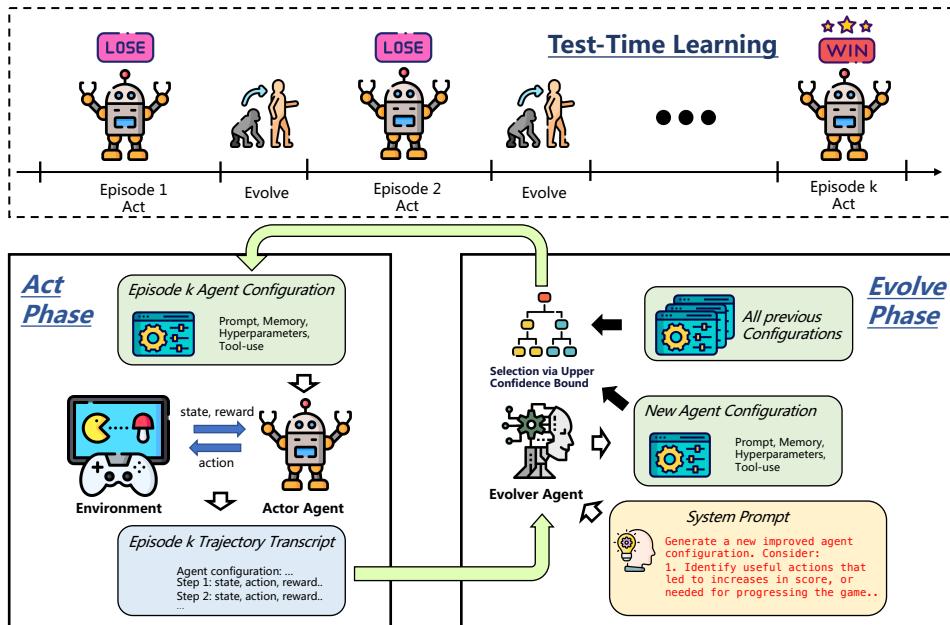


Figure 1: The **EvoTest** architecture, designed to enable **test-time learning** (TTL). The agent operates in a continuous Act-Evolve loop across multiple attempts at the same task. After each episode, the Evolver Agent analyzes the full trajectory transcript—rich narrative feedback to perform gradient-free, **whole-system evolution** on the agent’s entire configuration. This allows the agentic system to self-improve on the fly, directly from its own experience at test time.

out weight updates. While MemoryBank (Zhong et al., 2024) uses hierarchical summarization to retain information over long interactions. While powerful, these methods typically adapt a single component of the agentic system—either by adding reflective text to a fixed prompt or by populating a memory store. The core strategy, including the guiding policy, hyperparameters, and the rules for memory access, remains static.

Self-Evolving Agentic Systems. Another active area of research is the automated optimization of prompts that guide agent behavior. Generative approaches like APE (Zhou et al., 2022) and OPRO (Yang et al., 2023) use a powerful LLM to propose and score new prompts, iteratively refining them based on performance. Gradient-inspired methods like TextGrad (Yuksekgonul et al., 2024) refine prompts using LLM-generated textual feedback. Closely related to our work are evolutionary methods such as AlphaEvolve (Novikov et al., 2025), Promptbreeder (Fernando et al., 2023), and EvoPrompt (Guo et al., 2024), which maintain a population of prompts and apply genetic operators like mutation and crossover to discover more effective instructions. EvoTest generalizes prompt evolution to whole-system evolution, optimizing the entire agentic configuration—including the prompt, memory, hyperparameters, and tool-use routines. This allows for more holistic adaptations, such as tuning exploration strength, that are beyond the scope of prompt-editing alone. This vision for unified optimization is shared by EvoAgent (Yuan et al., 2024) and MASS (Zhou et al., 2025); Beyond ‘Aha!’ (Hu et al., 2025) complements this by aligning meta-abilities rather than only task prompts or single components.

3 THE JERICHO TEST-TIME LEARNING (J-TTL) BENCHMARK

To systematically measure and drive progress in on-the-fly agent learning, we introduce the **Jericho Test-Time Learning (J-TTL)** benchmark. This benchmark is built upon the Jericho (Hausknecht et al., 2020)¹ suite of Interactive Fiction (IF) games. IF games are fully text-based simulation environments where an agent issues text commands to effect change in the environment and progress through a story. While the richness of these environments makes them a challenging testbed for AI, existing evaluation has primarily focused on single-episode performance or generalization across

¹Jericho is available at <https://github.com/Microsoft/jericho>

162 different games (Hausknecht et al., 2020; Gulcehre et al., 2020; Li et al., 2025). The J-TTL benchmark refocuses the evaluation on a different, critical axis: an agent’s ability to learn and improve its
 163 strategy through repeated attempts at the same complex task within a single test session.
 164

166 **Datasets.** We use publicly available Jericho games that vary in difficulty and puzzle structure,
 167 including *Detective*, *Library*, *Zork1*, *Zork3*, *Balances*, and *Temple*. Games are launched via Jericho
 168 with default scoring. Each episode is capped by a step limit ($T = 110$ unless stated otherwise).

169 **The Jericho Game.** We model a Jericho game (Hausknecht et al., 2020) as a Partially Observable
 170 Markov Decision Process (POMDP), defined by the tuple $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \Omega, T)$. Here, \mathcal{S} is the latent
 171 state space, and \mathcal{A} is the (infinite) combinatorial action space of natural language commands. At
 172 each step t , an agent in a latent state $s_t \in \mathcal{S}$ takes an action $a_t \in \mathcal{A}$, causing a transition to a new
 173 state $s_{t+1} \sim \mathcal{T}(\cdot | s_t, a_t)$ and yielding a scalar reward $r_t = \mathcal{R}(s_t, a_t)$. The agent does not observe
 174 the true state s_t but instead receives a textual observation $o_t \sim \Omega(\cdot | s_t)$. An episode is a trajectory
 175 of interactions with a finite horizon of T steps:

$$\tau^{(e)} \triangleq \left(o_1^{(e)}, a_1^{(e)}, r_1^{(e)}, \dots, o_T^{(e)}, a_T^{(e)}, r_T^{(e)} \right). \quad (1)$$

178 The total return for an episode e is the sum of its rewards, $R(e) \triangleq \sum_{t=1}^T r_t^{(e)}$.
 179

180 **Test-Time Learning Process on Jericho.** The J-TTL benchmark protocol consists of a **session** of K
 181 consecutive **episodes** played in a single game. After each episode concludes, the game environment
 182 is reset to its identical initial state, $s_0^{(e)} = s_{\text{init}}$ for all $e \in \{1, \dots, K\}$. This ensures that any
 183 performance improvement is solely attributable to the agent’s internal learning process.
 184

185 Let the agent be parameterized by a set of learnable components $\theta \in \Theta$. In episode e , the agent’s
 186 behavior is governed by a policy conditioned on its current learnable components, $\pi_{\theta^{(e)}}$.
 187

188 The core of test-time learning lies in the update rule an agent applies between episodes. After com-
 189 pleting episode e and collecting the trajectory data $\tau^{(e)}$, the agent updates its learnable components
 190 for the next episode:
 191

$$\theta^{(e+1)} = U(\theta^{(e)}, \tau^{(e)}), \quad (2)$$

192 where $U : \Theta \times \mathcal{T}_{\text{hist}} \rightarrow \Theta$ is the agent’s learning algorithm.
 193

194 **Evaluation Metrics.** For a test-time learning session consisting of K episodes, we record total score
 195 for each episode, $R(e)$. This yields a performance sequence for the entire session:
 196

$$\{R(1), R(2), \dots, R(K)\}. \quad (3)$$

197 From this sequence, we derive two metrics:
 198

- 200 • **Learning Curve.** A plot of the final return $R(e)$ against the episode index e . This curve provides
 201 a direct visualization of an agent’s learning progress.
- 202 • **Area Under the Curve (AUC).** For quantitative comparison, we define the AUC as a normalized
 203 ratio. It measures the agent’s total achieved score against the maximum possible score over the
 204 session:
 205

$$\text{AUC} = \frac{\sum_{e=1}^K R(e)}{K \cdot R_{\max}}, \quad (4)$$

207 where R_{\max} is the maximum achievable score in a single episode of the game. This metric yields
 208 a value between 0 and 1, facilitating comparison across games with different scoring scales.
 209

210 4 THE EVO TEST FRAMEWORK

212 To address the challenges posed by the J-TTL benchmark, we introduce **EvoTest**, an evolutionary
 213 test-time learning framework, as illustrated in Figure 1. Unlike methods that perform gradient-based
 214 updates on model weights, EvoTest operates on a fixed, non-trainable backbone LLM. It achieves
 215 learning by evolving the agent’s entire high-level configuration between episodes, leveraging the
 rich, narrative feedback from game transcripts rather than just sparse, scalar rewards.

216 4.1 THE AGENTIC CONFIGURATION
217

218 In the context of EvoTest, the general learnable components θ from the J-TTL formulation (Section 3) are instantiated as a holistic **agentic configuration**, denoted by $\chi \in \mathcal{X}$. This configuration is
219 a tuple $\chi = (p, M, h, u)$ that defines the agent’s complete operational strategy:
220

- 221 • **Policy Prompt (p):** A system prompt that provides high-level strategic guidance, heuristics, and
222 behavioral guardrails to the backbone LLM.
- 223 • **Deployment-time Memory (M):** A structured, queryable database populated by the Evolver
224 Agent after each episode. It stores the agent’s prior experiences, effectively creating a persistent
225 knowledge base. The memory is organized into distinct components, such as: (a) a *success memory*
226 that logs state-action pairs which led to score increases (e.g., `(state_hash, action) -> score_delta`), and (b) a *failure memory* that records patterns associated with stalls or neg-
227 ative outcomes (e.g., repetitive action loops in a specific location). This allows the agent to recall
228 specific, effective actions and avoid known pitfalls from past playthroughs. **For a detailed break-
229 down of the memory’s data structure and concrete examples of its contents, please see Appendix J.**
230
- 231 • **Hyperparameters (h):** A set of parameters controlling the LLM’s inference and the agent’s
232 decision-making, such as temperature, exploration strength, and stopping criteria.
- 233 • **Tool-Use Routines (u):** Active components executed at each decision-making step to operational-
234 ize knowledge and create useful state abstractions. These routines consist of two functions:
235
 - 236 – **Memory Interaction Logic:** A set of rules governing how to query the memory (M). Before
237 deciding on an action, this routine might check if the current game state exists in memory. If a
238 match is found in the success memory, the routine can inject the proven action into the LLM’s
239 prompt as a strong suggestion (e.g., “Hint: In this exact situation before, the action ‘unlock door
with key’ worked well.”).
 - 240 – **State Abstraction Logic:** An evolvable Python function—the *state extractor*—that processes
241 the raw, verbose game history into a concise summary of progress. Instead of forcing the
242 LLM to re-read the entire history, this tool parses the log for key environmental cues (e.g., the
243 text “The ancient scroll disintegrates, revealing a map.”) and returns
244 a short, meaningful milestone string (e.g., “Milestone: Found the map.”). This ab-
245 stracted state is included in the prompt at every step, providing efficient situational awareness.

246 The agent’s policy in episode e , $\pi_{\chi^{(e)}}$, is therefore a function of the fixed LLM’s behavior as modu-
247 lated by this entire configuration.
248

249 4.2 THE TWO-AGENT LEARNING LOOP
250

251 EvoTest operationalizes the test-time learning update rule, $\chi^{(e+1)} = U(\chi^{(e)}, \tau^{(e)})$, through a coop-
252 erative two-agent design that separates acting from adaptation.
253

254 **1. The Actor Agent.** For a given episode e , the Actor Agent is provided with a single, fixed config-
255 uration $\chi^{(e)}$. It plays through the episode by repeatedly querying the backbone LLM, conditioning
256 its actions on the current observation o_t and any information retrieved from its memory $M^{(e)}$ as
257 dictated by its tool-use routines $u^{(e)}$. The result of its playthrough is the trajectory $\tau^{(e)}$ and the final
258 return $R(e)$.
259

260 **2. The Evolver Agent.** After the episode, the Evolver Agent takes the trajectory transcript $\tau^{(e)}$
261 and the parent configuration $\chi^{(e)}$ as input. It performs **whole-system evolution** by generating a
262 set of new candidate child configurations, $C^{(e+1)} = \{\tilde{\chi}_1^{(e+1)}, \dots, \tilde{\chi}_m^{(e+1)}\}$. This is the core learn-
263 ing step, where the Evolver uses an LLM to analyze the previous episode’s trajectory and propose
264 improvements to the agentic configuration. The evolutionary operators include:
265

- 266 • **Prompt Mutation:** The Evolver rewrites the policy prompt $p^{(e)}$ to create a new prompt \tilde{p} . It incor-
267 porates strategies that proved effective (e.g., adding “always examine objects before taking them”) or adds explicit rules to prevent observed failures (e.g., “avoid repetitive navigation loops”).
268
- 269 • **Memory Update:** The Evolver programmatically parses the transcript $\tau^{(e)}$ to identify and log im-
portant events. It records state-action pairs (o_t, a_t) that immediately preceded a score increase in a

270 “success” table. It also identifies sequences of interactions that led to no progress and records them
 271 in a “failure” table. This updated memory $M^{(e+1)}$ is then inherited by all child configurations.
 272

- 273 • **Hyperparameter Tuning:** The Evolver proposes adjustments to the hyperparameters $h^{(e)}$ to cre-
 274 ate \tilde{h} . For example, if the transcript shows the agent was stuck in a repetitive loop, the Evolver
 275 might suggest increasing the LLM’s ‘temperature’ to encourage more diverse actions.
- 276 • **Tool-Use Refinement:** The Evolver can modify the logic in $u^{(e)}$ to create \tilde{u} , changing when
 277 or how the agent consults its memory. For example, it might strengthen a rule in the prompt
 278 from a general suggestion to a direct instruction, such as telling the agent it must check its success
 279 memory before every action. **The specification of these tools and the required JSON output format**
 280 **are detailed in the master prompt provided in Appendix H.**

281 A child configuration $\tilde{\chi}$ is a new combination of these potentially mutated components, representing
 282 a new hypothesis for a more effective agent.

284 4.3 CONFIGURATION SELECTION VIA UCB

286 After the Evolver generates a set of new “child” configurations, EvoTest decide which one to use for
 287 the next episode. This creates a classic dilemma: should we *exploit* a configuration that has worked
 288 well in the past, or should we *explore* a new, untested one that might be even better?

289 To manage this trade-off, we select the next configuration, $\chi^{(e+1)}$, from a candidate pool containing
 290 the parent from the previous episode, $\chi^{(e)}$, and its new children, $C^{(e+1)}$. The selection is guided by
 291 the Upper Confidence Bound (UCB) algorithm, a strategy from multi-armed bandit theory (Vermorel
 292 & Mohri, 2005) designed to manage the exploration-exploitation dilemma. The rule selects the
 293 configuration that maximizes the following score:

$$295 \chi^{(e+1)} = \arg \max_{\tilde{\chi} \in \{\chi^{(e)}\} \cup C^{(e+1)}} \left(\hat{\mu}(\tilde{\chi}) + \beta \sqrt{\frac{\log N}{1 + n(\tilde{\chi})}} \right) \quad (5)$$

298 The score for each configuration $\tilde{\chi}$ is a sum of two parts:

- 300 • **Performance Term ($\hat{\mu}(\tilde{\chi})$):** This is simply the average score the configuration has achieved so
 301 far. It encourages us to re-use configurations that have a good track record.
- 302 • **Exploration Bonus ($\beta \sqrt{\dots}$):** This term gives a boost to configurations that are less certain about.
 303 $n(\tilde{\chi})$ is the number of times a configuration has been tried, N is the total number of episodes
 304 completed, and β is a hyperparameter controlling the exploration strength. The bonus is higher for
 305 configurations that have been tried fewer times ($n(\tilde{\chi})$ is small), making novel mutations attractive.

306 At the start of each new episode, the UCB algorithm selects a single configuration, which is then used
 307 for the entire duration of that episode. The episode’s final score is then used to update the statistics
 308 for that chosen configuration. Crucially, this UCB approach also makes our learning process more
 309 stable and helps prevent sharp drops in performance. A simpler, greedy strategy might get fooled by
 310 a new configuration that gets a lucky high score and then repeatedly fails. UCB avoids this pitfall.
 311 Because the reliable parent configuration is always in the selection pool, if a new child performs
 312 poorly after its initial trial, its average score ($\hat{\mu}$) will drop. The UCB rule can then naturally “fall
 313 back” to the time-tested parent, which retains its high score. This acts as a safety net, preventing the
 314 system from getting stuck on a bad evolutionary path and ensuring a more consistent improvement.

316 5 EXPERIMENTS

318 Our experiments are designed to answer three central research questions regarding test-time learning
 319 on our J-TTL benchmark:

- 321 • **RQ1:** Does test-time learning (TTL) lead to meaningful performance improvements on complex
 322 tasks compared to a non-learning agent?
- 323 • **RQ2:** Does our proposed EvoTest framework enable more effective test-time learning compared
 324 to existing methods, such as those based on memory, reflection, and prompt optimization?

324 Table 1: Comparison of Area Under the Curve (AUC) scores on the J-TTL benchmark
 325 across six Jericho games for two backbone LLMs: `google/gemini-2.5-flash` (*G*) and
 326 `anthropic/clause-4-sonnet-20250522` (*C*). Weight-update methods use a separate
 327 backbone and are not distinguished by model. Higher values indicate better overall performance
 328 throughout the test-time learning session. The best performance in each column is highlighted in
 329 **bold**. Our method, **EvoTest**, consistently outperforms all baselines across both backbones.

Method	<i>Detective</i>		<i>Library</i>		<i>Zork1</i>		<i>Zork3</i>		<i>Balances</i>		<i>Temple</i>		<i>Avg.</i>	
	<i>G</i>	<i>C</i>	<i>G</i>	<i>C</i>	<i>G</i>	<i>C</i>	<i>G</i>	<i>C</i>	<i>G</i>	<i>C</i>	<i>G</i>	<i>C</i>	<i>G</i>	<i>C</i>
<i>Non-learning Baseline</i>														
Static	0.21	0.23	0.15	0.16	0.03	0.04	0.05	0.06	0.11	0.12	0.08	0.09	0.11	0.12
<i>Memory-based & Reflection-based Methods</i>														
Memory	0.25	0.26	0.18	0.20	0.04	0.05	0.06	0.07	0.13	0.14	0.10	0.11	0.13	0.14
RAG	0.32	0.34	0.24	0.25	0.07	0.08	0.09	0.10	0.18	0.20	0.15	0.16	0.18	0.19
Summary	0.45	0.47	0.33	0.35	0.12	0.13	0.15	0.16	0.25	0.27	0.21	0.22	0.25	0.27
Reflexion	0.58	0.60	0.41	0.44	0.09	0.11	0.25	0.27	0.30	0.32	0.29	0.31	0.32	0.34
<i>Automated Prompt Optimization Methods</i>														
TextGrad	0.61	0.62	0.45	0.47	0.11	0.13	0.28	0.30	0.16	0.18	0.23	0.25	0.31	0.33
Promptbreeder	0.63	0.65	0.47	0.49	0.10	0.12	0.29	0.31	0.23	0.25	0.30	0.32	0.34	0.36
EvoPrompt	0.65	0.67	0.48	0.50	0.10	0.12	0.30	0.32	0.24	0.26	0.27	0.29	0.34	0.36
<i>Weight-Update Methods (Online Fine-Tuning)</i>														
SFT (online)	0.40		0.30		0.11		0.18		0.22		0.19		0.23	
GRPO (online)	0.55		0.38		0.07		0.22		0.30		0.26		0.30	
EvoTest (Ours)	0.94	0.95	0.77	0.80	0.14	0.16	0.35	0.38	0.32	0.35	0.31	0.34	0.47	0.50

348 • **RQ3:** How does the gradient-free, evolutionary approach of EvoTest compare to traditional
 349 gradient-based RL methods in the context of test-time learning?

351 5.1 SETUP

353 **Backbone LLM.** We use two powerful API models for the Actor Agent:
 354 the cost-effective `google/gemini-2.5-flash` and the highly capable
 355 `anthropic/clause-4-sonnet-20250522`. The Evolver Agent, which performs the
 356 most complex reasoning, is powered by `openai/o3-2025-04-16`. The fine-tuning base-
 357 lines (SFT and GRPO) are implemented on `qwen/qwen3-32b`, a large open-source model.
 358 Experiments with more LLM backbones can be found in Appendix N

359 **Baselines.** As a foundational reference, a non-learning **Static** agent with a fixed configuration es-
 360 tablishes zero-shot performance. The learning baselines are grouped into four main categories:
 361 (1) **Memory-based Methods**, which include (1a) **Memory**, which places the complete session trans-
 362 script history into the context for in-context learning, automatically truncating the oldest parts if the
 363 context limit is exceeded, and (1b) **RAG**, which retrieves relevant snippets from past trajectories;
 364 (2) **Reflection-based Methods**, which learn by appending textual feedback to the prompt, including
 365 (2a) **Summary**, which uses an LLM to progressively summarize the entire history of all past trans-
 366 scripts and feeds this condensed summary into the context, and (2b) **Reflexion** (Shinn et al., 2023),
 367 which generates structured textual self-reflections after each episode to critique performance and
 368 guide the next attempt; (3) **Automated Prompt Optimization Methods**, which iteratively refine the
 369 guiding prompt, including (3a) **TextGrad** (Yuksekgonul et al., 2024), where after each episode, an
 370 optimizer LLM analyzes the trajectory to generate a “textual gradient”—a critique describing how
 371 to improve the prompt—which is then applied for refinement, and two evolutionary approaches,
 372 (3b) **Promptbreeder** (Fernando et al., 2023) and (3c) **EvoPrompt** (Guo et al., 2024), which evolve
 373 a population of prompts; and (4) **Weight-Update Methods**, which contrast with our gradient-free
 374 approach by performing online fine-tuning, including (4a) **SFT (online)**, which performs Super-
 375 vised Fine-Tuning on the actor model after each episode using the state-action pairs collected from
 376 the trajectory, and (4b) **GRPO (online)** (Shao et al., 2024), which applies gradient-based Reinforce-
 377 ment Learning policy updates to the model using the scalar rewards collected during the episode.
 378 All methods are evaluated under the same step budget and use the same backbone LLMs for their
 379 respective roles to ensure a fair comparison. Our EvoTest begins each session with a generic, game-

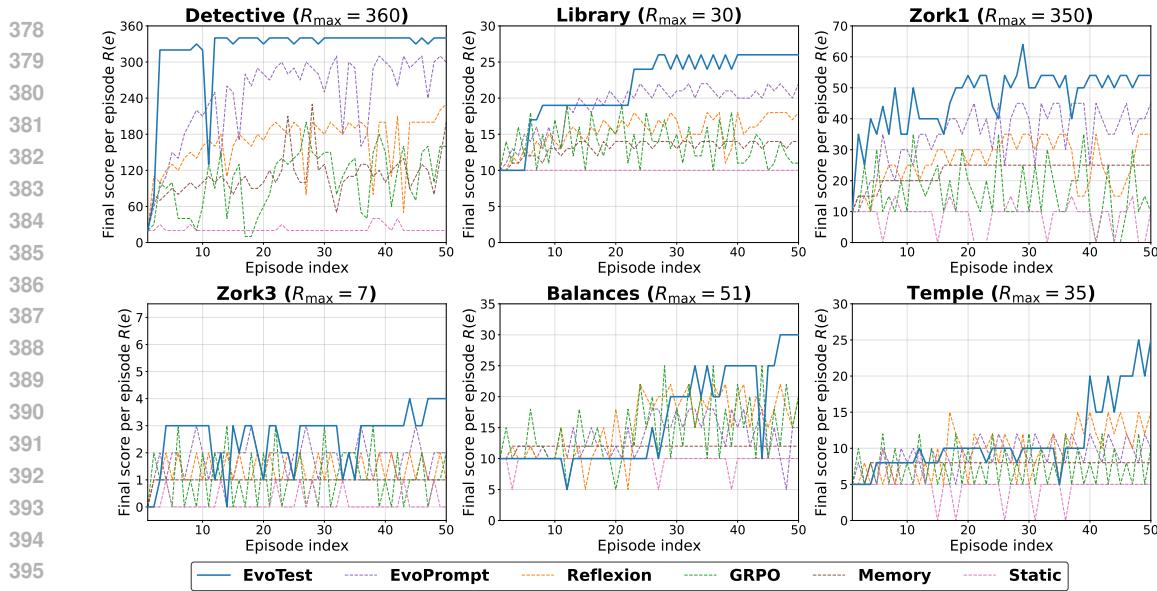


Figure 2: Learning curves showing final score per episode across six Jericho games with `google/gemini-2.5-flash` as LLM. EvoTest consistently demonstrates a steeper and more stable learning trajectory compared to baselines, validating its effectiveness for test-time learning. agnostic initial configuration, which includes a simple prompt (“Explore systematically and examine objects to make progress.”), an empty memory, and tool-use routines. The detailed introduction can be found in appendix C, D.

5.2 RESULTS

Table 1 presents the AUC scores for all methods across the six selected Jericho games, while the learning curves in Figure 2 illustrate the per-episode performance progression. We analyze these results through the lens of our three research questions.

RQ1: Test-Time Learning is Effective. The results provide a clear affirmative answer. Across all games, every learning-based method achieves a higher average AUC score than the **Static** baseline (Table 1). The learning curves (Figure 2) further confirm this, showing that methods capable of learning from experience consistently exhibit upward-trending scores, whereas the Static agent’s performance remains flat. This demonstrates that even with just a few attempts, test-time learning is a valid and effective paradigm for improving agent performance on complex, long-horizon tasks.

RQ2: EvoTest Outperforms Existing Adaptation Methods. EvoTest consistently and substantially outperforms all other gradient-free baselines. In Table 1, EvoTest achieves the highest AUC score on all six games, with an average score of 0.47/0.50—a significant improvement over the next best baseline, EvoPrompt (0.34/0.36). The learning curves reveal not just higher final scores but a steeper rate of improvement, indicating more efficient learning.

The key insight lies in the limitation of single-channel adaptation. **Memory** and **RAG** provide raw information but offer no strategic guidance, leading to a low performance ceiling. **Reflexion** and other prompt-focused optimizers like **Promptbreeder** perform better by refining strategy, but they are constrained to a single axis: the prompt. An agent can have a perfect prompt but still fail due to poor exploration (e.g., low temperature) or inefficient use of its knowledge. EvoTest’s strength is its **holistic, whole-system evolution**. By concurrently optimizing the prompt, memory-access routines, and decision hyperparameters, it discovers and resolves complex performance bottlenecks that single-channel adaptations cannot. For example, it can learn to increase exploration temperature in early episodes and simultaneously add a new strategic heuristic to its prompt based on its findings, a multi-faceted adaptation that other methods are incapable of, as detailed and plotted in Appendix M.

RQ3: Evolutionary Adaptation is More Data-Efficient than RL at Test Time. The comparison between EvoTest and the weight-update methods clearly favors the evolutionary approach in this set-

432
433 Table 2: Practical costs for a single learning update.
434

Method	Update Time	LLM Calls
SFT (online)	5–10 min	0
GRPO (online)	5–10 min	0
TextGrad	30–50 sec	2
EvoTest (Ours)	20–30 sec	1
EvoPrompt	20–30 sec	1
Reflexion	15–25 sec	1
RAG	5–15 sec	0 (emb.)
Memory	<1 sec	0

444
445 Table 3: Ablation study on EvoTest components,
446 showing Area Under the Curve (AUC) scores.
447

	<i>Detective</i>	<i>Zork1</i>	<i>Balances</i>
EvoTest	0.94	0.14	0.32
w/o Prompt	0.52	0.05	0.16
w/o UCB	0.68	0.08	0.22
w/o Memory	0.82	0.11	0.28
w/o Hyperpara.	0.89	0.12	0.30
w/o Tool-Use	0.91	0.13	0.30

455 ting. EvoTest’s average AUC (0.47/0.50) is substantially higher than that of **GRPO (online)** (0.30),
456 the gradient-based RL baseline. This improvement indicates that EvoTest successfully addresses a
457 fundamental challenge in test-time learning: extreme data scarcity.

458 Moreover, EvoTest alleviate traditional RL’s reliance on scalar rewards for credit assignment. In
459 sparse-reward environments like Jericho, a single episode provides a noisy and insufficient signal
460 for effective gradient updates. It is an inefficient way to learn from one complex success or failure.
461 In contrast, EvoTest bypasses this issue by leveraging the **entire episode transcript as a rich, narrative**
462 **feedback signal**. The Evolver Agent performs credit assignment through semantic analysis
463 of the game’s story, identifying causal chains of failure (e.g., “the agent got stuck in a loop here”) and
464 success. This allows it to make explicit, targeted, and structural edits to the agent’s configuration.
465 In essence, EvoTest shifts from credit assignment via backpropagation to **credit assignment via narrative analysis**, a more data-efficient mechanism for learning from a single experience.

467 5.3 MODEL ANALYSIS

470 **Ablation on Key Components.** Our ablation study reveals the distinct contributions of each component
471 in the EvoTest framework. The AUC scores in Table 3 quantify the overall impact, showing that removing any component degrades performance. The largest performance drop occurs when removing prompt evolution (*w/o Prompt*), confirming that evolving the high-level policy is the primary driver of strategic adaptation.

475 The learning curves in Figure 3 offer deeper insight into the *dynamics* of these failures, particularly for the UCB ablation. While removing UCB also causes a significant drop in AUC, the curve reveals a more nuanced issue: instability. The *w/o UCB* agent, which uses a greedy selection strategy, is prone to catastrophic drops in performance. This happens when the agent over-commits to a high-risk mutation that achieved a lucky high score in one episode but is not robust. Without UCB’s exploration mechanism—which encourages revisiting more reliable past configurations—the agent gets stuck on a suboptimal evolutionary path and is unable to correct its course after a poor choice. In contrast, the full EvoTest model leverages UCB to maintain a stable learning trajectory.

483 **Efficiency Analysis.** For learning at test-time, the update step between episodes is a major bottleneck. Our experiments show a clear divide in practicality between different approaches, as detailed in Table 2. Weight-update methods like online RL are not practical for this setting. A fine-tuning pass on one episode’s data took 5 to 10 minutes on 4 H100 GPUs. This is not just slow; it demands

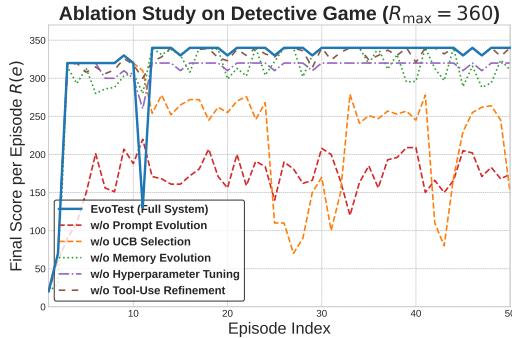
Figure 3: Learning curves from the component ablation study on the *Detective* game.

Table 4: Ablation on the Evolver agent’s LLM. Performance, measured by AUC, correlates with model quality.

Evolver LLM	<i>Detective</i>	<i>Zork1</i>	<i>Balances</i>
openai/o3	0.94	0.14	0.32
deepseek/r1	0.90	0.12	0.29
qwen3-32b	0.82	0.10	0.25
qwen3-8b	0.68	0.07	0.20
<i>Static</i>	0.21	0.03	0.11

Table 5: Ablation on the structure of prompt evolution. “Simple Mutation” uses a generic improvement instruction, while “Full” uses our structured, multi-part master prompt. Scores are AUC.

	<i>Detective</i>	<i>ZorkI</i>	<i>Balances</i>
EvoTest (Full Structured Evolution)	0.94	0.14	0.32
EvoPrompt	0.65	0.10	0.24
EvoTest (w/ Simple Mutation)	0.65	0.07	0.20
EvoTest (w/o Prompt Evolution)	0.52	0.05	0.16

Table 6: Comparison of AUC scores in a setting where the Actor LLM is fixed to qwen/qwen3-32b for all methods.

Method	Actor LLM	Evolver LLM	Detective	Zork1	Balances	Avg.
SFT (online)	qwen3-32b	N/A	0.40	0.11	0.22	0.24
GRPO (online)	qwen3-32b	N/A	0.55	0.07	0.30	0.31
EvoTest (Ours)	qwen3-32b	qwen3-32b	0.68	0.10	0.28	0.35
EvoTest (Ours)	qwen3-32b	openai/o3	0.78	0.12	0.31	0.40

expensive hardware, making it a non-starter for a system that needs to learn at test time. In contrast, EvoTest and other gradient-free methods operate on a different timescale. Instead of a costly training run, our learning step is a single API call to an LLM, which takes about 20-30 seconds. **We provide a more formal analysis of the computational complexity of EvoTest in Appendix E**

Impact of the LLM of Evolver Agent. To assess the sensitivity of our framework to the Evolver’s reasoning capabilities, we conducted an ablation study on its underlying LLM (Table 4). The results reveal a clear correlation between model scale and agent performance; more powerful models like openai/o3 consistently yield higher scores, likely due to their superior ability to distill complex strategic insights from raw episode transcripts. Notably, even with a significantly smaller model such as qwen3-8b, performance remains substantially above the non-learning *Static* baseline. This finding demonstrates the robustness of the EvoTest framework: while a more capable Evolver LLM acts as a performance amplifier, the fundamental act-evolve loop is effective in its own right.

Ablation on the Structure of Prompt Evolution. To further isolate the contribution of our prompt-evolving logic, we create a baseline, “EvoTest (w/ Simple Mutation),” which replaces our detailed Evolver prompt with a generic instruction to simply “analyze the trajectory and generate an improved prompt.” The results in Table 5 show that while this simpler mutation improves over no prompt evolution, it significantly underperforms our system, with the AUC score on Detective dropping from 0.94 to 0.65. This demonstrates that the performance gains are not just from evolving the prompt, but from the Evolver’s sophisticated analysis, which is a core contribution of our framework.

A Fairer Comparison with Fine-Tuning Methods. In this setup, we normalize the underlying model capabilities by using `qwen/qwen3-32b` as the backbone for all compared methods. As shown in Table 6, we evaluate two versions of our method: one using `qwen/qwen3-32b` for both the Actor and Evolver, and a second version that pairs the `qwen/qwen3-32b` Actor with the stronger `openai/o3` Evolver. The results confirm that even when both components use `qwen/qwen3-32b`, `EvoTest` (Avg. AUC 0.35) outperforms the strongest weight-update baseline, `GRPO` (0.31). Furthermore, the performance leap when using a more capable Evolver (Avg. AUC 0.40) underscores the impact of the optimizer’s reasoning ability within our framework.

6 CONCLUSION

We introduce the J-TTL benchmark to measure test time agent learning and proposed EvoTest, a novel evolutionary framework that improves agentic systems at test-time without gradients. By analyzing entire episode transcripts, EvoTest evolves the complete agent configuration—policy, memory, and hyperparameters—to rapidly adapt. Our experiments show EvoTest significantly outperforms strong baselines, including reflection, prompt optimization, and online fine-tuning. Its strength lies in using rich, narrative feedback for credit assignment, a far more data-efficient paradigm than relying on sparse rewards. This work provides a concrete step toward building truly autonomous agents that learn and self-improve from experience.

540 ETHICS STATEMENT
541542 All authors of this paper have read and adhered to the ICLR Code of Ethics. Our work focuses
543 on foundational research into the learning capabilities of AI agents within simulated, text-based
544 environments. We have identified and considered the following potential ethical dimensions of this
545 research:
546547 **Inherited Bias in Language Models** Our framework, along with the baselines, utilizes large lan-
548 guage models (LLMs) as backbones. It is well-documented that LLMs can inherit and amplify
549 societal biases present in their training data. While the fictional context of the Jericho games is
550 unlikely to surface common social biases, we acknowledge that the agents' generated language and
551 decisions are fundamentally shaped by the underlying models. Our research does not introduce new
552 sources of bias but operates within the existing limitations of current LLM technology.
553554 **Dual-Use and Long-Term Implications** Research into self-improving autonomous agents con-
555 tributes to a long-term vision of more capable and independent AI. Such technology could, in the
556 distant future, have dual-use potential. However, our work is situated at a very early, foundational
557 stage and is confined to a controlled, non-physical, and non-real-world gaming environment. The
558 primary goal is to understand and measure test-time learning in a sandboxed setting, which is a
559 critical step for developing safer and more reliable AI systems.
560561 **Research Integrity** This research does not involve human subjects, personal data, or any form of
562 deception. The datasets (Jericho games) and software are publicly available. We have been trans-
563 parent about the use of LLMs for language polishing in the manuscript, as detailed in Appendix B.
564 We have no conflicts of interest that could have influenced the results or their interpretation.
565566 REPRODUCIBILITY STATEMENT
567568 We are committed to the full reproducibility of our work. To this end, we have made extensive
569 efforts to document our methodology and provide all necessary artifacts. The primary resources for
570 reproducing our results are detailed in the appendices.
571572 Our complete source code, which includes the implementation of the EvoTest framework, the
573 J-TTL benchmark setup, and all baseline methods described in the paper, is provided in the
574 supplementary materials (anonymized at <https://anonymous.4open.science/status/evotest-38C1>). **Appendix C** provides comprehensive implementation details, including the
575 specific Jericho games used, the environment configuration (e.g., episode step limits), hardware re-
576 quirements, and the exact model identifiers for all LLMs used. **Appendix D** offers detailed descrip-
577 tions of all baseline methods, enabling a faithful re-implementation of our comparisons. The core of
578 our method's learning mechanism, the Evolver Agent's master prompt, is provided in its entirety in
579 **Appendix H**). Further experimental details, including a controlled comparison on the Qwen3-32B
580 backbone, are available in **Appendix L**). All key hyperparameters and random seeds will be in-
581 cluded in our public code release to ensure deterministic replication. We believe these resources
582 provide a clear and complete pathway to reproduce our experiments and validate our findings.
583584 REFERENCES
585586 Jinyuan Fang, Yanwen Peng, Xi Zhang, Yingxu Wang, Xinhao Yi, Guibin Zhang, Yi Xu, Bin Wu,
587 Siwei Liu, Zihao Li, et al. A comprehensive survey of self-evolving ai agents: A new paradigm
588 bridging foundation models and lifelong agentic systems. *arXiv preprint arXiv:2508.07407*, 2025.
589590 Chrisantha Fernando, Dylan Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel.
591 Promptbreeder: Self-referential self-improvement via prompt evolution. *arXiv preprint
592 arXiv:2309.16797*, 2023.
593594 Stan Franklin and Art Graesser. Is it an agent, or just a program?: A taxonomy for autonomous
595 agents. In *International workshop on agent theories, architectures, and languages*, pp. 21–35.
596 Springer, 1996.
597

594 Huan-ang Gao, Jiayi Geng, Wenyue Hua, Mengkang Hu, Xinzhe Juan, Hongzhang Liu, Shilong
 595 Liu, Jiahao Qiu, Xuan Qi, Yiran Wu, et al. A survey of self-evolving agents: On path to artificial
 596 super intelligence. *arXiv preprint arXiv:2507.21046*, 2025.

597

598 Caglar Gulcehre, Ziyu Wang, Alexander Novikov, Thomas Paine, Sergio Gómez, Konrad Zolna,
 599 Rishabh Agarwal, Josh S Merel, Daniel J Mankowitz, Cosmin Paduraru, et al. RI unplugged: A
 600 suite of benchmarks for offline reinforcement learning. *Advances in neural information processing*
 601 *systems*, 33:7248–7259, 2020.

602 Qingyan Guo, Rui Wang, Junliang Guo, Bei Li, Kaitao Song, Xu Tan, Guoqing Liu, Jiang Bian, and
 603 Yujiu Yang. Evoprompt: Connecting llms with evolutionary algorithms yields powerful prompt
 604 optimizers. *ICLR*, 2024.

605

606 Matthew Hausknecht, Prithviraj Ammanabrolu, Marc-Alexandre Côté, and Xingdi Yuan. Interactive
 607 fiction games: A colossal adventure. In *Proceedings of the AAAI Conference on Artificial*
 608 *Intelligence*, volume 34, pp. 7903–7910, 2020.

609 Yufei He, Ruoyu Li, Alex Chen, Yue Liu, Yulin Chen, Yuan Sui, Cheng Chen, Yi Zhu, Luca Luo,
 610 Frank Yang, et al. Enabling self-improving agents to learn at test time with human-in-the-loop
 611 guidance. *arXiv preprint arXiv:2507.17131*, 2025.

612

613 Zhiyuan Hu, Chumin Liu, Xidong Feng, Yilun Zhao, See-Kiong Ng, Anh Tuan Luu, Junxian He,
 614 Pang Wei Koh, and Bryan Hooi. Uncertainty of thoughts: Uncertainty-aware planning enhances
 615 information seeking in large language models. *arXiv preprint arXiv:2402.03271*, 2024.

616 Zhiyuan Hu, Yibo Wang, Hanze Dong, Yuhui Xu, Amrita Saha, Caiming Xiong, Bryan Hooi, and
 617 Junnan Li. Beyond'aha!': Toward systematic meta-abilities alignment in large reasoning models.
 618 *arXiv preprint arXiv:2505.10554*, 2025.

619

620 Xu Huang, Weiwen Liu, Xiaolong Chen, Xingmei Wang, Hao Wang, Defu Lian, Yasheng Wang,
 621 Ruiming Tang, and Enhong Chen. Understanding the planning of llm agents: A survey. *arXiv*
 622 *preprint arXiv:2402.02716*, 2024.

623

624 Zhengyang Li, Qijin Ji, Xinghong Ling, and Quan Liu. A comprehensive review of multi-agent
 625 reinforcement learning in video games. *IEEE Transactions on Games*, 2025.

626

627 Pattie Maes. Modeling adaptive autonomous agents. *Artificial life*, 1(1-2):135–162, 1993.

628

629 Grégoire Mialon, Clémentine Fourrier, Thomas Wolf, Yann LeCun, and Thomas Scialom. Gaia:
 630 a benchmark for general ai assistants. In *The Twelfth International Conference on Learning*
 631 *Representations*, 2023.

632

633 Alexander Novikov, Ngan Vn, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt
 634 Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian,
 635 et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint*
 636 *arXiv:2506.13131*, 2025.

637

638 Charles Packer, Vivian Fang, Shishir_G Patil, Kevin Lin, Sarah Wooders, and Joseph_E Gonzalez.
 639 Memgpt: Towards llms as operating systems. 2023.

640

641 Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang,
 642 Mingchuan Zhang, YK Li, Yang Wu, et al. Deepseekmath: Pushing the limits of mathematical
 643 reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

644

645 Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion:
 646 Language agents with verbal reinforcement learning. *Advances in Neural Information Processing*
 647 *Systems*, 36:8634–8652, 2023.

648

649 Yashar Talebirad and Amirhossein Nadiri. Multi-agent collaboration: Harnessing the power of
 650 intelligent llm agents. *arXiv preprint arXiv:2306.03314*, 2023.

651

652 Joannes Vermorel and Mehryar Mohri. Multi-armed bandit algorithms and empirical evaluation. In
 653 *European conference on machine learning*, pp. 437–448. Springer, 2005.

648 Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai
 649 Tang, Xu Chen, Yankai Lin, et al. A survey on large language model based autonomous agents.
 650 *Frontiers of Computer Science*, 18(6):186345, 2024.

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

654 Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun
 655 Chen. Large language models as optimizers. In *The Twelfth International Conference on Learning
 656 Representations*, 2023.

657 Siyu Yuan, Kaitao Song, Jiangjie Chen, Xu Tan, Dongsheng Li, and Deqing Yang. Evo-
 658 agent: Towards automatic multi-agent generation via evolutionary algorithms. *arXiv preprint
 659 arXiv:2406.14228*, 2024.

660 Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and
 661 James Zou. Textgrad: Automatic” differentiation” via text. *arXiv preprint arXiv:2406.07496*,
 662 2024.

663 Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. Memorybank: Enhancing large
 664 language models with long-term memory. In *Proceedings of the AAAI Conference on Artificial
 665 Intelligence*, volume 38, pp. 19724–19731, 2024.

666 Han Zhou, Xingchen Wan, Ruoxi Sun, Hamid Palangi, Shariq Iqbal, Ivan Vulić, Anna Korhonen,
 667 and Sercan Ö Arik. Multi-agent design: Optimizing agents with better prompts and topologies.
 668 *arXiv preprint arXiv:2502.02533*, 2025.

669 Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng,
 670 Tianyue Ou, Yonatan Bisk, Daniel Fried, et al. Webarena: A realistic web environment for build-
 671 ing autonomous agents. *arXiv preprint arXiv:2307.13854*, 2023.

672 Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and
 673 Jimmy Ba. Large language models are human-level prompt engineers. In *The eleventh interna-
 674 tional conference on learning representations*, 2022.

675 CONTENTS

682 1	Introduction	1
683		
684 2	Related Work	2
685		
686 3	The Jericho Test-Time Learning (J-TTL) Benchmark	3
687		
688 4	The EvoTest Framework	4
689		
690 4.1	The Agentic Configuration	5
691 4.2	The Two-Agent Learning Loop	5
692 4.3	Configuration Selection via UCB	6
693		
694 5	Experiments	6
695		
696 5.1	Setup	7
697 5.2	Results	8
698 5.3	Model Analysis	9
699		
700 6	Conclusion	10
701		

702	Appendix	14
703		
704		
705	A Limitations	15
706		
707	B Use of Large Language Models for Language Polishing	15
708		
709	C Implementation Details	16
710		
711	C.1 Datasets and Environment	16
712	C.2 Hardware	16
713	C.3 Reproducibility	16
714		
715		
716	D Baselines.	17
717		
718	E Computational Complexity Analysis	19
719		
720	F Detailed Multi-Seed Experimental Results	20
721		
722		
723	G The Test-Time Learning Problem: A Detailed Analysis	20
724		
725	G.1 The Task: Learning and Mastery in Text-Adventure Games	20
726	G.2 A Concrete Failure Case: The Navigation Loop	21
727	G.3 The EvoTest Solution	22
728		
729		
730	H The Evolver Agent’s Master Prompt	22
731		
732	I Case Study: Learning to Navigate the Library	23
733		
734		
735	J The Memory Component in Practice: Concrete Examples	24
736		
737	J.1 Success Memory: Building a Database of What Works	24
738	J.2 Failure Memory: Identifying and Pruning Unproductive Actions	29
739		
740	K Evolution of Agent Behavior: Key Transcript Snapshots from <i>Detective</i>	30
741		
742	K.1 Episode 3: Early Exploration and Basic Mistakes	30
743	K.1.1 Key Transcript Steps: Episode 3	30
744	K.2 Episode 22: Mid-Stage Execution of a Learned Plan	31
745	K.2.1 Key Transcript Steps: Episode 22	32
746	K.3 Episode 49: Near-Perfect Execution and Refinement	32
747	K.3.1 Key Transcript Steps: Episode 49	33
748		
749		
750		
751	L Experiment with Qwen3-32B Backbone	34
752		
753	M Dynamic Hyperparameter Evolution: A Case Study	35
754		
755	N Scalability with More Capable Models	35

756 **A LIMITATIONS**
757758 Our work introduces a novel evolutionary framework for test-time learning, but it is important to
759 acknowledge the inherent trade-offs and limitations of this approach.
760761 **The Drawback of Not Modifying Neural Weights.** Our framework’s foundation is its gradient-
762 free nature, which is key to its practicality and data efficiency at test time. This strategic decision,
763 however, introduces fundamental trade-offs by constraining learning to the symbolic level, rather
764 than the parametric level of the model’s weights. 1) **Bounded by Pre-trained Capabilities:** Our
765 approach, like other LLM-based agents, assumes the backbone model possesses latent knowledge
766 sufficient for the task. The central challenge we address is not the knowledge itself, but the devel-
767 opment of an efficient test-time learning mechanism to surface and apply this knowledge on the fly.
768 **EvoTest** operates as a high-level orchestrator; it learns to discover and refine the existing capabilities
769 of its backbone LLM. However, it cannot instill fundamentally new, low-level reasoning patterns
770 that lie outside the model’s pre-trained knowledge base. For instance, if a game required a form
771 of complex spatial reasoning that the LLM has never mastered, no amount of prompt evolution or
772 memory management could create this capability from scratch. Fine-tuning, in contrast, holds the
773 long-term promise of teaching a model truly novel skills. The performance of **EvoTest** is there-
774 fore capped by the inherent intelligence of its frozen backbone. Future work could explore hybrid
775 approaches that use **EvoTest** for rapid, episode-to-episode adaptation while employing very slow,
776 background fine-tuning to gradually enhance the agent’s core competencies over hundreds of ses-
777 sions. 2) **Task-Specific Overfitting vs. Generalizable Skill Acquisition:** The J-TTL benchmark and
778 our method are designed for rapid improvement on a single task. The highly specific strategies and
779 “verbal guardrails” that **EvoTest** learns (e.g., *unlock door with key*) are a form of strategic
780 overfitting to the current task instance. This is highly effective for the TTL setting, but the learned
781 knowledge is brittle and may not generalize to even minor variations in the task environment (e.g.,
782 if the key were in a different room). In contrast, traditional RL and fine-tuning, when applied across
783 a diverse distribution of tasks, aim to learn more robust and generalizable policies. A key direction
784 for future research is to investigate how the symbolic strategies evolved by **EvoTest** for specific tasks
785 could be collected and abstracted to build a library of general-purpose skills.
786787 **Dependency on a Powerful Evolver Agent.** Our framework decouples acting from adaptation, but
788 in doing so, it creates a strong dependency on the reasoning capabilities of the Evolver LLM. As our
789 ablation in Table 4 demonstrates, the quality of the evolutionary step is correlated with the power
790 of the model performing the analysis. This means that the success of the entire system hinges on
791 access to a capable-and potentially-expensive-“optimizer” model.
792793 **Complexity of the Evolutionary Search Space.** The agentic configuration $\chi = (p, M, h, u)$ forms
794 a vast and complex combinatorial search space. Our current framework employs a simple $(1 +$
795 $m)$ evolutionary strategy with UCB selection, which, while effective, is a relatively simple search
796 heuristic. It may be prone to converging on local optima, especially in more complex games. Future
797 work could explore more sophisticated population-based evolutionary algorithms, quality-diversity
798 methods, or techniques from automated program synthesis to more effectively navigate this complex
799 strategic landscape.
800797 **B USE OF LARGE LANGUAGE MODELS FOR LANGUAGE POLISHING**
798800 In the preparation of this manuscript, we used Large Language Models (LLMs) as a writing assis-
801 tance tool to enhance language, clarity, and readability. This usage was strictly confined to polishing
802 text that was already drafted by the human authors.
803804 Our process was interactive. After writing the core content, we used LLMs with specific prompts
805 to refine the text. These prompts included requests to “check for grammatical errors,” “rephrase
806 this sentence for clarity,” “make this paragraph more concise,” or “suggest alternative phrasing to
807 improve flow.”
808809 All suggestions generated by the LLM were critically reviewed, and the human authors retained
810 full editorial control, making all final decisions regarding the manuscript’s content and wording.
811 The LLMs were not used to generate any scientific ideas, experimental results, data analysis, or
812 other core intellectual contributions of the paper. The role of the LLM was analogous to that of an
813

810 advanced grammar and style checker, and all research and conclusions presented are entirely the
 811 work of the authors.
 812

814 C IMPLEMENTATION DETAILS

816 C.1 DATASETS AND ENVIRONMENT

- 818 • **Datasets:** Our evaluation is conducted on six publicly available Interactive Fiction (IF) games
 819 from the Jericho suite (Hausknecht et al., 2020). These games were chosen to represent a di-
 820 verse range of puzzle structures and difficulties: Detective, Library, Zork1, Zork3,
 821 Balances, and Temple.
- 822 • **Environment:** All experiments are run using Jericho’s standard Python API, FrotzEnv. For
 823 each game, a test-time learning session consists of $K = 50$ consecutive episodes. To ensure
 824 fair comparison, each episode is capped at a maximum of $T = 110$ interaction steps. The game
 825 environment is reset to its identical initial state after each episode, meaning any performance gains
 826 are solely attributable to the agent’s learning algorithm. All textual observations from the game
 827 are converted to lowercase before being processed by the agent. All methods, including our own
 828 and all baselines, operate under these identical environmental constraints.

829 C.2 HARDWARE

831 There is a significant difference in hardware requirements between the gradient-free and gradient-
 832 based methods evaluated.
 833

- 834 • **Gradient-Free Methods:** EvoTest and all non-weight-update baselines (e.g., Static, Memory,
 835 RAG, Reflexion, TextGrad, EvoPrompt) do not require specialized local hardware. Their learning
 836 steps are executed via API calls to external LLMs. As such, these methods can be run on a standard
 837 machine with a CPU and a stable internet connection.
- 838 • **Gradient-Based Methods:** The weight-update methods, SFT (online) and GRPO (online), have
 839 substantial hardware demands. The online fine-tuning and policy gradient updates were performed
 840 on a dedicated cluster equipped with **4 NVIDIA H100 GPUs**. This hardware is necessary to
 841 accommodate the model’s weights, gradients, and optimizer states in memory and to complete the
 842 training step in a reasonable time frame.

843 C.3 REPRODUCIBILITY

845 We are committed to ensuring that our results are fully reproducible. To this end, we will make our
 846 code and experimental artifacts publicly available.
 847

- 848 • **Code Release:** The complete source code for the EvoTest framework, the J-TTL benchmark setup,
 849 and all baseline implementations will be released at a public GitHub repository upon publication.
 850 The current code is available for review at <https://anonymous.4open.science/status/evotest-38C1>.
- 851 • **Model and Environment Identifiers:** We have specified the exact model identifiers used for all
 852 roles and baselines (e.g., google/gemini-2.5-flash, openai/o3-2025-04-16). We
 853 will also provide the version of the Jericho library and specific game files used in our experiments
 854 to prevent discrepancies arising from environment updates.
- 855 • **Configuration and Prompts:** The initial configuration files, including the generic starting
 856 prompts for the Actor Agent and the detailed master prompts used to guide the Evolver Agent,
 857 will be included in the code release. This is essential for reproducing the evolutionary trajectory
 858 of the agents.
- 859 • **Full Experimental Logs:** To facilitate detailed analysis and verification, we will release the com-
 860 plete logs from all our experimental runs. These logs will include the per-episode transcripts, the
 861 sequence of evolved configurations (prompts, hyperparameters, etc.), UCB selection scores, and
 862 final episode scores for every method on every game.
- 863 • **Seeds and Hyperparameters:** All random seeds used for LLM sampling and environment ini-
 864 tialization will be provided. Additionally, all fixed hyperparameters, such as the UCB exploration

864 constant β and the number of children m generated per evolution step, will be documented in the
 865 repository.
 866

Algorithm 1 EvoTest: Evolutionary Test-Time Learning

```

869 1: Input: Number of episodes  $K$ , initial configuration  $\chi^{(1)}$ 
870 2: Initialize history of returns  $\mathcal{H} \leftarrow \emptyset$ 
871 3: for  $e = 1, \dots, K$  do
872 4: ▷ === Acting Phase ===
873 5: Select configuration  $\chi^{(e)}$  for the current episode.
874 6: Actor Agent: Execute episode using  $\chi^{(e)}$  to get trajectory  $\tau^{(e)}$  and return  $R(e)$ .
875 7: Update statistics for  $\chi^{(e)}$ :  $n(\chi^{(e)}) \leftarrow n(\chi^{(e)}) + 1$ , update  $\hat{\mu}(\chi^{(e)})$ .
876 8: ▷ === Evolution Phase ===
877 9: Evolver Agent:
878 10: Update Memory:  $M^{(e+1)} \leftarrow \text{UpdateMemory}(M^{(e)}, \tau^{(e)})$ .
879 11: Generate child configurations  $C^{(e+1)} = \{\tilde{\chi}_1, \dots, \tilde{\chi}_m\}$  by applying evolutionary operators
880 (prompt mutation, etc.) to  $\chi^{(e)}$  and incorporating  $M^{(e+1)}$ .
881 12: ▷ === Selection Phase ===
882 13: Select next configuration  $\chi^{(e+1)}$  from  $\{\chi^{(e)}\} \cup C^{(e+1)}$  using the UCB rule (Eq. 5).
883 14: end for
884 15: Return: Sequence of episode returns  $\{R(1), \dots, R(K)\}$ .
  
```

885

886 **D BASELINES.**

887

888 To rigorously evaluate the performance of EvoTest on the J-TTL benchmark, we compare it against a
 889 comprehensive suite of baseline methods. These baselines are organized into four distinct categories
 890 based on their underlying learning strategy: memory-based methods, reflection-based methods,
 891 automated prompt optimization methods, and weight-update methods. We also include a non-learning
 892 static agent to establish a zero-shot performance floor. For all comparisons, methods are allocated
 893 the same step budget per episode and, where applicable, utilize the same backbone LLMs to ensure
 894 a fair and controlled evaluation environment.

895 **Non-Learning Baseline.**

896

897 • **Static:** This agent serves as the fundamental, non-learning baseline. It operates with a single,
 898 fixed configuration—including a generic, hand-crafted prompt (e.g., “Explore the environment
 899 and try to score points.”) and default hyperparameters—for the entire duration of the test session.
 900 It performs no updates between episodes and has no mechanism for cross-episode memory. Its
 901 purpose is to measure the zero-shot performance of the backbone LLM on the task and establish a
 902 reference point against which all learning-based improvements can be quantified. Any variation in
 903 its score across episodes is attributable solely to the inherent stochasticity of the LLM’s generation
 904 process.

905 **Memory-based Methods.** These methods learn by accumulating and accessing information from
 906 past interactions. However, they do not modify the agent’s core strategic prompt or its decision-
 907 making logic.

909 • **Memory (Full History):** This method attempts to learn by providing the agent with maximum
 910 historical context. After each episode, the complete transcript of that episode is appended to a
 911 growing history of all previous transcripts in the session. This full session history is then placed
 912 into the LLM’s context window for the subsequent episode. The primary learning mechanism
 913 is in-context learning, where the LLM is expected to identify patterns and successful strategies
 914 from the raw text of past attempts. Its main limitation is the finite context window of the LLM;
 915 when the session history exceeds the context limit, the oldest parts of the history are automatically
 916 truncated.

917 • **RAG (Retrieval-Augmented Generation):** This agent enhances its decision-making by actively
 918 retrieving relevant information from past experiences. All trajectories from the current session are

918 stored in a vector database. At each step within an episode, the agent’s current observation is used
 919 to query this database to find the most similar or relevant snippets from past trajectories. These
 920 retrieved snippets, which may contain successful or failed state-action sequences from similar
 921 situations, are then injected into the prompt as additional context. This allows the agent to dynam-
 922 ically access pertinent past knowledge without being constrained by a fixed context window. The
 923 base prompt and hyperparameters, however, remain static throughout the session.
 924

925 **Reflection-based Methods.** This category includes methods that use an LLM to generate high-level
 926 textual analyses of past performance, which are then used to guide future behavior.
 927

- 928 • **Summary:** This agent learns by creating a condensed narrative of its experiences. After each
 929 episode, an LLM is prompted to progressively summarize the entire history of all past transcripts.
 930 This summary is updated after every episode to incorporate the latest attempt, creating a concise,
 931 high-level overview of the session’s progress, including key discoveries and persistent challenges.
 932 This condensed summary is then prepended to the agent’s prompt for the next episode, aiming to
 933 provide strategic context without consuming the entire context window with raw transcripts.
 934
- 935 • **Reflexion (Shinn et al., 2023):** A prominent “verbal reinforcement learning” baseline. After each
 936 episode concludes, the agent reflects on its performance by analyzing the trajectory transcript. It
 937 generates a structured self-reflection that identifies specific failures, hypothesizes their root causes,
 938 and formulates an explicit, actionable plan to avoid those mistakes in the future (e.g., “I got stuck
 939 in the kitchen because I kept trying to ‘open the locked pantry’”. In the next attempt, I must
 940 first ‘find the pantry key’ in the living room.”). This textual reflection is then added to the agent’s
 941 prompt, accumulating over episodes to build a rich, strategy-focused memory that directly informs
 942 future decision-making.
 943

944 **Automated Prompt Optimization Methods.** These methods focus on iteratively refining the
 945 agent’s core policy by directly modifying its guiding system prompt.
 946

- 947 • **TextGrad (Yuksekgonul et al., 2024):** This method is adapted for our test-time learning setting
 948 by treating the prompt as a set of “textual parameters” to be optimized. After each episode,
 949 the trajectory and the prompt that generated it are passed to a separate “optimizer” LLM. This
 950 optimizer generates a “textual gradient”—a short, critical analysis describing a flaw in the prompt
 951 and suggesting a direction for improvement. A subsequent LLM call then “applies” this gradient
 952 by editing the original prompt based on the critique. This creates a refined prompt for the next
 953 episode, directly evolving the agent’s high-level strategy.
 954
- 955 • **Promptbreeder (Fernando et al., 2023) & EvoPrompt (Guo et al., 2024):** These two methods are
 956 adapted from their original formulations to our sequential, single-session setting. Both employ an
 957 evolutionary algorithm to optimize a population of prompts. The process begins with a set of initial
 958 seed prompts. For each episode, a prompt is selected from the population, and its performance
 959 is evaluated based on the final episode score, which serves as its “fitness.” After the episode, this
 960 fitness score is used to guide evolutionary operations. High-performing prompts are selected for
 961 “mutation” (where an LLM makes small modifications to the prompt) and “crossover” (where an
 962 LLM combines two successful prompts). The prompt for the next episode is then selected from
 963 this newly evolved population. This creates a competitive, population-based search for the most
 964 effective guiding instruction.
 965

966 **Weight-Update Methods (Online Fine-Tuning).** In contrast to the gradient-free methods above,
 967 this category includes baselines that directly modify the weights of the backbone LLM via online
 968 fine-tuning. These methods represent the traditional approach to model adaptation.
 969

- 970 • **SFT (online):** This agent learns by imitating its own past behavior. After each episode, the
 971 trajectory is converted into a dataset of (state, action) pairs. The backbone LLM is then fine-tuned
 972 on this dataset using a standard Supervised Fine-Tuning (SFT) objective. This update adjusts the
 973 model’s weights to increase the likelihood of generating the actions it took in the previous episode,
 974 given the same states. This approach reinforces the entire trajectory, which can be effective for
 975 successful runs but risks strengthening poor decision-making patterns from failed attempts.
 976

972 • **GRPO (online):** (Shao et al., 2024) This agent uses a gradient-based Reinforcement Learning
 973 (RL) approach to update its policy. After each episode, the trajectory’s state-action pairs and their
 974 associated rewards are used to compute a policy gradient. The model’s weights are then updated
 975 to “reinforce” actions that led to positive rewards and suppress those that did not. This allows
 976 for more nuanced, reward-guided credit assignment than SFT. However, its effectiveness is highly
 977 dependent on the quality and density of the reward signal, and it is computationally intensive,
 978 requiring significant GPU resources for backpropagation through the model.

980 **E COMPUTATIONAL COMPLEXITY ANALYSIS**
 981

982 In this section, we provide a more formal analysis of the computational complexity of EvoTest
 983 compared to the baselines, focusing on the cost per test-time learning cycle (one episode of acting
 984 and one phase of learning). We define the following variables for our analysis:

985 • K : The total number of episodes in a session.
 986 • T : The maximum number of steps per episode.
 987 • C_t : The context length (in tokens) provided to the Actor LLM at step t .
 988 • L_a : The average length (in tokens) of a generated action.
 989 • L_o : The average length (in tokens) of an environment observation.
 990 • $\tau_L = T \cdot (L_o + L_a)$: The approximate total length of an episode transcript.
 991 • m : The number of child configurations generated by the Evolver in EvoTest.
 992 • P_{actor} : The number of parameters in the actor LLM.
 993 • P_{evolver} : The number of parameters in the evolver/optimizer LLM.
 994 • d : The hidden dimension of the actor LLM’s transformer architecture.

995 We model the cost of an LLM forward pass for generating L_{out} tokens from an input of L_{in} tokens
 996 as $\text{Cost}_{\text{LLM}}(L_{\text{in}}, L_{\text{out}})$. This cost is primarily dependent on the model size and the total number of
 997 tokens processed.

1000 **Complexity of EvoTest.** The cost of a single EvoTest cycle can be decomposed into the Acting
 1001 Phase and the Evolution Phase.

1003 **1. Acting Phase:** In each episode, the Actor Agent takes T steps. At each step t , it queries the
 1004 backbone LLM.

$$1005 \text{Cost}_{\text{Act}} = \sum_{t=1}^T \text{Cost}_{\text{LLM}}(C_t, L_a) \approx T \cdot \text{Cost}_{\text{LLM}}(\bar{C}, L_a) \quad (6)$$

1007 where \bar{C} is the average context length. This cost is dominated by T forward passes through the actor
 1008 model.

1009 **2. Evolution Phase:** After the episode, the Evolver Agent performs a single, large query to generate
 1010 new configurations. The input is the full episode transcript (τ_L).

$$1012 \text{Cost}_{\text{Evolve}} = \text{Cost}_{\text{LLM}}(\tau_L, L_{\text{config}}) \quad (7)$$

1013 where L_{config} is the length of the generated configuration text. The UCB update step is $\mathcal{O}(m)$, which
 1014 is negligible compared to the LLM call.

1015 The total cost for one cycle of EvoTest is thus:

$$1017 \text{Cost}_{\text{EvoTest}} = T \cdot \text{Cost}_{\text{LLM}}(\bar{C}, L_a) + \text{Cost}_{\text{LLM}}(\tau_L, L_{\text{config}}) \quad (8)$$

1018 This cost is entirely composed of LLM forward passes, which can be served via APIs without
 1019 requiring local GPU memory for gradients.

1020 **Complexity of Baselines.**
 1021

1022 • **Static/Memory:** The cost is simply the acting phase, Cost_{Act} . These are the most efficient but
 1023 least effective methods.
 1024 • **Reflexion/EvoPrompt:** These methods have a similar complexity profile to EvoTest. Their learn-
 1025 ing phase also consists of a single large LLM call that takes the transcript τ_L as input to generate
 a reflection or a new prompt. Their total cost is structurally identical to Equation 3.

1026 • **Online RL (GRPO):** This is where the complexity profile differs fundamentally. The cycle con-
 1027 sists of an acting phase and a weight-update phase.
 1028

1029 **1. Acting Phase (RL):** The cost is identical to other methods: $\text{Cost}_{\text{Act}} = T \cdot \text{Cost}_{\text{LLM}}(\bar{C}, L_a)$.
 1030

1031 **2. Weight-Update Phase (RL):** This phase involves backpropagation to update the model weights.
 1032 The computational cost of a training step for a transformer model is approximately proportional to
 1033 the number of parameters and the total sequence length processed. For an entire episode trajectory
 1034 of length T , this cost is:
 1035

$$\text{Cost}_{\text{Update}} \approx \mathcal{O}(P_{\text{actor}} \cdot T) \quad (9)$$

1036 This cost reflects the computation for a full forward and backward pass through the trajectory to
 1037 compute gradients. More critically, this step has substantial hardware requirements. The GPU
 1038 VRAM must be large enough to store:
 1039

- 1040 • Model Weights: $\mathcal{O}(P_{\text{actor}})$
- 1041 • Gradients: $\mathcal{O}(P_{\text{actor}})$
- 1042 • Optimizer States (e.g., Adam): $\mathcal{O}(2 \cdot P_{\text{actor}})$
- 1043 • Activations: $\mathcal{O}(T \cdot d \cdot \text{batch_size})$

1044 The memory for activations scales with the episode length T , making online fine-tuning on long
 1045 trajectories very demanding. The total cost for one cycle of online RL is:
 1046

$$\text{Cost}_{\text{RL}} = T \cdot \text{Cost}_{\text{LLM}}(\bar{C}, L_a) + \mathcal{O}(P_{\text{actor}} \cdot T) \quad (10)$$

1047 **Comparative Summary.** As shown in Table 7, EvoTest’s architecture trades the expensive,
 1048 hardware-intensive backpropagation step of online RL for an additional LLM forward pass. While
 1049 a large LLM call is not free, it is computationally cheaper than a full fine-tuning pass and, most
 1050 importantly, can be offloaded to an API. This obviates the need for specialized local hardware (high-
 1051 VRAM GPUs) and makes EvoTest a more data-efficient and practical solution for the test-time
 1052 learning paradigm.
 1053

1054 Table 7: Complexity comparison of a single learning cycle.
 1055

Aspect	EvoTest (Ours)	Online RL (GRPO)
Acting Cost	$T \cdot \text{Cost}_{\text{LLM}}$	$T \cdot \text{Cost}_{\text{LLM}}$
Learning Cost	$\text{Cost}_{\text{LLM}}(\tau_L, L_{\text{config}})$	$\mathcal{O}(P_{\text{actor}} \cdot T)$
Mechanism	Gradient-Free (Forward Pass)	Gradient-Based (Backprop.)
Hardware Req.	CPU + Network	High-VRAM GPU
Scalability Driver	API Latency	GPU Compute & Memory

1064 F DETAILED MULTI-SEED EXPERIMENTAL RESULTS

1065 This section provides the detailed multi-seed results mentioned in our response to the reviewers. The
 1066 table 8 reports the mean Area Under the Curve (AUC) and standard deviation over 5 random seeds
 1067 for all methods using the `google/gemini-2.5-flash` backbone, validating the robustness of
 1068 our findings.
 1069

1071 G THE TEST-TIME LEARNING PROBLEM: A DETAILED ANALYSIS

1072 This section provides a more detailed analysis of the core research problem addressed by our J-
 1073 TTL benchmark, illustrating why traditional learning paradigms fail and motivating the design of
 1074 EvoTest.
 1075

1077 G.1 THE TASK: LEARNING AND MASTERY IN TEXT-ADVENTURE GAMES

1078 The task given to the agent in our J-TTL benchmark is designed to be a challenging test of on-the-fly
 1079 learning and adaptation.

1080 Table 8: Detailed comparison of Area Under the Curve (AUC) scores on the J-TTL benchmark,
 1081 showing **mean \pm std over 5 seeds** for the google/gemini-2.5-flash backbone. Higher
 1082 values indicate better overall performance. The best performance in each column is highlighted in
 1083 **bold**. These results robustly confirm that **EvoTest** consistently outperforms all baselines.

Method	Detective	Library	Zork1	Zork3	Balances	Temple	Avg.
Non-learning Baseline							
Static	0.22 \pm .02	0.14 \pm .01	0.03 \pm .01	0.05 \pm .01	0.11 \pm .02	0.08 \pm .01	0.11 \pm .01
Memory-based & Reflection-based Methods							
Memory	0.56 \pm .02	0.19 \pm .02	0.04 \pm .01	0.06 \pm .02	0.14 \pm .02	0.11 \pm .01	0.13 \pm .01
RAG	0.35 \pm .03	0.25 \pm .03	0.07 \pm .02	0.10 \pm .02	0.19 \pm .03	0.16 \pm .02	0.18 \pm .02
Summary	0.46 \pm .04	0.34 \pm .03	0.11 \pm .02	0.16 \pm .03	0.26 \pm .03	0.22 \pm .03	0.26 \pm .03
Reflexion	0.59 \pm .05	0.42 \pm .04	0.09 \pm .02	0.26 \pm .03	0.31 \pm .03	0.30 \pm .04	0.33 \pm .03
Automated Prompt Optimization Methods							
TextGrad	0.62 \pm .05	0.46 \pm .04	0.12 \pm .03	0.29 \pm .04	0.17 \pm .03	0.24 \pm .03	0.32 \pm .03
Promptbreeder	0.64 \pm .04	0.48 \pm .04	0.11 \pm .02	0.30 \pm .04	0.24 \pm .03	0.31 \pm .03	0.35 \pm .03
EvoPrompt	0.64 \pm .04	0.48 \pm .03	0.13 \pm .02	0.31 \pm .03	0.25 \pm .03	0.28 \pm .04	0.35 \pm .03
Weight-Update Methods (Online Fine-Tuning)							
SFT (online)	0.41 \pm .06	0.31 \pm .05	0.11 \pm .04	0.19 \pm .04	0.23 \pm .05	0.20 \pm .04	0.24 \pm .04
GRPO (online)	0.58 \pm .05	0.39 \pm .04	0.08 \pm .03	0.23 \pm .04	0.31 \pm .04	0.25 \pm .03	0.31 \pm .03
EvoTest (Ours)	0.93 \pm .02	0.78 \pm .03	0.15 \pm .02	0.36 \pm .02	0.33 \pm .02	0.32 \pm .02	0.48 \pm .01

1100
 1101
 1102 **The Environment.** The agent interacts with a classic text-based adventure game from the Jericho
 1103 suite (e.g., *Detective*, *Zork1*). In these games, the entire world is described through text. The agent
 1104 receives textual observations (e.g., "You are in the Chief's office. A piece of
 1105 white paper is on the desk.") and must issue text commands (e.g., TAKE PAPER) to
 1106 act. These games are notoriously difficult for AI due to complex puzzles, long-range planning
 1107 dependencies, sparse rewards, and irreversible consequences.

1108 **The Goal.** The agent's goal is structured at two levels:

1109

- 1110 • **The Episodic Goal (Maximize Score):** Within a single playthrough (one "episode"), the agent's
 1111 objective is to take actions that maximize its final score, which is awarded for discovering areas,
 1112 solving puzzles, and advancing the plot.
- 1113 • **The Learning Goal (Improve Across Episodes):** The ultimate task is to play the *same game*
 1114 *repeatedly* and demonstrate learning by progressively increasing its final score from one episode
 1115 to the next. The agent must use its experience from failed or suboptimal attempts to build a better
 1116 strategy for subsequent attempts.

1117 G.2 A CONCRETE FAILURE CASE: THE NAVIGATION LOOP

1118 To illustrate why this task is hard, consider an agent attempting to play *Detective*. It might correctly
 1119 execute GO WEST, but from the next location, it gets stuck by repeatedly attempting GO WEST
 1120 again, an invalid move the game rejects with "You can't go that way." This simple failure
 1121 highlights the limitations of existing methods.

1122
 1123 **The Static Agent.** A non-learning agent with a fixed, generic prompt has no mechanism to correct
 1124 this error. It will likely repeat the same mistake in every episode, resulting in a flat, low-scoring
 1125 performance curve. It cannot adapt.

1126
 1127 **The Online SFT Agent.** A more sophisticated agent might use Supervised Fine-Tuning (SFT) on
 1128 its prior trajectory. Our SFT baseline intelligently filters for "positive" actions (those that yielded a
 1129 score increase). This approach fails for two reasons:

1130

- 1131 • In the low-scoring episode where the agent got stuck, it generated very few, if any, positive actions.
 1132 The dataset for fine-tuning is therefore either empty or extremely small. With no good data to learn
 1133 from, the agent cannot improve.

1134 • Many critical actions in these games are “neutral” and provide no immediate reward (e.g.,
 1135 UNLOCK DOOR WITH KEY). An SFT agent that only trains on score-increasing actions will
 1136 never learn these essential intermediate steps, rendering it incapable of solving complex puzzles.
 1137

1138 **The Online RL Agent.** A Reinforcement Learning (RL) agent receives a reward of ‘0’ for the
 1139 invalid GO WEST action. In a sparse-reward environment like Jericho, this signal is incredibly weak
 1140 and ambiguous. It is indistinguishable from the ‘reward=0’ received for a neutral but necessary
 1141 action. A single gradient update based on this noisy signal is insufficient to meaningfully correct
 1142 the agent’s policy for the next attempt, demonstrating a failure of credit assignment in a low-data
 1143 regime.

1144 **G.3 THE EVO TEST SOLUTION**

1145 **EvoTest is designed to overcome these failures. Its Evolver Agent analyzes the entire episode**
 1146 **transcript, not just scalar rewards or positive actions.**

1147 **It Learns from Failure:** Unlike SFT, EvoTest learns most effectively from failures. The Evolver
 1148 semantically identifies the unproductive loop by reading the game’s textual feedback (“You can’t
 1149 go that way.”) paired with the repeated action and recognizes it as a problem to be solved.

1150 **It Performs Whole-System Evolution:** Based on its analysis, the Evolver directly rewrites the
 1151 agent’s prompt, generating a targeted, structural edit to correct the error (e.g., adding a new rule:
 1152 Step 5: From the street, GO EAST to enter the Mayor’s house.). This
 1153 is a far more direct and data-efficient learning mechanism than a small, gradient-based weight up-
 1154 date.

1155 **H THE EVOLVER AGENT’S MASTER PROMPT**

1156 The core of EvoTest’s learning capability resides in the Evolver Agent, which is guided by a com-
 1157 prehensive “master prompt.” This prompt structures the analysis of a completed episode transcript,
 1158 enabling the Evolver’s LLM to perform holistic, multi-faceted updates across the entire agentic
 1159 configuration $\chi = (p, M, h, u)$. Unlike simpler approaches that only modify the policy prompt,
 1160 our master prompt instructs the Evolver to act as a full-system optimizer, proposing changes to the
 1161 agent’s high-level strategy, its structured memory, its low-level decision-making parameters, and its
 1162 internal tool-use logic.

1163 The prompt is divided into four distinct parts, each targeting a specific component of the agentic
 1164 system.

1165 **EvoTest Master Prompt: Preamble and Context**

1166 You are an AI agent system optimizer. Your task is to analyze the transcript of a
 1167 text-adventure game session and generate a new, improved configuration for the next
 1168 agent that will play the same game. The goal is to help the agent score higher in
 1169 the next episode.

1170 The agent’s configuration has four components:
 1171 1. A guiding prompt (the agent’s high-level strategy).
 1172 2. Memory updates (structured data for a success/failure database).
 1173 3. Hyperparameters (like temperature, for decision-making).
 1174 4. Tool-use routines (Python code for state abstraction and rules for memory access).

1175 You will receive the previous guiding prompt and the full game history. Generate a
 1176 new, complete configuration by following the four parts below.

1177 The LLM agent used the following guiding prompt (which may not be accurate; rewrite
 1178 it if needed):
 1179 “{cur_prompt}”

1180 Here is the history of that game session:
 1181 --- GAME HISTORY START ---
 1182 {cur_history_str}
 1183 --- GAME HISTORY END ---
 1184 {negative_section}

1188

1189

1190

1191

1192

1193

1194

1195

1196

1197

1198

1199

1200

1201

1202

1203

1204

1205

1206

1207

1208

1209

1210

1211

1212

1213

1214

1215

1216

1217

1218

1219

1220

1221

1222

1223

1224

1225

1226

1227

1228

1229

1230

1231

1232

1233

1234

1235

1236

1237

1238

1239

1240

1241

EvoTest Master Prompt: Detailed Generation Instructions

PART 1: Generate a new improved guiding prompt.
 This prompt is the agent's high-level policy. Structure it clearly.

1. Create a 'Walkthrough' or 'Essential Actions' section. Identify all useful actions from the history that led to score increases or were strictly necessary for game progression. Synthesize these into a clear, step-by-step plan. Be precise with action phrasing (e.g., 'unlock door with key' instead of 'use key').
2. Create an 'Actions to Avoid' section. Identify actions that led to getting stuck, caused loops, produced repeated errors, or were clearly unproductive. List these as negative constraints or 'guardrails.'
3. If the agent has not yet won, create a final 'Exploration Plan' section. Brainstorm possible next steps. List rooms or objects that have been seen but not fully interacted with. Suggest a systematic approach for the agent to follow once the known walkthrough is complete (e.g., 'visit every room, systematically use LOOK, EXAMINE, SEARCH, and try actions like PUSH, PULL, READ on all objects.').

PART 2: Generate memory updates for the database.
 Analyze the transcript and extract all state-action pairs that resulted in a score increase. Format this as a JSON list of objects. This data will be added to the agent's success memory.

- The 'state_text' should be the full observation text right before the action was taken.
- The 'action' is the command the agent gave.
- The 'score_delta' is the positive score change.

If no new score increases were found, return an empty list '[]'.
 Example:

```
[{"state_text": "<<< closet >>>\nyou are in a closet. ...", "action": "take pistol", "score_delta": 10}, {"state_text": "<<< living room >>>\nyou are standing ...", "action": "get wood", "score_delta": 10}]
```

PART 3: Suggest hyperparameter adjustments.
 Analyze the agent's overall behavior to tune the exploration-exploitation balance. Output a JSON object.

- If the agent was stuck in repetitive loops or seemed overly cautious, suggest increasing the temperature to encourage more diverse actions (e.g., '"temperature": 0.8').
- If the agent's actions seemed chaotic, random, or deviated from a good existing plan, suggest decreasing the temperature to promote adherence to the strategy (e.g., '"temperature": 0.2').
- If the agent's behavior was optimal and it followed the plan well, suggest no changes by returning an empty JSON object '{}'.
 Example: '{} {"temperature": 0.75}'

PART 4: Generate new tool-use routines.

4a. State Extractor Code: Generate a Python code block for a function named 'extract_state'. This function takes the full 'game_history' string as input and returns a short summary of key milestones completed.

- Base the logic on the presence of specific *strings* from the game's output*, not the agent's actions.
- Keep the code simple to avoid bugs. Use 'if/elif/else' with 'in' checks.
- Do not include comments inside the function body.

4b. Memory Interaction Logic: Propose a single sentence to be added to the main guiding prompt that refines how the agent should use its memory. This logic should evolve to become more disciplined over time.

- Early-stage suggestion: 'Hint: You can check your memory of past successes for ideas in familiar situations.'
- Mid-stage suggestion: 'Strategy: In any room you've visited before, consult your success memory for proven actions.'
- Late-stage rule: 'Rule: Before every action, you MUST query your success memory for the current state and follow its suggestion if one exists.'

Format your response as follows with NO additional text. The function name MUST be 'extract_state'.

```
[Your generated prompt here]
[Your JSON for memory updates here]
[Your JSON for hyperparameter adjustments here]
<code>
def extract_state(game_history):
    # [Return a string summarizing the current state]
</code>
[Your one-sentence memory interaction logic here]
```

I CASE STUDY: LEARNING TO NAVIGATE THE LIBRARY

Analysis of EvoTEST's behavior in the *library* game reveals it is performing a form of Verbal Reinforcement Learning. In this paradigm, the agent's natural language prompt is its policy, the full episode transcript is the rich reward signal, and policy updates are semantic edits to the prompt. The evolution of this policy is detailed in the prompts shown in Figures 4 through 8.

Episode 0: The Credit Assignment Problem. The initial agent begins with a simple, high-level directive (Figure 4) and acquires the target biography, but gets stuck at the security alarm. It mis-

1242 attributes the failure, falling into loops of re-locking the rare books room, demonstrating a classic
 1243 credit assignment problem where a delayed, sparse negative signal is linked to the wrong proximate
 1244 cause.

1245 **Episode 1-3: Positive Verbal Policy Update.** The Evolver Agent analyzes the first transcript, iden-
 1246 tifies the successful action sequence for retrieving the biography, and distills it into a new heuris-
 1247 tic. This **positive policy update** is evident in the updated prompt for Episode 1 (Figure 5) and
 1248 its further refinement by Episode 3 (Figure 6). The Evolver performed credit assignment on the
 1249 score-increasing events and codified the successful trajectory into the agent’s strategy, effectively
 1250 abstracting a reusable skill from a single experience.

1251 **Episode 11: Latent Value Discovery.** By now, the evolved prompt has discovered a non-obvious
 1252 but critical action: ASK TECHNICIAN ABOUT GATES (Figure 7). This command provides no
 1253 immediate score increase but is a prerequisite for solving a later puzzle. This demonstrates **la-**
 1254 **tent value discovery.** A traditional RL agent would struggle to find such an unrewarded action.
 1255 EVO TEST uses semantic reasoning on the narrative—"an NPC is blocking an object; making them
 1256 leave could be useful"—to identify a state with high future value, bypassing the need for numerical
 1257 value propagation.

1258 **Episode 49: Negative Policy Update and Guardrails.** The agent’s policy is now a multi-stage
 1259 plan. Crucially, it includes an “Actions to Avoid” section (Figure 8), warning against interacting
 1260 with the security gates because it “causes an endless clarification loop.” This is a **negative policy**
 1261 **update.** The system has learned not just what to do, but what not to do, by identifying a frustrating
 1262 failure pattern (a parser bug) from the transcript. It created a “verbal guardrail” to prevent this
 1263 specific failure, a form of learning that is difficult to achieve with scalar rewards alone.

1264
 1265
 1266
 1267 Prompt of Episode 0
 1268 Explore systematically and examine objects to make progress.

1270 Figure 4: Case Study: Prompt of Episode 0.

1271
 1272 J THE MEMORY COMPONENT IN PRACTICE: CONCRETE EXAMPLES

1273
 1274
 1275
 1276
 1277
 1278
 1279 To illustrate precisely how the Evolver Agent constructs and utilizes memory, this section details the
 1280 process using interactions from the *Detective* game. The memory is not a monolithic block of text;
 1281 it is a structured database, programmatically populated by the Evolver after each episode.

1282
 1283 J.1 SUCCESS MEMORY: BUILDING A DATABASE OF WHAT WORKS

1284
 1285
 1286
 1287
 1288
 1289
 1290 The Success Memory functions as a high-reward state-action lookup table. The Evolver parses the
 1291 episode transcript for any step where the score increases and logs the preceding state and action.
 1292
 1293 For example, after a single episode, the Evolver identifies several score-increasing events from the
 1294 trajectory log.

1296
 1297
 1298
 1299
 1300
 1301
 1302
 1303
 1304
 1305
 1306
 1307
 1308 **Prompt of Episode 1**
 1309 Retrieve Graham Nelson biography efficiently:
 1310 Step-by-step actions that proved essential so far
 1311 1. In lobby: talk to reference librarian first { west to
 1312 Ground-Floor Stacks, ASK LIBRARIAN ABOUT GRAHAM NELSON and then
 1313 ASK LIBRARIAN ABOUT RARE.
 1314 2. Return to lobby and GIVE ID CARD TO ATTENDANT; you receive
 1315 the Rare-Books KEY.
 1316 3. Reach second-floor stacks (north, up, south) and UNLOCK RARE
 1317 BOOKS DOOR WITH KEY, OPEN DOOR, ENTER.
 1318 4. TAKE BIOGRAPHY OF GRAHAM NELSON.
 1319 5. Leave the room, CLOSE RARE BOOKS DOOR, then LOCK RARE BOOKS
 1320 DOOR WITH KEY.
 1321 6. Back in lobby GIVE KEY TO ATTENDANT; he returns your ID
 1322 card.
 1323 Avoid
 1324 • Wandering in the dusty card-catalogue corner { it contains no
 1325 useful data.
 1326 • Trying to leave the library while still holding the key or
 1327 while the rare-books door is open/unlocked: this triggers
 1328 alarms and extra trips.
 1329 • Vague commands such as \talk to attendant"; instead use verbs
 1330 ASK or GIVE.
 1331 Next places and ideas (not yet proven)
 1332 { Northwest Duplicating Services room: may let you photocopy
 1333 pages, reducing risk at security gate.
 1334 { \Computer Room" east of 2F stacks.
 1335 { The \Private" door north of lobby.
 1336 Systematically examine new rooms, objects, signage and NPCs;
 1337 look for legitimate ways to check out or copy the biography so
 1338 the alarm will not sound when passing the security gates.
 1339
 1340
 1341
 1342
 1343
 1344
 1345
 1346
 1347
 1348
 1349

Figure 5: Case Study: Prompt of Episode 1.

1350
 1351
 1352
 1353
 1354
 1355
 1356
 1357
 1358
 1359 Prompt of Episode 3
 1360 Goal: leave the library with the Biography of Graham Nelson
 1361 legitimately checked-out.
 1362 Essential steps that advanced the story and/or scored points
 1363 (use exactly these verbs { the game is picky}):
 1364 1. From the Lobby • ask attendant about book → he sends you to
 1365 Reference
 1366 2. West to Ground-floor Stacks (Reference Desk) • ask librarian
 1367 about Graham Nelson • ask librarian about rare → she tells you
 1368 key is at Circulation
 1369 3. Return to Lobby • give id card to attendant → you receive
 1370 key to rare books room (+5)
 1371 4. Reach the heavy wooden door (N → U → S) • unlock door with
 1372 key • open door • enter • take biography (+5)
 1373 5. Before leaving the second floor • close rare books door •
 1374 lock door with key
 1375 6. Back in the Lobby • give key to attendant → attendant
 1376 returns your ID
 1377 7. Still in the Lobby • ask attendant to check out biography
 1378 (or \"check out biography\" / \"borrow biography\") • east → leave
 1379 the building without the alarm
 1380 Actions that wasted time or caused trouble { avoid these:
 1381 • Re-examining the dusty card-catalog drawers (they contain
 1382 unreadable parchment).
 1383 • \"talk attendant\" { the correct form is \\ask attendant about
 1384 ...\".
 1385 • Dropping the biography in the stacks: you must keep it and
 1386 check it out.
 1387 • Trying to exit before key is returned, door locked, and book
 1388 checked-out { triggers alarms.
 1389 Where to explore next if something is still missing:
 1390 Rooms not yet inspected in detail: • Duplicating Services (NW
 1391 from Lobby)
 1392 • Private Room (N from Lobby)
 1393 • Computer Room (E from 2nd-floor stacks)
 1394 Systematically: in every room LOOK, EXAMINE all obvious
 1395 objects, ASK any NPCs about topics that appear in room
 1396 descriptions, and INV and SCORE regularly to monitor progress.
 1397
 1398
 1399
 1400
 1401
 1402
 1403

Figure 6: Case Study: Prompt of Episode 3.

1404
1405
1406
1407
1408
1409
1410
1411
1412
1413

Prompt of Episode 11

1414 ESSENTIAL WALKTHROUGH (all steps verified by score increases)
1415 1. Gain entry to Rare-Books • ASK ATTENDANT ABOUT GRAHAM •
1416 WEST to Ground-Floor Stacks • ASK LIBRARIAN ABOUT GRAHAM • ASK
1417 LIBRARIAN ABOUT KEY • EAST to Lobby • ASK ATTENDANT ABOUT KEY •
1418 GIVE ID CARD TO ATTENDANT→ receive Rare-Books key(+5)
1419 2. Retrieve the Graham Nelson biography • WEST { NORTH { UP {
1420 SOUTH to 2F Stacks • UNLOCK DOOR WITH KEY; OPEN DOOR; ENTER •
1421 TAKE BIOGRAPHY(+5) • Return to 2F Stacks, CLOSE DOOR, LOCK DOOR
1422 WITH KEY • NORTH { DOWN { SOUTH { EAST to Lobby • GIVE KEY TO
1423 ATTENDANT(ID card returned)
1424 3. Empty the Computer Lab and grab the encyclopedia • WEST {
1425 NORTH { UP { SOUTH { EAST to Computer Lab • ASK TECHNICIAN ABOUT
1426 GATES→ technician leaves(+2) • EXAMINE PRINTERS→ automatically
1427 take Encyclopedia Frobozzica(+5)
1428 4. Please the Reference Librarian • WEST { NORTH { DOWN {
1429 SOUTH { EAST to Ground-Floor Stacks • GIVE ENCYCLOPEDIA TO
1430 LIBRARIAN(+2)
1431 Confirmed score so far: 19 points
1432 ACTIONS TO AVOID • Getting stuck in the \which do you mean, the
1433 howto manual or the security gates?" loop { when it appears,
1434 simply type HOWTO MANUAL once and continue. • Trying to open
1435 / push / pry the \Private" north door. • Fussing with the
1436 burned-out copier after the explosion. • Repeating LOOK/EXAMINE
1437 on already-searched objects.
1438 PLACES & IDEAS TO INVESTIGATE NEXT (unverified) • Read the HOWTO
1439 MANUAL offered during the gate disambiguation; then SEARCH or
1440 LOOK IN the opened security gate Tom is fixing. • Examine
1441 paintings in both stairwells and the damaged picture on the
1442 ground-floor stairs (MOVE, PUSH, PULL). • Re-visit Duplicating
1443 Services after the smoke clears for anything the blast exposed.
1444 • Scrutinise every object named in room texts (front doors,
1445 notices, posters, card-catalog drawers, etc.). • ASK each NPC
1446 ABOUT any new topic you uncover.
1447 Systematic approach: in every room LOOK, then EXAMINE and
1448 SEARCH every noun that appears. Try TAKE, OPEN, READ, PUSH,
1449 PULL, TURN and ASK <npc> ABOUT <noun/topic> to expose further
1450 clues.

Figure 7: Case Study: Prompt of Episode 11.

1451
1452
1453
1454
1455
1456
1457

1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468

1469 Prompt of Episode 49

1470 ESSENTIAL ACTION LIST { VERIFIED TO 24 POINTS
1471 1. Borrow the Rare-Books key (+5) • In the Lobby: GIVE ID CARD
TO ATTENDANT
1472 2. Confirm the book's location • WEST to Ground-Floor Stacks •
ASK LIBRARIAN ABOUT GRAHAM
1474 3. Retrieve the Graham Nelson biography (+5) • WEST { NORTH
1475 { UP { SOUTH to 2F Stacks • UNLOCK DOOR WITH KEY ; OPEN DOOR ;
1476 ENTER • TAKE BIOGRAPHY • NORTH (leave) ; CLOSE DOOR ; LOCK DOOR
1477 WITH KEY
1478 4. Return the key • NORTH { DOWN { SOUTH { EAST to Lobby • GIVE
1479 KEY TO ATTENDANT (get ID card back)
1480 5. Clear the Computer Lab (+2 + 5) • WEST { NORTH { UP { SOUTH
1481 { EAST to Computer Lab • ASK TECHNICIAN ABOUT GATES (+2) •
1482 EXAMINE PRINTERS (auto-take Encyclopedia Frobuzzica, +5)
1483 6. Donate the Encyclopedia (+2) • WEST { DOWN { SOUTH { EAST {
1484 WEST to Ground-Floor Stacks • GIVE ENCYCLOPEDIA TO LIBRARIAN
1485 7. Extra find in 2F Stacks (+5) • NORTH { UP { SOUTH to 2F
1486 Stacks • SEARCH SHELVES (get \Debt of Honor")
1487 Confirmed score so far: 24 points.
1488 Avoid: • Interacting with the security gates (causes an
1489 endless clarification loop). • Repeatedly fiddling with the
1490 paintings/canvas until new clues appear. • Non-standard verbs
1491 (USE, TYPE, TALK TO, etc.). • Trying to leave the library while
1492 holding required books.
1493 Unexplored areas worth checking once the above is complete (no
1494 points verified yet): • \Private" door in Lobby • Duplicating
1495 Services room after explosion • Stairwell paintings (\grue"
1496 canvas, damaged logo) • Card-catalog drawers and miscellaneous
1497 shelving
1498 After finishing the essentials, visit every room, LOOK, EXAMINE,
1499 SEARCH, and ASK NPCs about any new noun you encounter, noting
any score changes.
1500 Good luck!

1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511

Figure 8: Case Study: Prompt of Episode 49.

1512
1513

Evolver Agent: Parsing Successes from Episode Log

1514

The Evolver's parser identifies the following successful interactions:

1515

1. STATE: ``<< Chief's office >> ... You can see a piece of
white paper..."

1516

ACTION: ``read paper"

1517

REWARD: +10 points

1518

2. STATE: ``<< closet >> ... there is a gun on the floor..."

1519

ACTION: ``get pistol"

1520

REWARD: +10 points

1521

3. STATE: ``<< living room >> ... you see a battered piece of
wood..."

1522

ACTION: ``get wood"

1523

REWARD: +10 points

1524

1525

1526

1527

1528

Based on these observations, the Evolver programmatically updates the `success_memory.json` file. This file stores a mapping from a hash of the state's descriptive text to the action that proved successful. The resulting database entries would look like this:

1529

1530

1531

1532

1533

Success Memory Database (`success_memory.json`)

1534

State Hash (Key)	Stored Action (Value)	Score Delta
hash("<< Chief's office..."")	"read paper"	+10
hash("<< closet >>...")	"get pistol"	+10
hash("<< living room >>...")	"get wood"	+10
...

1535

1536

1537

1538

1539

1540

1541

1542

1543

1544

1545

1546

1547

1548

J.2 FAILURE MEMORY: IDENTIFYING AND PRUNING UNPRODUCTIVE ACTIONS

1549

The Failure Memory's goal is to prevent the agent from repeating obvious mistakes, especially getting stuck in loops. The Evolver identifies these patterns by detecting sequences of actions that result in no change to the game state or score.

1550

From the provided logs, the Evolver can identify a wasted move:

1551

1552

1553

1554

1555

1556

1557

Evolver Agent: Parsing a Non-Productive Action

The Evolver's analysis routine detects the following loop:

1558

1559

1560

1561

1562

1563

1564

1565

- **At Step t:** Agent is in state << hallway >> you are still in the hallway....
- **Agent's Action:** west.
- **At Step t+1:** The environment returns the *exact same state description*: << hallway >> you are still in the hallway....
- **Analysis:** The state did not change, and the score did not change. This action is flagged as unproductive for this state.

1566 Unlike the Success Memory, this insight is not stored in a database to be queried. Instead, the
 1567 Evolver uses it to directly mutate the agent’s core policy—the prompt—by adding an explicit “verbal
 1568 guardrail.” This creates a more permanent and proactive change to the agent’s strategy.
 1569

1570 **Policy Prompt Mutation: Creating a Verbal Guardrail**
 1571

1572 **Original Prompt Section (Episode N):**

1573
 1574 ## Strategy
 1575 - Explore systematically.
 1576 - Examine all objects.

1577 **Evolved Prompt Section (Episode N+1):**
 1578

1579 ## Strategy
 1580 - Explore systematically.
 1581 - Examine all objects.

1582 ## Known Dead Ends / Wasted Actions
 1583 - In ‘‘<< hallway >>’’, avoid the action ‘‘west’’. It leads back to
 1584 the same room.
 1585

1586
 1587 This example shows how EvoTest moves beyond simple trial-and-error. It performs semantic credit
 1588 assignment on the narrative of the game, identifies a specific unproductive behavior, and encodes a
 1589 rule to prevent it in the future. This form of learning, which prunes the search space by identifying
 1590 and forbidding useless actions, is a key reason for its data efficiency compared to methods that rely
 1591 solely on scalar rewards.

1592
 1593 **K EVOLUTION OF AGENT BEHAVIOR: KEY TRANSCRIPT SNAPSHOT FROM**
 1594 *Detective*
 1595

1596
 1597 To demonstrate how the agent’s behavior evolves over a session, this section presents key moments
 1598 from the transcripts of early, mid-stage, and late-stage episodes for the *Detective* game. Each
 1599 snapshot includes the agent’s guiding prompt at that stage, highlighting the direct link between the
 1600 evolved policy and the agent’s actions.
 1601

1602
 1603 **K.1 EPISODE 3: EARLY EXPLORATION AND BASIC MISTAKES**
 1604

1605 In early episodes, the agent operates with a generic prompt and relies on broad exploration. Its
 1606 behavior is characterized by discovering simple, high-reward actions while also making fundamental
 1607 errors, such as getting stuck in loops.
 1608

1609 **Guiding Prompt: Episode 3**
 1610

1611 You are a detective playing a text-based game. Your goal is to solve the murder of the Mayor
 1612 and maximize your score. Explore your surroundings, examine items, and talk to people to
 1613 gather clues. Be systematic.
 1614

1615
 1616 **K.1.1 KEY TRANSCRIPT STEPS: EPISODE 3**
 1617

1618 The agent successfully discovers initial points but then gets stuck in a simple navigational loop,
 1619 revealing a lack of long-term strategy.

```

1620
1621 Transcript Snapshot: Steps 2-9
1622
1623 [STEP 2]
1624 [OBS] confidential: detective was created by matt barringer...
1625 [ACTION] west
1626
1627 [STEP 3]
1628 [OBS] << closet >> ... there is a gun on the floor. better get it.
1629 [ACTION] take pistol
1630 [REWARD] +10
1631
1632 [STEP 4]
1633 [OBS] taken. [your score has just gone up by ten points.]
1634 [ACTION] east
1635
1636 [STEP 5]
1637 [OBS] << chief's office >> ... go north or west.
1638 [ACTION] north
1639
1640 [STEP 6]
1641 [OBS] << outside >> ... to the east is a dead end. to the west
1642 is ... street.
1643 [ACTION] west
1644 [REWARD] +10
1645
1646 [STEP 7]
1647 [OBS] << outside >> ... to the north is a restraurant ... to the
1648 east is ... home.
1649 [ACTION] west
1650
1651 [STEP 8]
1652 [OBS] you can't go that way. << outside >> ... north is a
1653 restraurant...
1654 [ACTION] west
1655
1656 [STEP 9]
1657 [OBS] you can't go that way. << outside >> ... north is a
1658 restraurant...
1659 [ACTION] look

```

1660 **Analysis:** The agent successfully finds points by taking the pistol (Step 3) and exploring west (Step 6). However, at Step 7, it fails to process the new information and tries to go ‘west’ again, an invalid move. It repeats this mistake at Step 8, demonstrating a classic failure mode of early-stage agents: getting stuck in a simple loop. This transcript provides a clear signal for the Evolver to create a rule about how to navigate this specific “outside” area.

1663 K.2 EPISODE 22: MID-STAGE EXECUTION OF A LEARNED PLAN

1664 By the middle of the session, the prompt has evolved into a partial walkthrough. The agent can now
1665 flawlessly execute complex sequences it previously struggled with but may fail on details not yet
1666 encoded in its strategy.

1667 Guiding Prompt: Episode 22

1668 **Goal: Get to the Holiday Inn. Known Path:** 1. Office: ‘read paper’, ‘west’, ‘take pistol’,
1669 ‘east’, ‘north’. 2. Mayor’s House: Go east to living room for wood, then west to dining
1670 room for note. Then go north to exit. 3. Video/Music Stores: Navigate past them to reach
1671 the alley. **Avoid:** Going west multiple times at the street hub.

1674 K.2.1 KEY TRANSCRIPT STEPS: EPISODE 22
 1675
 1676 The agent expertly navigates the Mayor's house but fails at a crucial, un-memorized interaction due
 1677 to incorrect action syntax.
 1678

1679 Transcript Snapshot: Steps 8-13 and 30-32
 1680
 1681 [STEP 8]
 1682 [OBS] << mayor's house >> ... to the east is the living room ...
 1683 [ACTION] east
 1684
 1685 [STEP 9]
 1686 [OBS] << living room >> ... you see a battered piece of wood.
 1687 [ACTION] take wooden wood
 1688 [REWARD] +10
 1689
 1690 [STEP 10]
 1691 [OBS] taken. [your score has just gone up by ten points.]
 1692 [ACTION] west
 1693
 1694 [STEP 11]
 1695 [OBS] << mayor's house >> ... to the west is the dining room.
 1696 [ACTION] west
 1697
 1698 [STEP 12]
 1699 [OBS] << dining room >> ... see a note on the table.
 1700 [ACTION] take paper note
 1701 [REWARD] +10
 1702
 1703 [STEP 13]
 1704 [OBS] taken. [your score has just gone up by ten points.]
 1705 [ACTION] east
 1706
 1707 [STEP 30]
 1708 [OBS] << back of music store >> ... to the west there is a
 1709 dazed looking man ...
 1710 [ACTION] west
 1711
 1712 [STEP 31]
 1713 [OBS] << music store >> ... you need to ... kill him. best
 1714 chance: use your gun.
 1715 [ACTION] use gun
 1716
 1717 [STEP 32]
 1718 [OBS] what do you want to use the gun on?
 1719 [ACTION] dazed man

1720 **Analysis:** The sequence from steps 8-13 shows perfect execution of a learned sub-plan, retrieving
 1721 both items in the Mayor's house without error. This demonstrates successful memory and planning.
 1722 The failure occurs later at Step 31. The prompt did not specify the exact syntax for using the gun, so
 1723 the agent defaults to a generic 'use gun' command, which leads to a clarification question from the
 1724 game and wastes a turn. The Evolver will parse this failure and refine the prompt to use the correct
 1725 syntax: 'shoot dazed man with gun'.
 1726

1727 K.3 EPISODE 49: NEAR-PERFECT EXECUTION AND REFINEMENT

1728 In late-stage episodes, the agent possesses a nearly complete walkthrough. Its behavior is precise,
 1729 confident, and focused on executing the optimal path.

1728
1729

Guiding Prompt: Episode 49 (abridged)

1730
1731
1732
1733

Follow this exact sequence to score 330 / 360 and win... ... 8. Holiday Inn – capture • ...
 • north → elevator to 15th floor (+10) • west, north, west, north, north → Room 30 • enter
 room → automatic gun-fight, arrest killer (+100) • up → “info” room → *** YOU HAVE
 WON ***

1734
17351736 K.3.1 KEY TRANSCRIPT STEPS: EPISODE 49
17371738
1739

The agent confidently executes the final, complex maze-like sequence of the game without a single misstep.

1740
1741

Transcript Snapshot: Steps 43-49

1742
1743
1744
1745
1746
1747

[STEP 43]
 [OBS] << holiday inn >> ... you have to find it. to get
 started, go north.
 [ACTION] north
 [REWARD] +10

1748
1749
1750
1751

[STEP 44]
 [OBS] << holiday inn 15th floor >> ... you can go east or west.
 [ACTION] west
 [REWARD] +10

1752
1753
1754
1755
1756

[STEP 45]
 [OBS] << hallway >> ... you can only go north.
 [ACTION] north
 [REWARD] +10

1757
1758
1759
1760

[STEP 46]
 [OBS] << hallway >> ... you can go west or east.
 [ACTION] west
 [REWARD] +10

1761
1762
1763
1764
1765
1766

[STEP 47]
 [OBS] << hallway >> ... you are getting close! you can only go
 north.
 [ACTION] north
 [REWARD] +10

1767
1768
1769
1770

[STEP 48]
 [OBS] << hallway >> ... to the north is more hall.
 [ACTION] north
 [REWARD] +100

1771
1772
1773
1774
1775
1776
1777

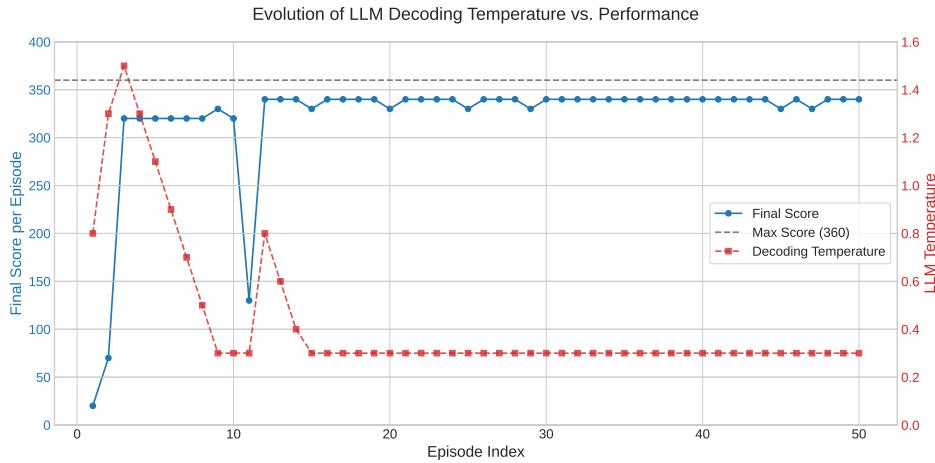
[STEP 49]
 [OBS] << room # 30 >> ... after a harrowing gun battle you conk
 him on the head...
 to leave this darned game, go up.
 [ACTION] up

1778
1779
1780
1781

Analysis: This transcript shows the agent in its most effective state. The sequence ‘west, north, west, north, north’ (Steps 44-48) is executed perfectly, demonstrating that the agent is no longer exploring but is following a precise, validated plan. Each action leads directly to progress. This flawless execution of a complex, non-obvious path is a hallmark of the converged EvoTest agent and highlights the power of evolving a detailed, procedural policy.

1782 Table 9: Comparison of AUC scores on a subset of games where all methods use
 1783 qwen/qwen3-32b as the actor backbone LLM. EvoTest continues to outperform both weight-
 1784 update baselines, highlighting the robustness of the evolutionary learning algorithm.

Method	Detective	Zork1	Balances	Avg.
SFT (online)	0.40	0.11	0.22	0.24
GRPO (online)	0.55	0.07	0.30	0.31
EvoTest (Ours)	0.78	0.12	0.31	0.40



1808 Figure 9: Evolution of the Actor’s LLM temperature versus its performance on the *Detective* game.
 1809

L EXPERIMENT WITH QWEN3-32B BACKBONE

1815 To provide a more direct and controlled comparison against the weight-update baselines, we con-
 1816 ducted an additional experiment using qwen/qwen3-32b as the backbone LLM for all com-
 1817 pared methods. In our main experiments, the weight-update methods (SFT and GRPO) use
 1818 qwen/qwen3-32b, while EvoTest uses proprietary models. This supplementary experiment aims
 1819 to normalize the underlying model capabilities to better isolate the effectiveness of the learning
 1820 algorithms themselves.

1821 In this setup, the Actor Agent in the EvoTest framework was switched from
 1822 google/gemini-2.5-flash to qwen/qwen3-32b. The Evolver Agent continued to
 1823 use openai/o3-2025-04-16 to maintain its strong analytical capabilities, ensuring that
 1824 EvoTest’s performance reflects its architectural strengths rather than being limited by a weaker
 1825 optimizer. The results, presented in Table 9, compare EvoTest against SFT and GRPO on a rep-
 1826 resentative subset of games, with all three methods now relying on the same qwen/qwen3-32b
 1827 backbone for their acting component.

1828 The results confirm that EvoTest’s performance advantage holds even in this controlled setting. As
 1829 shown in Table 9, EvoTest surpasses both SFT (online) and GRPO (online) on all three tested games.
 1830 The average AUC score for EvoTest (0.40) is substantially higher than GRPO (0.31) and SFT (0.24).
 1831 This indicates that the superiority of our evolutionary approach is not merely an artifact of using a
 1832 more powerful backbone model in the main experiments. Instead, it underscores the fundamental
 1833 data efficiency of EvoTest’s learning mechanism. By leveraging rich, narrative feedback from the
 1834 entire episode transcript for whole-system evolution, EvoTest can make more significant and tar-
 1835 geted improvements from a single experience than gradient-based methods relying on sparse scalar
 1836 rewards.

1836 Table 10: Comparison of AUC scores on a subset of games different LLM backbones. While
 1837 stronger models moderately improve the Static agent’s performance, the benefit is significantly
 1838 greater for EvoTest, demonstrating the scalability of our learning framework.

Method	Backbone LLM	Detective	Zork1	Balances	Avg.
Static	gemini-2.5-flash	0.21	0.03	0.11	0.12
EvoTest	gemini-2.5-flash	0.94	0.14	0.32	0.47
Static	claude-4-sonnet	0.23	0.04	0.12	0.13
EvoTest	claude-4-sonnet	0.95	0.16	0.35	0.49
Static	gemini-3-pro-preview	0.25	0.05	0.14	0.15
EvoTest	gemini-3-pro-preview	0.96	0.19	0.40	0.52
Static	gpt-5.1	0.28	0.06	0.17	0.17
EvoTest	gpt-5.1	0.98	0.24	0.48	0.57

M DYNAMIC HYPERPARAMETER EVOLUTION: A CASE STUDY

1854 Figure 9 illustrates how the Evolver intelligently manages the Actor’s LLM decoding temperature
 1855 in response to its performance. The Evolver’s logic is not random; it follows an adaptive strategy to
 1856 balance exploration and exploitation.

N SCALABILITY WITH MORE CAPABLE MODELS

1860 To investigate how our framework scales with model improvements, we evaluated EvoTest and
 1861 the non-learning Static baseline using two next-generation models: `openai/gpt-5.1` and
 1862 `google/gemini-3-pro-preview`. The results, presented in Table 10, show a positive trend:
 1863 as the underlying model’s capability improves, the performance of both agents increases. As models
 1864 become more capable, EvoTest has more raw material to work with, leading to disproportionately
 1865 larger gains compared to a simple zero-shot approach.