EVOHARNESS: Self-Evolving Tools, Memory, and Skills for Long-Horizon Play in Slay the Spire 2

Anonymous Authors[1]

Abstract

Self-evolving agents need mechanisms for improving their procedures without turning every lesson into an opaque prompt edit or a model update. We present EVOHARNESS, a method for evolving the external harness around a fixed language-model player. In *Slay the Spire 2* (STS2), the harness is divided into three adaptive layers: typed retrieval tools for game facts, scoped memory for run-local and cross-run state, and reusable skills for procedural decisions. Each run produces auditable artifacts; the same fixed model is then invoked in reflection, evolution, and validation roles, while durable changes must pass through a governed promotion protocol before entering the long-term harness. STS2 provides a long-horizon testbed with tactical combat, deck construction, routing, and delayed credit assignment, making it useful for studying when a lesson should become a tool, a memory, a skill, or remain an untrusted hypothesis. In our pilot evaluation, the EVOHARNESS profile successfully completes STS2 using GPT-5.4, showing that external harness evolution can substantially improve long-horizon play without training. Our code is available at `https://anonymous.4open.science/r/submissionC04C`

1. Introduction

Large Language Models (LLMs) have achieved strong performance on short-horizon tasks such as question answering, tool calling, and code generation (Tan et al., 2025; Xu et al., 2026; Huynh & Lin, 2025; Zhu et al., 2026). However, real-world applications typically demand complex long-horizon tasks, where an agent must use tools, manage context, convert repeated experience into reusable procedures, and avoid corrupting its future behavior with noisy lessons (Wan et al., 2025; Zhou et al., 2025). Naively appending every observation to the prompt is not a solution: it increases context cost, mixes transient state with durable knowledge, and blurs the boundary between useful rules and one-off anecdotes. We identify two key challenges facing long-horizon agents. *First*, the agent must decide what to remember, what to forget, and how to structure memories so that relevant information is accessible without overwhelming the working context (Liang et al., 2026). *Second*, the agent must distill noisy trial outcomes into generalizable strategies while balancing exploration of new approaches against exploitation of known solutions (Yang et al., 2025)

To overcome these challenges, we propose **EVOHARNESS**, a self-evolving framework that jointly optimizes tools, memory, and skills for long-horizon tasks. We evaluate EVOHARNESS on *Slay the Spire 2* (STS2) (CharTyr, 2026), a turn-based roguelike deck-building game that demands sustained strategic reasoning across extended trajectories. STS2 spans 48 floors, with the 17th, 35th, and 48th floors serving as challenging *Boss* encounters. During combat, the agent makes decisions through a tool-mediated game interface while the harness supplies factual lookup tools, scoped memory, and procedural skills; between combats, the agent also handles shop purchases, route selection, event choices, and card drafting from randomized rewards—encompassing the full decision surface illustrated in Figure 1. Succeeding at STS2 therefore requires drafting synergistic cards, managing limited resources (energy, gold, and potions), anticipating enemy behaviors, and navigating branching map routes under partial observability. Early missteps in deck construction or route selection can cascade into failure dozens of floors later, and a high density of cards, relics, events, and enemy patterns continually reshapes viable strategies. This makes STS2 a compact yet demanding testbed for studying long-horizon reasoning and self-evolution in LLM agents.

EVOHARNESS evolves its external harness over repeated sessions through a closed-loop pipeline. Each run produces auditable artifacts: raw model and tool events, a compact trajectory, metrics, and final state. A reflection role then diagnoses reusable lessons and proposes candidate updates to the narrowest suitable layer: factual tools, scoped memory, or procedural skills. An evolution role stages admitted candidates in an overlay and records promotion decisions in a manifest. When save/load snapshots are available, high-value candidates can be tested through replay from named

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

game states before being promoted to the long-term harness. The key design choice is that the player model remains fixed; adaptation occurs entirely within governed external artifacts whose provenance, validation status, and rollback boundaries are explicit.

With EvoHarness, we conduct comprehensive comparative experiments across different random seed pairs. All baseline methods fail at the 17th floor, where the Act 1 boss awaits; in contrast, EvoHarness substantially improves performance, consistently bypassing at least the 33rd floor and even achieving victory by clearing all 48 floors, demonstrating the efficacy of our agentic pipeline.

This paper makes three contributions:

- We formulate harness evolution as a layered object of study, separating typed retrieval tools, scoped memory, and reusable skills around a fixed player model.
- We describe a working implementation with clean baseline and evolved profiles, floor/run reflection, constrained evolution write surfaces, and auditable run artifacts.
- We report a first paired pilot from the current evaluation protocol and define the promotion and validation rules needed for larger repeated trials.

2. Related Work

Tool use, reflection, and skills. Tool-augmented agents such as ReAct (Yao et al., 2023) show that language models can combine reasoning with external actions, while MCP (Anthropic, 2024) standardizes one practical interface for tool servers. Reflection-based agents such as Reflexion (Shinn et al., 2023) and skill-library systems such as Voyager (Wang et al., 2023) show that agents can reuse verbal or procedural experience without changing model weights. EvoHarness builds on these ideas but treats post-run adaptation as a governed systems problem: tools, memory, and skills are separate layers with different write permissions.

Memory and verification. Long-term memory systems such as MemGPT (Packer et al., 2024) motivate explicit memory management instead of unlimited prompt accumulation. Formal mathematics resources such as miniF2F (Zheng et al., 2022) and LeanDojo (Yang et al., 2023) highlight the value of verifier-backed progress signals, while scientific-agent systems such as ChemCrow (Bran et al., 2023) and The AI Scientist (Lu et al., 2024) motivate tool-using agents that revise workflows over time. STS2 is not a formal scientific domain, but it exposes delayed credit assignment and evolving state in a controlled environment. We use it to study promotion, provenance, and rollback policies that can later be paired with stronger external checkers.

LLMs in games. Recent work studies LLM agents in strategic game environments (Costarelli et al., 2024; Bakhtin et al., 2022). Deck-building roguelike games add a particular mixture of deck construction, tactical combat, and long-term planning. Bateni & Whitehead (2024) found that language agents can be competitive in long-horizon planning while remaining weaker at short-term tactical optimization, matching the motivation for a harness that separates factual lookup, memory, and procedural guidance.

3. Problem Formulation

3.1. Game Task Anatomy

Slay the Spire 2 (STS2) is an Early Access roguelike deck-building game whose difficulty comes from repeatedly connecting local actions to run-level consequences (Mega Crit, 2026). Figure 1 breaks this decision surface into four recurring interfaces. In a combat turn, the agent must choose cards, targets, and potions while reconciling persistent state (HP, gold, relics, and potions), the current action budget (energy and hand), and enemy intent or status. In a shop or deck-economy room, a limited gold budget constrains purchases among cards, relics, potions, and removal, and the choice changes the future deck trajectory. In route planning, the player commits to branches whose node types determine future risk and rewards, while previous path choices restrict which options remain reachable. In an event decision, a narrative observation is converted into cost/reward actions, often trading an immediate HP or deck cost for delayed value.

The environment is therefore not a pure text benchmark. The agent receives structured game state and legal actions through MCP tools, but it must still decide which facts to retrieve, which transient state to remember, and which local choice changes the long-horizon plan. There are no expert labels specifying the correct card, route, target, or potion at each boundary. The feedback that matters is sparse and delayed: death or victory, reached floor, boss and elite kills, invalid actions, and resource use.

3.2. Decision Surface and Scale

We use STS2 because it combines a large discrete fact space, long run horizon, and dense local interaction while remaining fully loggable. The current harness index covers 504 card records, 292 relic records, 63 potion records, 113 monster records, 56 event records, 95 ancient-option records. The horizon is also long: the completed evolved pilot reaches floor 48, so the player must preserve route, deck, and resource commitments across dozens of room decisions before receiving a terminal signal. Local rooms are interaction-heavy rather than single choices. In sampled local records, each combat-labeled floor requires 57 tool

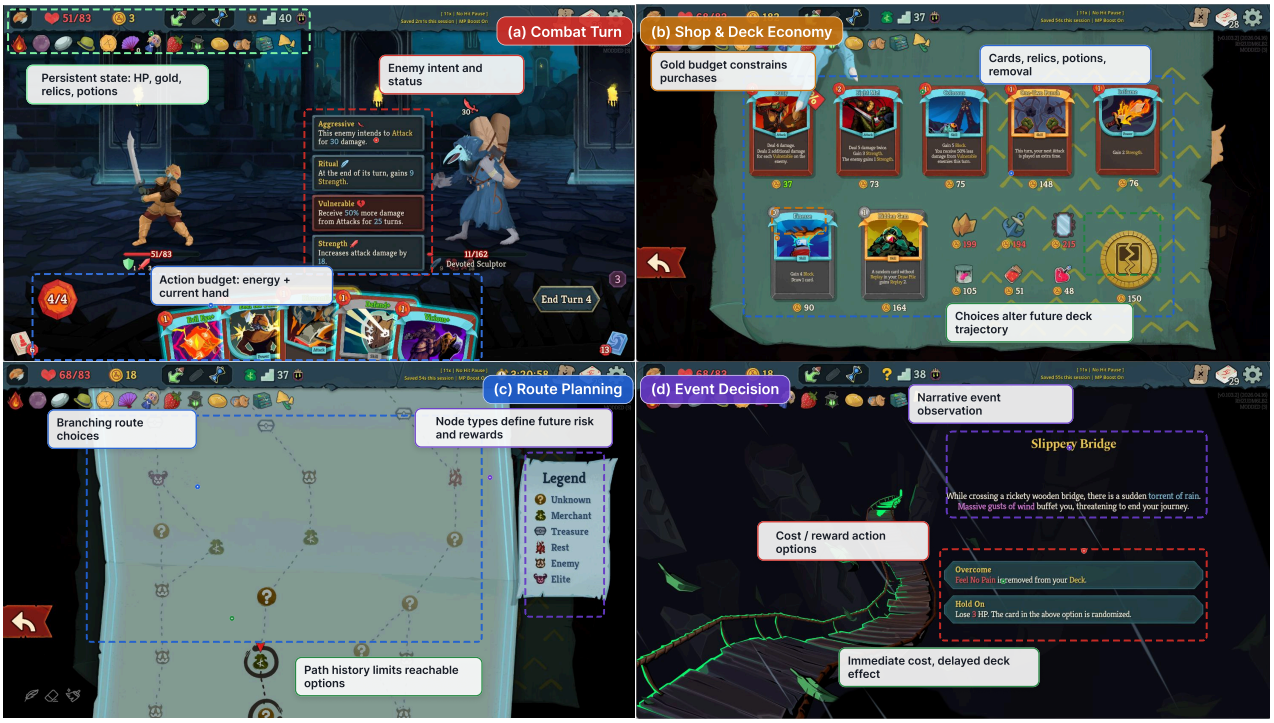


Figure 1. STS2 decision surface. The four annotated panels show the main decision interfaces used in our problem setting: turn-level combat under hand, energy, HP, and enemy-intent constraints; shop decisions that trade gold for cards, relics, potions, or removal; route planning over a branching map with typed nodes; and event choices with immediate costs and delayed deck effects.

calls on average and peaks at 141.

This scale matters because the hard part is not just choosing a legal action. The player must keep a route-level plan alive while repeatedly deciding which facts to retrieve, which transient observations to remember, which combat calculations to redo, which lessons should become reusable procedures, and which one-run anecdotes should be ignored. For example, a reward decision may require classifying cards as immediate attack, defense, draw, energy, or payoff; a route decision may require checking future campfires and shops; a combat decision may require matching enemy move patterns to the current hand and potion breakpoints. These dependencies make STS2 a useful stress test for self-evolving harnesses.

3.3. Game and Harness Objective

We model STS2 as an episodic partially observable Markov decision process:

$$\mathcal{G} = (\mathcal{S}, \mathcal{O}, \mathcal{U}, P, \Omega, \rho).$$

The full game state $s_t \in \mathcal{S}$ includes hidden or inconvenient-to-reconstruct details such as draw order, future map consequences, and accumulated run state. The agent receives a tool-mediated observation $o_t \in \mathcal{O}$ through the game API, including the visible screen, structured state, and legal action schema. At decision boundary t, STS2 exposes a legal action set $\mathcal{U}(o_t)$, and the player has access to the interaction history

$$h_t = (o_1, u_1, \ldots, o_t).$$

The player chooses an action

$$u_t \in \mathcal{U}(o_t),$$

after which the environment transitions according to $P(s_{t+1} \mid s_t, u_t)$ and exposes the next observation through $\Omega(o_{t+1} \mid s_{t+1})$. A run terminates when the game reaches death, victory, or an experiment budget. The terminal and dense evaluation signals ρ include reached floor, outcome, boss and elite kills, invalid action rate, token cost, tool-call cost, and token-normalized progress.

We study the setting where the player model π_θ is fixed and the adaptable object is an external harness $H \in \mathcal{H}$. The harness does not change the game dynamics P, the observation process Ω, or the model parameters θ. Instead, it changes the information, memory, and procedures available to the player, inducing a harness-conditioned policy

$$\pi_\theta^H(u_t \mid h_t) = \pi_\theta(u_t \mid h_t, H), \qquad u_t \in \mathcal{U}(o_t).$$

For a run trajectory τ, let $J(\tau)$ denote the scalar utility induced by ρ, including both progress and cost terms. The harness optimization problem is to design an update rule

$$H_{k+1} = \textsc{Evolve}(H_k, \pi_\theta, \mathcal{G}, D_{\le k})$$

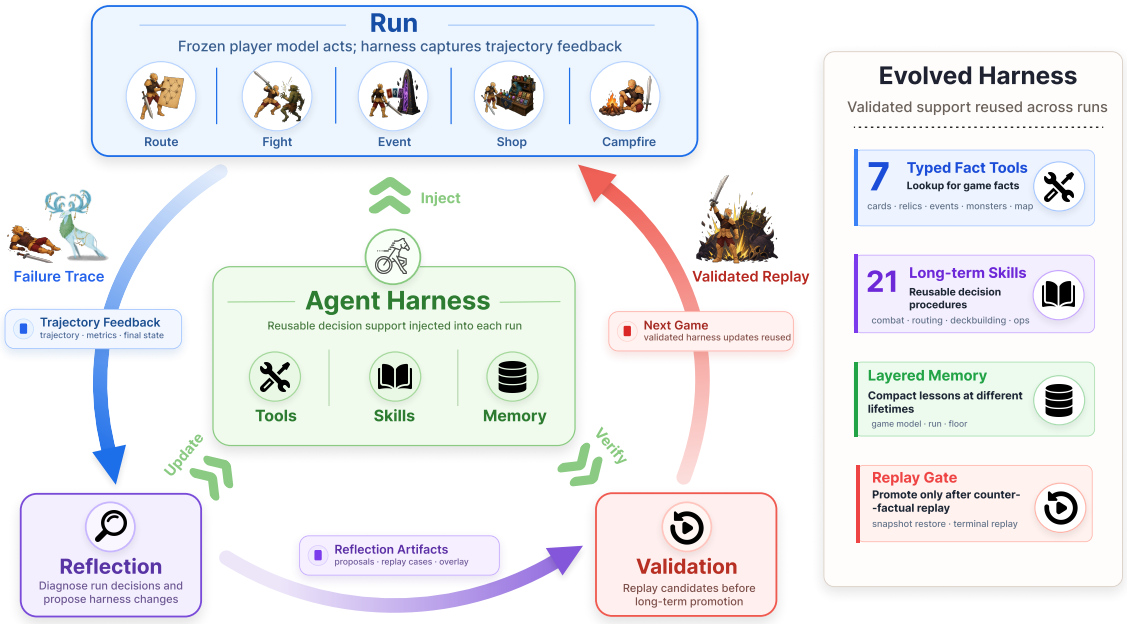


Figure 2. AGENTHARNESS method overview. A fixed player model produces run artifacts from STS2 gameplay, then the same model is invoked in reflection, evolution, and validation roles to update the external harness rather than the model weights. The harness is organized into typed tools, scoped memory, and procedural skills; candidate changes are staged and promoted only after validation, with the right-hand summary highlighting the kinds of durable tool, skill, and memory updates produced by the loop.

that improves

$$\mathbb{E}_{\tau\sim(\mathcal{G},\pi_\theta^{H_{k+1}})}[J(\tau)]$$

while keeping θ fixed and respecting the legal action set of the game.

4. Method

4.1. Overview

EVOHARNESS is a validation-gated method for updating a three-layer harness around a fixed model. The implementation names separate play, reflection, evolution, and validation roles, but these roles are invocations of the same frozen model rather than independently trained agents. The method augments the fixed player π_θ with an external harness

$$H_k = (T_k, M_k, S_k),$$

where T_k is the typed tool layer, M_k is scoped memory, and S_k is the skill library available during run k. During play, the model samples from the legal actions using both game history and the current harness,

$$u_t^k \sim \pi_\theta(\cdot \mid h_t^k, H_k),$$

while the model parameters θ remain fixed across all runs. The rollout produces method-side artifacts

$$A_k = \{\tau_k, m_k, s_k^{\text{final}}\},$$

where τ_k is the logged trajectory, m_k contains progress and cost metrics, and s_k^{final} is the final game state. The reflection role maps these artifacts into candidate lessons,

$$R_k = \textsc{Reflect}(A_k, H_k, \pi_\theta, \mathcal{G}).$$

The central method design is the update rule

$$H_{k+1} = \textsc{Evolve}(H_k, \pi_\theta, \mathcal{G}, A_k, R_k),$$

which may stage, reject, validate, or promote changes without updating θ. Figure 2 summarizes the pipeline:

$$H_k \xrightarrow{\text{play}} A_k \xrightarrow{\text{reflect}} R_k \xrightarrow{\text{stage}} C_k \xrightarrow{\text{validate}} H_{k+1}.$$

4.2. Harness Components

The harness has three components, each with a distinct role in the agent's decision process and a different admission rule for updates. Table 1 summarizes the division. The purpose of the split is practical: facts should be looked up through tools, transient context should be kept in scoped memory, and repeatable decision procedures should be packaged as skills. This prevents the system from solving every problem by appending more prose to the main prompt.

Tools define the player's observable and executable interface to STS2. The pinned STS2 MCP server (CharTyr, 2026) exposes game state, legal actions, and action endpoints for combat, rewards, shops, events, rests, upgrades, and route

Table 1. Harness layers and their roles in EVOHARNESS. Each layer has a different update boundary, so reflected lessons are routed before they can become durable state.

Layer	Role during play	Admissible evolution
Tools	Observe game state, enumerate legal actions, execute selected actions, and retrieve card, relic, potion, monster, event, synergy, and map facts.	Add or revise factual interfaces, schemas, indexes, and registry entries only after import, schema, and test checks.
Memory	Compact state across turns, floors, and runs without replaying the full history.	Append run notes, replace floor context, prune stale state, or promote repeated lessons.
Skills	Loadable procedures for combat, deckbuilding, routing, and operations.	Require triggers, state signals, cautions, evidence, and validation plans.

choices. The harness-owned extension tools add read-only typed lookup over cards, relics, potions, monsters, events, synergies, and map branches. Thus, tools ground decisions in current state and external game facts: they tell the agent what can be done now and what relevant objects mean.

Memory stores scoped state. The long-term game model holds compact cross-run priors. The run overlay tracks current deck direction, route goals, shop priorities, and unresolved weaknesses. The floor overlay holds short-lived combat or room context and is pruned at boundaries. Memory updates therefore implement a write, overwrite, retrieve, and forget policy rather than cumulative logging.

Skills store reusable procedures. A skill specifies load triggers, state signals, reference checks, decision frames, and cautions. Skills are organized by decision type, such as combat, deckbuilding, and routing. They guide attention at future decision boundaries without enforcing fixed card-pick or route rules. This division lets the evolution role route a lesson to the narrowest layer that can express it.

4.3. Reflection and Candidate Generation

After a run, the reflection role reads logs, metrics, final state, and optional floor reflections. It emits a structured reflection containing failure classification, evidence spans, target layer, proposed change, risk, and validation plan. The reflection role is deliberately only a candidate generator: it may diagnose that a death involved slow deck construction or poor potion timing, but that diagnosis does not automatically become a long-term rule. This separation lets the system treat reflection outputs as testable hypotheses rather than trusted updates.

The evolution role converts admitted proposals into a candidate overlay C_k. The overlay may contain staged skill files, run-local memory, or tool-registry candidates. Thin or unsafe lessons are rejected before staging when they lack evidence, duplicate existing guidance, encode a one-run accident, conflict with the current harness, or merely increase prompt length. This overlay gives the system a place to inspect and test changes without directly mutating H_k.

Algorithm 1 Post-Run Harness Evolution Pipeline

Input: run artifacts $\mathcal{A}$ (trajectory, metrics, final state)
Output: evolved harness H' and manifest $\mathcal{M}$
$R \leftarrow \textsc{ReflectRun}(\mathcal{A})$
$C \leftarrow \emptyset$ {candidate overlay}
for each proposal $p \in R.\text{tool\_changes} \cup R.\text{memory\_changes} \cup R.\text{skill\_changes}$ **do**
  if $\textsc{ValidateProposal}(p)$ **then**
    $C \leftarrow C \cup \{\textsc{StageCandidate}(p)\}$ with provenance and evidence
  end if
end for
$V \leftarrow \textsc{ValidateOverlay}(H, C)$ {schemas, budget, tests, optional replay}
$H' \leftarrow \textsc{PromoteValidated}(H, C, V)$
$\mathcal{M} \leftarrow \textsc{WriteManifest}(H, H', R, V)$

4.4. Validation-Gated Promotion

Promotion is target-specific. Text artifacts must satisfy schema, evidence, conflict, and memory-budget checks. Harness-owned code must pass import and test checks before it can be considered for promotion. When save/load snapshots are available, a staged candidate can be evaluated from a named game state through counterfactual replay. Restore alone is only a smoke check; promotion requires evidence that the staged change supports a better or safer trajectory. This gate keeps unverified lessons out of the durable harness. It also limits the risk that a plausible but accidental post-run explanation is reused as general guidance in later runs. As a result, harness evolution remains a controlled update process rather than an unrestricted accumulation of self-generated advice.

The evolution role writes a manifest that records which candidates were promoted, staged, rejected, or left pending, together with the evidence and validation outcome for each decision. This manifest provides an audit trail and a rollback boundary for later harness changes. The next run uses the promoted long-term harness plus its own session overlay. Algorithm 1 gives the post-run procedure.

5. Experiments

5.1. Setup

Harness construction. Our evaluation separatess *harness construction* from *harness evaluation*. Starting from a seed harness H_0, we run ten evolution cycles on prior prior STS2 trajectories. In these cycles, the configured fixed model is invoked in identifies and evolution roles: it reads trajectory and metric artifacts, identifies failures or reusable lessons, stages candidate harness changes, and promotes only validated updates. These between-run invocations update external harness artifacts, not the player model parameters. This process produces the evolved harness H_{10} used in our evaluation. Concretely, H_{10} used in our evaluation contains 7 harness-owned harness-owned typed fact tools andand 21 long-term skills distilled from prevevious trajectories.

Evaluation protocol. We evaluate H_{10} under a fixed-player protocol. All reported gameplay runs use the same configured model, invoked through the same runner interface.

We compare two runner profiles. The **baseline** profile exposes only the pinned STS2 agent MCP server (CharTyr, 2026) and a fixed player prompt, so it can observe structured game state, query legal actions, and execute legal game actions. It receives no evolved skills, long-term memory, run/floor memory, extension lookup tools, floor reflection, or post-run evolution context. The **evolved** profile uses H_{10}: the same gameplay MCP server plus typed fact tools, scoped memory, memory-management system and categorized skills. Further details are provided in Appendix A.

We evaluate three paired starts, where each pair is a recorded STS2 snapshot. We run the baseline profile until death or victory, then restore the same snapshot and run the evolved profile under the same budget. Snapshot restore is performed by the experiment runner and is not exposed. The trajectories may diverge after restore because the two profiles can make different choices, but the initial state, player model, runner interface, and budget are controlled.

Metrics. We report both progress and cost. The progress columns are Victory and Floor: Victory marks a winning run (✓) versus death (×), while Floor is the terminal reached floor F. For STS2, floor 48 awaits the final-boss and victory is counted only if this encounter is cleared. The efficiency columns normalize run totals by reached floor: New/F includes input, output, and reasoning tokens excluding cache reads; Cache/F adds cache-read tokens for context-reuse accounting; and Tool/F counts one invocation per pending tool-use event, including gameplay and harness tools.

5.2. Main Result

Table 2 reports three paired pilot runs from matched STS2 starts with the player model fixed. The clearest signal is the

Table 2. Paired main results on three matched starts with the player model fixed. Victory marks a winning run (✓) or death (×).

	Victory	Floor	New/F	Cache/F	Tool/F
Pair 1					
Baseline	×	17	1,489k	622k	15.0
Evolved	✓	48	3,305k	1,443k	30.3
Pair 2					
Baseline	×	17	1,823k	1,325k	21.9
Evolved	×	33	1,591k	927k	20.2
Pair 3					
Baseline	×	17	1,935k	1,361k	22.3
Evolved	×	48	2,961k	1,785k	27.0

Act 1 boss bottleneck: all baseline runs die on floor 17. This is a stronger finding than generic early death. It indicates the baseline handles local first-act demands—hallway fights, shops, events, short-horizon rewards—but fails when the run is tested by a boss fight that reflects the quality of earlier run-level management.

In STS2 terms, the Act 1 boss is an early strategic checkpoint. By floor 17, the player must have converted the first act into a boss-ready state through coupled decisions: card rewards and removals, upgrades, relic use, potion conservation, health preservation, elite and shop routing, rest-site choices, and boss-specific preparation. A run adequate against hallway enemies can still fail here if these decisions do not form a coherent risk-management strategy. The baseline's repeated death at the same floor show that it does not reliably turn early-act opportunities into a state that survives the first major boss test.

The evolved harness breaks this bottleneck in every matched pair, reaching floors 48, 33, and 48. Pair 1 is a victory; Pair 3 reaches the final boss but does not win. Averaged across the three starts, reached floor improves from 17.0 to 43.0. This does not show uniform local superiority, given the small sample, but it does show a consistent qualitative change: the fixed player inside the evolved harness survives past the first major run-management threshold in all three cases.

The cost columns show this improvement is not free. In Pair 1, the evolved profile spends 3,305K new tokens and 30.3 tool calls per floor, versus 1,489K and 15.0 for the baseline. Pair 3 shows a similar tradeoff: 2,961K new tokens and 27.0 tool calls per floor, above the baseline's 1,935K and 22.3. These pairs suggest that longer, more successful runs often require more model context and tool interaction, even after normalizing by floor.

Pair 2 is different and important for interpretation. The evolved profile reaches floor 33, nearly double the baseline depth, while using *fewer* new tokens per floor (1,591K vs. 1,823K), fewer cache-inclusive tokens per floor (927K vs.

1,325K), and fewer tool calls per floor (20.2 vs. 21.9). This shows the evolved harness is not only spending more computation to get farther; it can also make computation more productive by reducing wasted reasoning, repeated state reconstruction, or poorly targeted tool use. The harness can improve decision quality either by adding useful structure at higher cost (Pairs 1 and 3) or by redirecting the same fixed player toward more efficient long-horizon play (Pair 2).

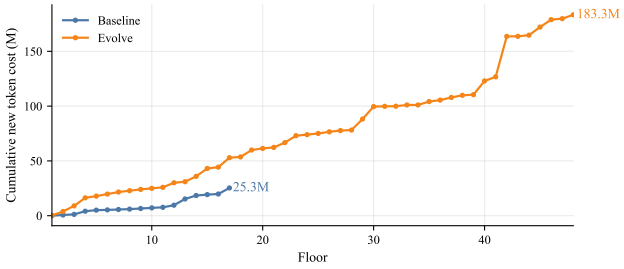
These results support a specific claim: harness evolution helps the fixed player cross the first major run-management threshold in STS2. The baseline repeatedly reaches the Act 1 boss and fails there, indicating a persistent weakness in converting early rewards, routing, health, potions, relics, and combat resources into a boss-ready state. The evolved profile, with access to skills, scoped memory, and typed factual tools, survives past that threshold in all three matched starts. This suggests the evolved harness improves the agent's ability to preserve run-level commitments across many local decisions, rather than only improving isolated combat actions.

Figure 3 explains the resource and survival patterns behind Pair 1. The baseline curves stop at floor 17 because the run ends at the Act 1 boss, while the evolved curves continue to the victory floor. Its cumulative token and tool-call costs therefore grow much larger, reflecting both the longer trajectory and the greater planning burden of later floors. As the run progresses, decisions become more history-dependent: the agent must reason over the accumulated deck, relics, potions, health, upgrades, removals, route commitments, and upcoming boss or elite threats.

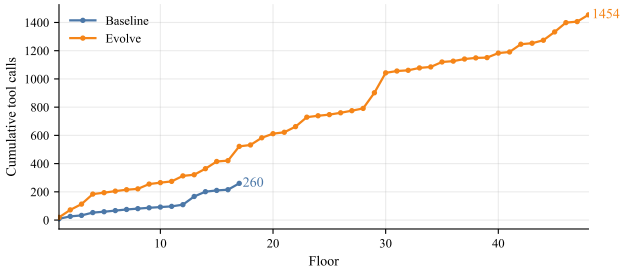
The HP trace adds a risk-management view. It should be read with two game conventions in mind: clearing a boss restores HP at the act transition, and the final zero HP in the evolved run marks the terminal victory state rather than a death. With these conventions, the contrast is still informative. The baseline collapses at the Act 1 boss, whereas the evolved profile survives several later low-HP regions and completes the run. Taken together, the traces show that EvoHarness reaches states the baseline does not, but doing so can require higher systems cost. Across all three pairs, this supports a progress–cost tradeoff: the evolved harness consistently moves the fixed player beyond the Act 1 boss bottleneck, while token and tool-call costs remain first-class metrics for judging how that progress is obtained.

5.3. Ablation Study

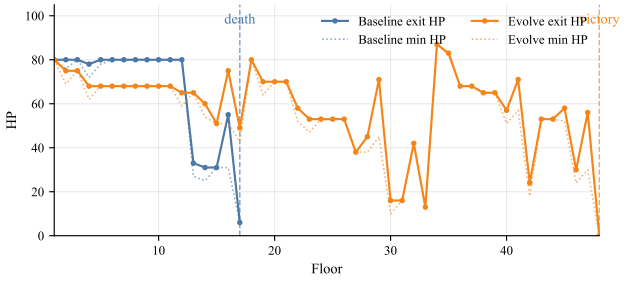
Table 3 studies which harness layers are responsible for the Pair 1 improvement. The baseline reaches floor 17 with 1,489K new tokens per floor and 15.0 tool calls per floor. Adding only typed factual tools still ends at floor 17, with slightly lower new-token cost but slightly higher tool use. This suggests that tools are not enough as a passive interface:



(a) Cumulative new-token cost.



(b) Cumulative tool calls.



(c) HP trajectory.

Figure 3. Trajectory diagnostics for the first pair. The evolved profile survives to victory, with higher cumulative token and tool-call costs over the longer trajectory. The HP trace shows the corresponding survival dynamics and the final HP drop marks the terminal victory state rather than a death.

without skills, the fixed player lacks a reliable procedure for deciding when to query a tool, which fact to request, and how to use the returned information in route, reward, or combat decisions. Adding only scoped memory also ends at floor 17, while reducing new-token cost. This is consistent with memory mainly reducing repeated state reconstruction, rather than creating a stronger run-management policy by itself.

The skill-only condition is the first partial condition that changes run progress, reaching floor 30. This suggests that skills are the strongest individual layer because they provide explicit procedures for recurring decision points, such as card reward selection, elite readiness, potion timing, boss preparation, route risk, and combat resource accounting. However, skill-only still falls short of the full harness. Without typed factual tools and scoped memory, the player must still recover many game facts and run-local commitments

Table 3. Layer-wise ablation results on Pair 1 with the player model fixed. Tools denotes typed factual tools, while Tool/F counts all tool calls. New/F reports average new tokens per reached floor, shown in thousands.

Condition	Tools	Memory	Skills	Floor	New/F	Tool/F
Baseline	–	–	–	17	1,489k	15.0
Tool-only	✓	–	–	17	1,363k	16.4
Memory-only	–	✓	–	17	1,279k	16.3
Skill-only	–	–	✓	30	1,909k	24.1
Full harness	✓	✓	✓	48	3,305k	30.3

through ordinary context, making the procedural guidance useful but incomplete.

The full harness is the only condition that reaches floor 48, but it also has the highest per-floor cost: 3,305K new tokens and 30.3 tool calls. This supports the central design hypothesis of EVOHARNESS: tools, memory, and skills are complementary rather than interchangeable. Skills provide reusable decision procedures, typed factual tools ground those procedures in game facts, and scoped memory preserves run-local commitments such as deck direction, route goals, boss preparation, and unresolved weaknesses. The key observation is that this added cost corresponds to a qualitative progress change: only the full harness crosses the Act 1 boss boundary and reaches the final-floor region.

6. Conclusion and Discussion

What improved. The pilot results suggest that harness evolution helps when the main failure is not a lack of legal actions, but a failure to organize long-horizon state. The baseline repeatedly reaches floor 17 and dies at the Act 1 boss, which indicates that it can survive many early local decisions but cannot reliably convert them into a boss-ready run. By that point, the player has already committed to a deck direction, route, health budget, potion plan, upgrade plan, and boss preparation state. The evolved harness helps the same fixed player preserve these commitments across floors, so the main gain appears as crossing the Act 1 boss bottleneck rather than only as better single-turn play. This supports that external harness evolution can change the reachable trajectory without model training.

Why layers must interact. The ablation results show that the three harness layers are complementary. Typed factual tools alone do not improve reached floor because, without skills, the player has no stable procedure for deciding when to query a fact or how to use it. Memory alone mainly reduces repeated context reconstruction, but it does not create a better run-management policy. Skills are the strongest individual layer because they provide procedures for recurring decision boundaries such as card rewards, elite readiness, potion timing, route risk, and boss preparation. However, skill-only still stops before the full harness. This indicates that procedures, facts, and run-local commitments need to work together: skills define how to decide, tools supply needed game facts, and memory keeps the current run plan available across floors.

Cost and risk management. The progress gains come with a systems cost. In Pair 1 and Pair 3, the evolved harness reaches much deeper states but uses more tokens and more tools per floor. This is partly due to the longer and more history-dependent trajectory: later decisions depend on the accumulated deck, relics, potions, health, upgrades, removals, route commitments, and upcoming elite or boss threats. The HP trace in Pair 1 adds a survival view. After accounting for boss-clear HP recovery and the terminal victory zero, the evolved profile still passes through several low-HP regions and completes the run, while the baseline collapses at the Act 1 boss. Thus, the harness improves both action selection and risk management over a longer run.

Evidence boundary and failure modes. The current evaluation is intended as a systems pilot: it tests whether external harness evolution can change the reachable trajectory of a fixed player under controlled starts. Under this goal, the paired results provide a clear signal: the baseline repeatedly stops at the Act 1 boss, while the evolved harness crosses that boundary in all three starts and completes one run. The ablation further identifies a concrete mechanism, showing that skills drive the largest individual gain and that full progress requires tools, memory, and skills to work together. A larger repeated study is still needed to estimate win-rate gains across more starts, characters, boss matchups, and route structures. The main scaling risks are single-run overfitting, memory growth, skill conflicts, and tools that hide policy behind an apparently factual interface; these risks motivate the governed update design, where candidate status, evidence, validation, pruning, and rollback boundaries remain explicit.

Impact Statement

This paper studies infrastructure for long-horizon self-evolving agents. The experiments use a commercial game environment for research; the methods are not intended for safety-critical deployment without stronger validation. Potential positive impacts include more auditable self-improvement pipelines for scientific agents. Potential risks include over-trust in self-generated skills, memory, or tool changes without external verification. Our design mitigates this risk by separating candidate generation from promotion and by recording provenance, validation status, and rollback-relevant manifests for durable harness edits.

References

Anthropic. Model context protocol. https://modelcontextprotocol.io/, 2024. Technical specification for tool-mediated agent interfaces.

Bakhtin, A., Brown, N., Dinan, E., Farina, G., Flaherty, C., Fried, D., Goff, A., Gray, J., Hu, H., Jacob, A. P., Komeili, M., Konath, K., Kwon, M., Lerer, A., Lewis, M., Miller, A. H., Mitts, S., Renduchintala, A., Roller, S., Rowe, D., Shi, W., Spisak, J., Wei, A., Wu, D. J., Zhang, H., and Zijlstra, M. Human-level play in the game of diplomacy by combining language models with strategic reasoning. *Science*, 378:1067 – 1074, 2022. URL https://api.semanticscholar.org/CorpusID:253759631.

Bateni, B. and Whitehead, J. Language-driven play: Large language models as game-playing agents in slay the spire. *Proceedings of the 19th International Conference on the Foundations of Digital Games*, 2024. URL https://api.semanticscholar.org/CorpusID:270963094.

Bran, A. M., Cox, S., Schilter, O., Baldassari, C., White, A. D., and Schwaller, P. Chemcrow: Augmenting large-language models with chemistry tools, 2023. URL https://arxiv.org/abs/2304.05376.

CharTyr. STS2-Agent: A slay the spire 2 mod and mcp server for ai agent control. https://github.com/CharTyr/STS2-Agent, 2026. Accessed: 2026-05-26.

Costarelli, A., Allen, M., Hauksson, R., Sodunke, G., Hariharan, S., Cheng, C., Li, W., Clymer, J., and Yadav, A. Gamebench: Evaluating strategic reasoning abilities of llm agents, 2024. URL https://arxiv.org/abs/2406.06613.

Huynh, N. and Lin, B. Large language models for code generation: A comprehensive survey of challenges, techniques, evaluation, and applications, 2025. URL https://arxiv.org/abs/2503.01245.

Liang, J., Han, J., Li, W., Wang, X., Zhang, Z., Jiang, Z., Liao, Y., Li, T., Huang, Y., Shen, H., Wu, H., Guo, F., Wang, K., Hong, Z., Lu, Z., Ma, L., Jiang, S., and Xiao, Y. Genericagent: A token-efficient self-evolving llm agent via contextual information density maximization (v1.0), 2026. URL https://arxiv.org/abs/2604.17091.

Lu, C., Lu, C., Lange, R. T., Foerster, J. N., Clune, J., and Ha, D. The AI scientist: Towards fully automated open-ended scientific discovery. *CoRR*, abs/2408.06292, 2024. doi: 10.48550/ARXIV.2408.06292. URL https://doi.org/10.48550/arXiv.2408.06292.

Mega Crit. Slay the Spire 2 on Steam. https://store.steampowered.com/app/2868840/Slay_the_Spire_2/, 2026. Early Access store page; accessed 2026-05-24.

Packer, C., Wooders, S., Lin, K., Fang, V., Patil, S. G., Stoica, I., and Gonzalez, J. E. Memgpt: Towards llms as operating systems, 2024. URL https://arxiv.org/abs/2310.08560.

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

Tan, N., Seng, Z., Zhang, L., Shih, Y., Yang, D., and Salunkhe, A. Improved LLM agents for financial document question answering. *CoRR*, abs/2506.08726, 2025. doi: 10.48550/ARXIV.2506.08726. URL https://doi.org/10.48550/arXiv.2506.08726.

Wan, G., Ling, M., Ren, X., Han, R., Li, S., and Zhang, Z. Compass: Enhancing agent long-horizon reasoning with evolving context, 2025. URL https://arxiv.org/abs/2510.08790.

Wang, G., Xie, Y., Jiang, Y., Mandlekar, A., Xiao, C., Zhu, Y., Fan, L., and Anandkumar, A. Voyager: An open-ended embodied agent with large language models, 2023. URL https://arxiv.org/abs/2305.16291.

Xu, M., Wang, Z., DENG, M., Li, Z., Yang, Z., Zhu, X., Liu, Y., Zhu, B., Huang, B., Chen, C., Deng, H., Mi, F., Shang, L., Zeng, X., and Guo, Z. Envfactory: Scaling tool-use agents via executable environments synthesis and robust rl, 2026. URL https://arxiv.org/abs/2605.18703.

Yang, C., Yang, X., Wen, L., Fu, D., Mei, J., Wu, R., Cai, P., Shen, Y., Deng, N., Shi, B., Qiao, Y., and Li, H. Learning on the job: An experience-driven self-evolving agent for long-horizon tasks, 2025. URL https://arxiv.org/abs/2510.08002.

Yang, K., Swope, A. M., Gu, A., Chalamala, R., Song, P., Yu, S., Godil, S., Prenger, R. J., and Anandkumar, A. Leandojo: Theorem proving with retrieval-augmented language models. In Oh, A., Naumann, T., Globerson, A., Saenko, K., Hardt, M., and Levine, S. (eds.), *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December*

10 - 16, 2023, 2023. URL http://papers.nips.cc/paper_files/paper/2023/hash/4441469427094f8873d0fecb0c4e1cee-Abstract-Datasets_and_Benchmarks.html.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K. R., and Cao, Y. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

Zheng, K., Han, J. M., and Polu, S. minif2f: a cross-system benchmark for formal olympiad-level mathematics. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022. URL https://openreview.net/forum?id=9ZPegFuFTFv.

Zhou, Z., Qu, A., Wu, Z., Kim, S., Prakash, A., Rus, D., Zhao, J., Low, B. K. H., and Liang, P. P. Mem1: Learning to synergize memory and reasoning for efficient long-horizon agents, 2025. URL https://arxiv.org/abs/2506.15841.

Zhu, X., Zhou, X., Zhu, B., Hu, H., Du, M., Zhang, H., Wang, H., and Guo, Z. Codescaler: Scaling code llm training and test-time inference via reward models, 2026. URL https://arxiv.org/abs/2602.17684.

A. Implementation Details

A.1. Runner Profiles and Controls

The implementation keeps the clean comparison lane and the evolved lane separate. The baseline command, `npm.cmd run baseline`, plays a game with the pinned STS2 gameplay MCP server and a fixed baseline player prompt. It captures state, extracts trajectory and metric files, and builds a local viewer, but it does not inject skills, load memory, call harness-owned extension tools, run reflection, or write an evolution manifest. The evolved command, `npm.cmd run evolve`, starts a run with the evolved profile, enables floor reflection, then performs post-run reflection and evolution after the game segment finishes. For existing runs, `npm.cmd run evolve:post` runs only the post-processing lane, while `npm.cmd run evolve:dry-run` previews promotion decisions without writing long-term harness files.

Both profiles are intended to use the same fixed player model and the same terminal condition. The controlled object is therefore the harness profile, not the base model. The baseline profile exposes only the gameplay server; the evolved profile loads categorized skills, the STS2 Game Model, run and floor overlay memory, memory-management tools, the save/load snapshot server, and the harness-owned typed extension tools. The runner configuration sets a context limit, compaction threshold, maximum turn budget, request timeout, skill and memory directories, and a long-term memory budget of `4000` characters.

A.2. MCP Servers

Table 4. MCP surfaces used by the runner. The extension surface is intentionally factual: tactical and strategic judgment remains in the player, memory, and skills.

Server	Role	Exposed behavior
`sts2`	Live gameplay interface	Reads compact game state, enumerates legal actions, and executes game actions under the guided profile with debug actions disabled.
`sts2_sl`	Snapshot interface	Lists, inspects, and restores save/load snapshots through keys such as `pre:<act>:<floor>` and `post:<act>:<floor>`. Restore mutates the live game and is reserved for validation.
`sts2_ext`	Harness-owned reference interface	Provides typed local lookup over cards, relics, events, potions, monsters, synergies, and full-map branch summaries.

The extension server currently exposes `sts2_ext_search_cards`, `sts2_ext_search_relics`, `sts2_ext_search_events`, `sts2_ext_search_potions`, `sts2_ext_search_monsters`, `sts2_ext_search_synergies`, and `sts2_ext_view_full_map`. Search is local, typed, and lexical in the current version. Advisor-style tools that directly recommend combat lines or routes were removed from the stable surface because they risk hiding policy in code; the extension layer should provide facts, while route, deck, and combat judgment belongs to skills and memory.

A.3. Prompt Profiles

The runner selects the player prompt from the active harness profile. For transparency, Section E reproduces the baseline player prompt, evolved player prompt, run-reflection prompt, and replay-validation prompt used by the current implementation. The prompt text is shown directly rather than summarized.

B. Skill and Memory Layers

B.1. Skill Taxonomy and Admission Test

Skills are stored as markdown files under `harness/skills/<category>/<skill-name>/SKILL.md`. The seed taxonomy is open-ended and currently covers combat, deckbuilding, routing, and operations. A valid skill must specify a purpose, load triggers, state signals, optional reference checks, a decision frame, and cautions. The Evolver may create, rewrite, merge, split, deprecate, or delete skills, but the admission test rejects thin one-line lessons, duplicate guidance, hard card-pick orders, and advice without evidence or a trigger.

Table 5. Representative skill categories. Skills are procedural attention guides, not fixed policies.

Category	Example skills	Decision boundary
Combat	`combat-risk-and-targeting`, `potion-tempo`, `enemy-specific-tactics`	Lethal checks, block-versus-kill tradeoffs, target priority, potion timing, and elite or boss fight planning.
Deckbuilding	`reward-pick-discipline`, `act-1-frontload-and-coverage`, `synergy-readiness`	Card reward, transform, removal, archetype, coverage, and boss-matchup choices.
Routing	`choose-map-path`, `choose-shop-node`, `choose-elite-node`, `prepare-for-boss`	Branch comparison, campfire timing, shop leverage, event risk, elite readiness, and boss preparation.
Operations	`sts2-mcp-player`, `sts2-sl-snapshots`, `run-boundary-*`	Tool discipline, replay validation, and separation of distinct game episodes.

B.2. Memory Scope

Long-term game understanding lives in `harness/memory/game_model.md`. It stores compact cross-run priors such as treating HP as route currency, checking visible combat state before relying on memory, and selecting the narrow skill that matches the current decision boundary. The current run uses an overlay for route, deck, relic, potion, boss, and unresolved weakness notes. The current floor uses a shorter overlay for combat- or room-local details and should be replaced or pruned at safe boundaries.

The evolved player can search memory, append bounded run notes, replace floor memory, prune session memory, and write temporary session skills. It cannot directly write trusted long-term game-model memory during play. Post-run evolution owns long-term consolidation so that transient observations do not become durable knowledge without evidence.

C. Reflection, Staging, and Promotion

C.1. Run Reflection Schema

Run reflection uses schema run-reflection. The reflector reads run artifacts such as `metrics.json`, `trajectory.jsonl`, and `final_state.json`, then emits an outcome, failure classification, key mistakes, skill-change proposals, tool candidates, retrieval-query patterns, and optional snapshot replay cases. Each skill-change proposal includes an action type (`create`, `rewrite`, `merge`, `split`, `deprecate`, or `delete`), target path, category, decision boundary, insufficiency of existing skills, risk-benefit model, evidence spans, target layer, priority, risk, and validation plan.

Reflection is treated as candidate generation rather than truth. The schema is deliberately evidence-heavy because a plausible post-hoc explanation can still be a one-run accident. Candidate lessons must therefore remain linked to their source run, decision boundary, and validation plan.

C.2. Candidate Overlay

The Evolver stages admitted changes under the run's `evolution/candidate_overlay/` before any long-term write. Skill candidates are staged as markdown skills, generated extension tools are staged under the harness-owned generated-tool path, and memory additions are checked against the memory budget. Long-term skill, memory, or tool-registry writes are skipped when a proposal is too broad, duplicates existing guidance, lacks evidence, encodes a brittle one-run rule, conflicts with the current harness, or merely increases prompt length.

The evolution manifest records promoted, staged, rejected, pending, and skipped candidates. This manifest is part of the audit trail: a later run can inspect not only what changed, but also which plausible changes were rejected or blocked by validation.

C.3. Snapshot Replay Gate

When save/load snapshots are available, high-value candidates can be validated through replay. A replay case names a snapshot key, the skills under test, a replay mode, a safety turn cap, and concrete success criteria. The default replay mode is terminal: hitting the safety cap is not counted as a pass. Restore by itself is only a smoke check that the anchor exists and can be loaded; promotion requires replay evidence that the candidate supports a better or safer trajectory.

Snapshot keys are act-local, for example `pre:0:17` or `post:0:17`. Because restore mutates the live game, ordinary evolved gameplay is instructed not to restore snapshots. Replay validation is controller-owned and records availability, restore results, replay outputs, and promotion decisions in validation files and the evolution manifest.

D. Artifacts and Reproducibility

D.1. Run Artifacts

Each run writes a compact set of artifacts for analysis: `agent_events.jsonl` contains raw runner events aggregated across continuations; `agent_events.<NN>.jsonl` stores per-continuation events; `trajectory.jsonl` extracts a compact floor/action/state timeline; `viewer.html` provides a local timeline for messages, tool calls, actions, states, tokens, and costs; `initial_state.json` and `final_state.json` record captured game states; `state_status.json` records terminal or non-terminal status; `metrics.json` records outcome, reached floor, token usage, tool calls, and related run metrics; and `run_control.json` records continuation-loop stop reason and iteration summaries. Evolved runs additionally write floor reflections, session overlay memory or skills, `reflection.json`, `evolution_manifest.json`, and, when applicable, replay validation outputs.

The baseline runner uses a continuation loop. After each model stop, it captures the live STS2 state, checks whether the run is terminal, and resumes the same runner session if the game is still in progress and the continuation, elapsed-time, and cache-read budgets have not been exhausted. This prevents a non-terminal model stop from being misread as a completed game.

D.2. Evaluation Accounting

The main paper reports reached floor, outcome, new tokens per reached floor, cache-inclusive tokens per reached floor, and tool calls per reached floor. New tokens include input, output, and reasoning tokens, while cache-inclusive totals retain cache-read cost for systems accounting. The pilot result is therefore interpreted as a progress-cost tradeoff rather than a standalone win claim: the evolved profile reached the victory floor from the matched initial state, but it also spent more new tokens and more tool calls per floor.

For the planned repeated study, each pair should record the initial snapshot and initial state, run the baseline profile to death or victory, restore the same snapshot, run the evolved profile to death or victory, and then report mean and standard deviation across pairs. Divergence after restore is allowed because actions can branch game randomness; the controlled variables are the initial state, model identity, and runner profile.

D.3. Reproducibility Boundary

The released artifact should include prompts, runner configuration, MCP server configuration, skill files, memory files, trajectory and metric extraction scripts, run logs, reflection outputs, evolution manifests, and derived tables or figures. It should not redistribute copyrighted game assets or live save-state contents that depend on a local STS2 installation. The practical setup requires a local STS2 install with the gameplay agent mod enabled, environment variables for the OpenAI-compatible model endpoint, and the configured MCP servers for gameplay, snapshot replay, and typed local lookup.

E. Prompt Templates

Baseline Player Prompt

```
# Slay the Spire 2 Player

You are playing Slay the Spire 2 as Ironclad. Your goal is to win the game.

## Tools

- Use `sts2_*` tools to read game state and take actions.
- Do not use shell, file editing, web search, web fetch, or skills.
- Do not ask the user what to do. Make the best move yourself.
- Do not call `get_raw_game_state` unless a legal action failed and the
  compact state is not enough to recover.

## Play Loop

1. Start by calling `health_check`.
2. Before each action, call `get_game_state` and `get_available_actions`.
   Only choose actions and indices from the latest response.
3. At the main menu or character select, always choose Ironclad.
4. If an action fails, refresh state and retry once. After two consecutive
   failures on the same action, skip it and move on.
5. Treat `REST` with empty `available_actions` as a loading transition:
   wait and re-read state until legal actions return.
6. Continue until the run reaches a terminal state.

## Strategy

### Combat

Evaluate in this order each turn:
1. Lethal check: can current hand kill the enemy? If yes, do it.
2. Survival check: does enemy intent kill you next turn? If yes, block first.
3. Read enemy intent: check damage numbers, buffs, and debuffs before acting.
4. Kill fast-scaling enemies first.
5. Card order matters: play Strength/buff cards before attacks; play
   block-buff cards before block cards.
6. Apply Vulnerable before big hits: Bash or Thunderclap first, then attacks.
7. Save AoE for multi-enemy rooms.
8. Use potions in hard fights. Do not hoard.

### Deck Building

- Keep the deck at 15-25 cards. Quality over quantity.
- Remove Strikes first, then basic Defends at shops. Curse removal is highest
  priority.
- Commit to one archetype early when possible. Do not mix incompatible
  archetypes.
- Zero-cost cards are high tempo.
- Upgrade keystone cards before generic damage or block cards.
- Avoid 2-3 energy cards unless you have energy generation.
- Ancient boons at act start can define the run; transform and remove options
  are often better than raw stat boosts.

### Path & Map

Look at the full map before choosing the first path. Plan around rest sites,
elites, shops, and question marks.

Elites:
- HP > 70%: fight them; relics are essential.
```

```
- HP 40-70%: fight only with potions or a strong deck.
- HP < 40%: skip. After elites, prefer paths with a rest site.

Rest sites:
- HP < 50%: heal.
- HP >= 50%: upgrade a keystone card.
- Before a boss where you expect to need full HP: rest regardless.

Shops:
- Priority order: card removal, high-value relics, missing key cards, potions.

Question mark rooms:
- Events that remove cards are valuable.
- Transform events on Strikes/Defends are usually worth taking.
- Read risky events carefully; skip if the downside outweighs the upside.

Per act:
- Act 1: establish deck foundation and archetype.
- Act 2: test and reinforce scaling.
- Act 3: confirm win condition; enter boss with HP > 50% if possible.

## Final Answer

Do not give a progress-only final answer while legal game actions remain.
When the run ends, return a short factual summary: outcome, floor reached,
character, ascension, and the main reason the run ended.
```

Evolved Player Prompt

```
# Slay the Spire 2 Evolved Player

Play Slay the Spire 2 as Ironclad. Win the game.

## Loop

- Call `health_check` once. Before every game action call `get_game_state`
  and `get_available_actions`.
- Use only latest legal actions/indices; after an action failure, refresh and
  retry once.
- At main menu or character select, choose Ironclad.
- Continue until victory, death, game over, or a real unrecoverable tool
  failure.

## Harness Use

- Use skills, memory, and `sts2_ext_*` as decision support.
- At a meaningful decision boundary, load the narrowest relevant skill from
  the index.
- Use `sts2_ext_*` for fact lookup when it could change the move.
- Before `choose_map_node`, load `choose-map-path` and compare route segments,
  not only the next node.
- Treat high-gold shop leverage as a route objective; prefer a visible shop
  over speculative elite value unless clearly worse.
- At Neow/initial Ancient-relic offers, load `choose-ancient-relic` and
  compare candidates with `sts2_ext_search_relics`.
- Use `sts2_sl_list_snapshots()` for replay/checkpoint awareness and
  `sts2_sl_get_snapshot(key)` only for inspection.
- Do not call `sts2_sl_restore_snapshot(key)` in ordinary gameplay; replay
  validation is controller-owned.
- Use `write_session_skill(...)` for nontrivial reusable current-run rules;
  use memory for one-off facts.
```

```
## Memory

- `STS2 Game Model` is long-term context, not a fixed rulebook.
- Combat-local details live only in floor memory while they are useful.
- At combat, reward, event, and floor boundaries, replace or prune stale floor
  memory.
- Use `memory_replace_floor(summary)` for short current context.
- Use `memory_append_run(topic, note)` for route, deck, relic, potion, boss,
  or run goals that should survive floors.
- Use `memory_prune_session(keep_run=True, keep_floor=False)` when floor
  detail is no longer useful.
- Do not write trusted long-term `game_model.md`; the Evolver handles
  promotion.

## Guardrails

- Do not ask the user what to do.
- Do not use non-game tools or external browsing.
- Do not call `get_raw_game_state` unless legal action recovery needs it.
- Final answer only after run end: outcome, floor, character, ascension,
  reason.
```

Run Reflection Prompt

```
You are a Slay the Spire 2 expert analyst. Review this agent run and produce
a structured reflection.

## Run Summary
- Run ID: {run_id}
- Character: {character}, Ascension {ascension}
- Outcome: {outcome} at floor {floor}
- Total tool calls: {tool_calls}
- Final note: {final_text}

## Final Deck
{deck}

## Relics
{relics}

## Action Trajectory (compact)
{trajectory_summary}

---

Analyse the run and respond with ONLY a JSON object matching this schema
(no markdown, no explanation):

{
  "schema_version": "run-reflection-v2",
  "run_id": "string",
  "outcome": "death | victory | unknown",
  "failure_classification": "combat | map | card_reward | relic | unknown",
  "death_cause": "string",
  "floor_reached": integer,
  "evidence": ["specific trajectory or final-state facts"],
  "action_ordering_mistakes": [
    {
      "turn_or_floor": "where the ordering issue happened",
      "issue": "ordering error",
      "better_sequence": ["ordered checks or actions"],
      "evidence": ["supporting facts"],
```

```
      "priority": "low | medium | high"
    }
  ],
  "key_mistakes": ["3-6 specific mistakes"],
  "skill_change_proposals": [
    {
      "action": "create | rewrite | merge | split | deprecate | delete",
      "proposed_path": "routing/choose-event-node",
      "description": "when this skill should be loaded",
      "category": "combat | deckbuilding | routing | ops",
      "purpose": "how this changes attention without forcing a fixed answer",
      "load_when": ["specific trigger conditions"],
      "state_signals": ["state signals to inspect"],
      "why_existing_skills_are_insufficient": "short reason",
      "required_rag_queries": ["typed lookup queries"],
      "tool_hooks": ["sts2_ext_* support tools"],
      "decision_frame": ["flexible comparisons"],
      "cautions": ["ways this can be over-applied"],
      "target_layer": "skills",
      "priority": "low | medium | high",
      "risk": "low | medium | high",
      "evidence": ["supporting fact"],
      "validation_plan": "SL snapshot or future-run validation plan"
    }
  ],
  "sl_replay_cases": [
    {
      "snapshot_key": "pre:0:17",
      "purpose": "candidate behavior to validate",
      "skills_under_test": ["combat/enemy-specific-tactics"],
      "success_criteria": ["observable outcome or decision behavior"],
      "replay_mode": "terminal",
      "max_turns": 500
    }
  ],
  "candidate_skills": [],
  "candidate_tool_improvements": [
    {
      "tool": "tool name",
      "proposed_tool_name": "optional sts2_ext_* tool name",
      "issue": "what was wrong or missing",
      "suggestion": "concrete improvement",
      "tool_contract": "optional factual input/output contract",
      "target_layer": "tools",
      "priority": "low | medium | high",
      "risk": "low | medium | high",
      "evidence": ["supporting fact"]
    }
  ],
  "rag_query_patterns": [
    {
      "query": "query the agent should have made",
      "target_layer": "knowledge",
      "priority": "low | medium | high",
      "risk": "low"
    }
  ]
}

Focus on:
1. What specific decision led to death, if applicable.
2. What strategic knowledge would have prevented the mistakes.
3. What reusable skill changes should update the seed skill library.
```

```
4. What improvements are needed to the small existing tool surface.
5. What knowledge-base queries would have been useful mid-run.
6. Combat action ordering: audit draw-before-spend, setup-before-survival,
   block-before-end-turn, potion timing, target/action-removal ordering, and
   card-index sequencing mistakes.
7. SL replay validation: add `sl_replay_cases` for high-value snapshot-backed
   counterfactual checks, especially the failure floor.
8. Route-tool self-evolution: if the map view hid future shops/campfires, add
   a factual `sts2_ext_view_full_map` candidate.

Constraints:
- Return skill changes in `skill_change_proposals`; leave `candidate_skills`
  as an empty backward-compatibility field.
- Always fill `action_ordering_mistakes`; use an empty list only when the
  trajectory shows no meaningful ordering issue.
- Always fill `sl_replay_cases` when outcome is death and the failure floor is
  known.
- Do not create many tiny skills.
- Do not propose fixed card-pick rules.
- Do not propose narrow hard-coded advisor tools.
- Put decision procedures into skills and memory; use the tool layer only for
  local knowledge lookup unless a future tool can be validated rigorously.
```

Replay Validation Prompt

```
SL validation replay from snapshot {snapshot_key}.
Use the candidate overlay skills under test before making the risky decision.

Skills under test: {skills_under_test}

Success criteria: {success_criteria}

Play legal actions through the restored state until victory or death.
The {max_turns}-turn cap is only a safety limit, not a success condition.

Validation policy:
- `sts2_sl_restore_snapshot(key)` is controller-owned and should not be called
  during ordinary gameplay.
- A successful restore is only a smoke check.
- Promotion requires replay evidence that the candidate supports a better or
  safer trajectory.
- In terminal replay mode, passing requires a terminal victory; stopping at the
  safety cap is recorded as incomplete, not successful.
- Record replay outcome, floor, HP, victory flag, stop reason, runner exit
  code, candidate overlay files, and validation reason in the replay results.
```