DO LLM AGENTS STILL NEED TO BE TOLD HOW TO SELF-IMPROVE?

Anonymous authors
Paper under double-blind review

ABSTRACT

LLM agents increasingly drive their own learning. Self-improvement methods delegate various components of model optimization to the models themselves. As models become more capable, this motivates a basic question: do these increasingly capable models still need us to specify how they should self-improve? In this paper we study an autonomous paradigm in which, given basic primitives, resources, and a general goal for self-improvement, agents construct their own self-learning algorithm. Using the setting of gameplay as a testbed, a frozen LLM agent is given a compute budget and tools for collecting experience and updating text documents, and allowed to freely experiment without an externally imposed loop or access to real-world data. We find that sufficiently capable models do self-improve under these conditions, demonstrating strategies that align with classic cognitive frameworks for self-regulated learning and constructs from machine learning literature on self-generated curricula. Furthermore, we study the extent to which the learned text artifacts transfer to external benchmarks, observing improvement across different game horizons, mechanics, and 3D navigation tasks.

1 INTRODUCTION

As LLM agent capabilities improve, we increasingly delegate parts of the machine learning process to the learners themselves. Self-improvement aims to reduce the role of human effort and bias in generating experience and translating it to improved performance. Various works leverage models to generate supervised data, construct interactive environments, and even rewrite their own tools and harnesses (Zhang et al., 2026a; Liu et al., 2026; Zelikman et al., 2022; Wang et al., 2022b). Asymmetric self-play offers a particularly compelling approach in which models learn by simultaneously generating and solving tasks (Sukhbaatar et al., 2018). In parallel with efforts to develop better methods, there are growing concerns about the need for safeguards against misaligned self-improvement. Both directions motivate a basic question:

Do capable models still need us to specify how they should self-improve?

We study the natural behavior of an LLM agent when provided with the primitive tools and resources for self-improvement, but without being wrapped in an explicit self-improvement algorithm. Rather than designing a loop for experience generation and learner updates, we allow the agent to compose actions into a self-learning procedure. The agent does not have access to real-world tasks or validation scores. It must experiment with different ways of generating tasks, reflecting on failures, and assessing progress in a constrained sandbox without being directed how to do so. Our object of study is thus the construction of a learning procedure from basic tools. Our setting is important for understanding both capabilities, where models may no longer require prescribed self-optimization algorithms, and safety, where the absence of a specified loop does not necessarily prevent self-improvement.

We find that *sufficiently capable models successfully self-organize learning procedures and self-improve on held-out tasks.* We examine this phenomenon in the setting of gameplay with limited information. A *player* agent plays turn-based games with undisclosed mechanics under strict compute and game-horizon budgets. This task requires diverse reasoning skills including efficient exploration, information acquisition, multi-step planning, and world modeling. An instance of that same agent serves as the *teacher*. The teacher is given a general goal to improve the player through text (e.g., editing the prompt, creating skill files), a compute budget, and a set of tools for observing the player

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

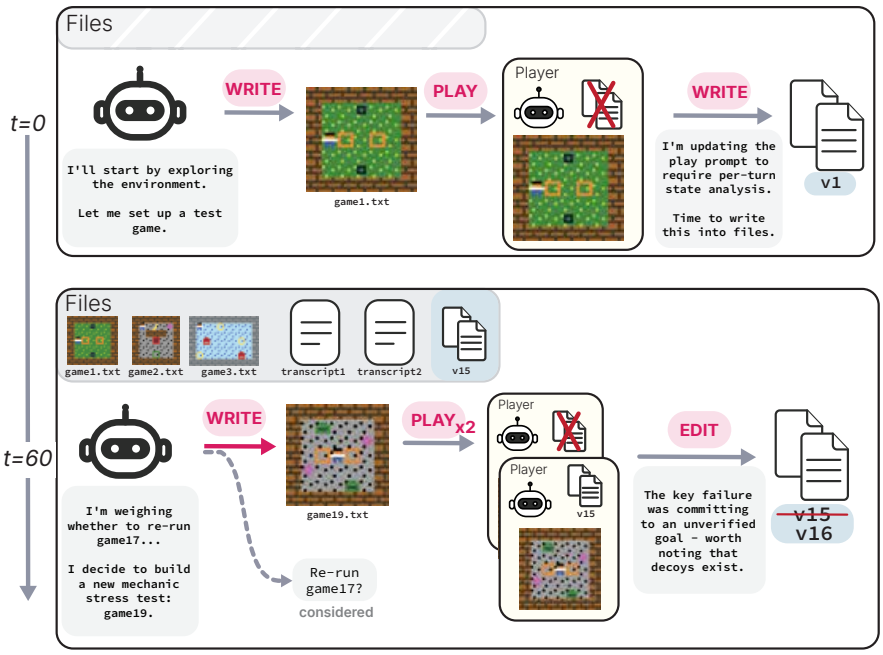

Figure 1: **Do capable models specify their own self-improvement?** We give a frozen LLM (here, Opus 5) the goal of improving itself at gameplay, a toolset {READ, WRITE, BASH, PLAY}, and a workspace. We do not specify an optimization loop or how the teacher should use these tools; instead, it organizes its own learning procedure. At the start of optimization ($t = 0$) the teacher uses its tools to build a test game, spawn a player session to attempt the game, analyze the result, and update the player context (e.g., skill files). Later ($t = 60$), the teacher writes an adversarial game and observes the player attempt it with and without the optimized context. The teacher then uses the observed failure mode to make another context update.

attempt self-generated games and editing player context. We do not instruct the teacher when to use the tools; the teacher freely experiments with the player.

Across the set of effective self-learners, models from *different* families employ *similar* recurring, recognizable learning strategies, such as generating games with calibrated difficulty, running controlled A/B comparisons, and developing validation suites to measure progress. These strategies align to behaviors described by social-cognitive frameworks for self-regulated learning (Zimmerman, 2000; 2002). We also find that agents implement learnable task-selection, rooted in developmental psychology and implemented in machine learning literature (Vygotsky, 1978; Wang et al., 2022a; Rutherford et al., 2024). Furthermore, the self-constructed learning procedures generalize: Learned text files transfer across multiple benchmarks that require reasoning over different game horizons, engines, and representations. We study how providing information on evaluation settings to the teacher, via context or environment constraints, shapes this downstream generalization.

Our main findings are:

1. Given just primitive tools for self-improvement, a sufficiently capable agent constructs an effective learning procedure that improves its own performance on held-out gameplay.
2. Models from different families implement recognizable strategies that align to self-regulated learning frameworks, and machine learning literature on self-generated curricula.
3. The learned text generalizes across game benchmarks and representations. Evaluation-setting awareness biases the teacher towards relevant tasks, however can encourage overfitting.

2 Related Work

For an extended view of related work, see Appendix A. We categorize self-improvement methods based on which components of the learning process remain externally specified, how learning takes place, and the scope of generalization that is studied.

Asymmetric self-play. In asymmetric self-play a teacher generates challenging problems for a student to solve, resulting in an emergent, adaptive curriculum (Sukhbaatar et al., 2018). Such methods have found success in post-training reasoning LLMs (Zhao et al., 2025; Huang et al., 2026; Poesia et al., 2024; Dong & Ma, 2025; Liu et al., 2025; Zhang et al., 2025b; Teodorescu et al., 2023). In these methods the teacher self-generates the source of experience by synthesizing student problems, while learning proceeds in weight-space. The optimization procedure (e.g., update schedules, teacher rewards, the number of sampled tasks in a batch) remains externally specified. Such methods can also suffer from diversity or performance collapse (Chae et al., 2025), motivating additional mechanisms for assessing the learning value of generated problems. Various methods use judges (Bailey et al., 2026) or ground the agent in real-data hard questions as goalposts (Jana et al., 2026). These mechanisms help stabilize self-play but introduce additional externally specified choices about what constitutes a useful learning experience. Collapse can also be addressed by rewarding the teacher based on whether its generated problems induce student improvement on external hard problems (Sundaram et al., 2026). In this case, the feedback is externally obtained while still relying on self-generated data to drive learning. Recent works have the teacher generate interactive environments rather than question-answer tasks (Liu et al., 2026) but retain externally specified optimization and task-selection procedures.

Context and harness optimization. A separate line of work considers self-improvement through updates to the model's context or agent harness (Suzgun et al., 2026; Wang et al., 2025; Hu et al., 2025; Zhang et al., 2025a). GEPA uses natural-language reflection over execution trajectories to optimize model prompts via evolutionary search (Agrawal et al., 2026), while ACE maintains an evolving playbook that accumulates strategies from previous interactions (Zhang et al., 2026c). These works still leverage an externally specified task distribution and pre-specified context optimization procedure. MCE partly relaxes the latter by co-evolving the context with a context-engineering skill (Ye et al., 2026). Meta-Harness expands the object of optimization further to executable harness code, allowing the model to change how context is managed, including what information is stored and retrieved (Lee et al., 2026). The optimization still uses externally provided tasks, and candidate harnesses are evaluated with an externally specified objective. Recent work on self-modifying agents further expands the components under the model's control. The Darwin Gödel Machine iteratively modifies its own agent code and evaluates the resulting agents on external benchmarks (Zhang et al., 2026a), while Hyperagents make the meta-level procedure used to generate future improvements itself editable (Zhang et al., 2026b). These approaches allow increasing parts of the self-improvement mechanism to change, while still operating within an externally specified task and evaluation setting.

Self-constructed learning procedures. Xue & Yang (2026) introduce Open-Ended Optimization, where the objective, available interactions, resource budget, data boundary and evaluation are fixed, but the model organizes the improvement process itself. Performance is evaluated on held-out examples from the same benchmark setting used for optimization. Closest to our setting, in concurrent work, Wu et al. (2026) introduce ASPIRE, a benchmark for self-evolution from vague capability goals. Agents (sometimes a stronger teacher model optimizing a smaller student) improve weights or harnesses with access to external data and, in some settings, aggregate scores from the hidden evaluation. Performance is then measured on hidden items from the same goal. In contrast, we study a model improving itself from self-generated experience alone, and evaluate transfer to different environments.

3 Preliminaries: Gameplay with limited information.

Our goal is to study agent self-improvement in interactive settings requiring multi-step reasoning. As a testbed, we use the setting of gameplay with undisclosed mechanics. The agent plays a turn-based game with limited knowledge of the rules and objectives. To succeed it must efficiently acquire information, plan, and execute within the given horizon. This setting exercises diverse capabilities, including adaptive efficiency, goal discovery, planning, and world modeling (Foundation, 2026).

We consider a partially observable Markov decision process setting. For each game g, let $s_t \in \mathcal{S}$ denote the state at turn t, $o_t \in \mathcal{O}$ the observation received by the player, and $a_t \in \mathcal{A}$ the player's action. The game engine determines the transition dynamics $s_{t+1} \sim P_g(\cdot \mid s_t, a_t)$ and corresponding observation $o_{t+1} \sim O_g(\cdot \mid s_{t+1})$. The reward is binary, $r_t = R_g(s_t, a_t, s_{t+1}) \in \{0, 1\}$; the player receives reward 1 when reaching a winning state and 0 otherwise. The episode terminates upon

success or after at most H turns. The player does not have access to P_g or R_g and infers the game mechanics through interaction. It acts based on its interaction history $h_t = (o_1, a_1, \ldots, o_t)$ and context c, with $a_t \sim \pi_\theta^S(\cdot \mid h_t, c)$.

4 A CONTROLLED SELF-IMPROVEMENT SETTING

Do LLM agents, given primitive tools for text editing and experience generation, compose those tools into a useful self-optimization procedure? And can this be achieved in a *data-free* setting with no access to externally-supplied environments? We study these questions in a controlled environment: A frozen model (the *teacher*) is given a budget, a prompt, and a set of tools, and allowed to experiment freely. The teacher's given goal is to optimize text (e.g., a prompt, skill files) that improves a player's performance on unseen games, where the player is another instance of the same model. We measure the success of an agent's self-organized learning through performance on held-out game benchmarks.

Algorithm 1 Self-organized self-improvement

Require: Budget B, teacher context I
$S \leftarrow \text{INITSANDBOX}(c_0, I, B)$
$\mathcal{A}_T \leftarrow \{\text{READ}, \text{WRITE}, \text{BASH}, \text{GAMEPLAY}\}$
while $B > 0$ **do**
  $a \sim \pi_{teacher}(\cdot \mid S),\ a \in \mathcal{A}_T$
  $S \leftarrow \text{EXECUTE}(S, a)$
end while
return player context c_{final}

Our setting has similarities to asymmetric self-play, in which a teacher generates tasks for a student to solve (Sukhbaatar et al., 2018). When applied to LLMs the same model often takes on both roles (Zhao et al., 2025). These methods prescribe procedures for sampling tasks and rollouts, and updating the student and teacher. We adopt the notion of an agent self-generating tasks, but do not guide how the teacher must use its budget and tools, and optimize in text-space rather than weight-space.

4.1 TEACHER-PLAYER SETUP

Let π denote an LLM with frozen weights θ. The LLM is instantiated as a **player agent** π_{player} that has context $c = (c_P, c_L)$ where c_P is the player prompt, and c_L is auxiliary text files (e.g., skill files). The player policy induced by c is then $\pi_{player}(a_t \mid h_t, c)$ where h_t denotes the game trajectory $(o_1, a_1, ..., o_t)$ up to turn t. Another instance of the same LLM, the **teacher agent** $\pi_{teacher}$, modifies c with text edits. We define an update as *a single edit tool-call to one file in* c. Let c_0 be the initial context state (e.g., a minimal prompt) and c_{final} be the state of c after the final teacher update.

Context optimization allows us to study closed-source models, but also has useful properties. Gradient-based learning is in continuous space and tied to predefined rewards. In text optimization, the teacher can reflect on player rollouts and dynamically decide to generate tasks, experience, or knowledge updates. Those updates being in text preserves a readable explanation for changes in player behavior.

4.2 TEACHER ENVIRONMENT

The teacher is an agent in a persistent, sandboxed harness session (e.g., Claude Code, Pi). The harness is minimal and controlled. We disable the default prompt, toolset, automatic memory, web search, subagents and other features (Appendix C.1). The teacher has access only to the sandbox files and features that we explicitly provide: context, a specified toolset, and a budget.

Tools. The teacher is given the following tools: **Read/Write** for sandbox files, **Bash** for terminal commands (subject to sandbox constraints), and a **Gameplay tool**. The Gameplay tool allows the teacher to observe a separately-instantiated player play a game with only teacher-granted context. The teacher calls the tool with a game source file and, optionally, a path to the designated directory holding the player text artifacts. The tool executes the game in a fresh, external player session, and returns to the teacher the player score, transcript, and the teacher's budget state (described below).

In our main experiments the Gameplay tool plays games in *PuzzleScript*, a domain-specific language for 2D grid-based puzzle games. PuzzleScript admits a diverse set of games that share basic structures and action spaces, enabling a controlled setting for studying generalization. The teacher issues games as PuzzleScript text files; the Gameplay tool rejects games that do not compile or where the win state is verifiably unreachable. We ablate the PuzzleScript constraint in experiments that instead play Python games, and study generalization to non-PuzzleScript games in Section 6.1.

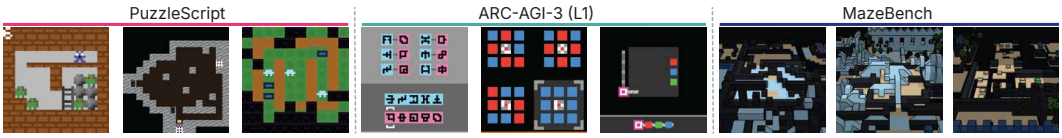

Figure 2: **Examples of evaluation benchmarks.** Sample games from our transfer benchmark, ordered in increasing distance from our canonical self-optimization setting: (1) PUZZLESCRIPT (same game engine, different horizon splits), (2) ARC-AGI-3 (different engine and representation), (3) MAZEBENCH (3D navigation and mechanics in isometric ASCII projections).

These tools can serve as a mechanism for giving the teacher information. For instance, we can allow the teacher to specify the game horizon (turn budget) in the Gameplay tool, or impose such a budget ourselves — effectively communicating the evaluation game horizon via the constraint.

Context. The teacher's system prompt specifies a *main goal*: to improve a player's performance on single-player games by writing text artifacts, including if desired the player prompt. We also specify a *generalizability goal*: optimized text should extend to unseen games with different tasks or mechanics. We do not provide guidance on how to structure the text artifacts, instead allowing the teacher to freely create and modify player text files. When granted the PuzzleScript Gameplay tool, the teacher also receives official PuzzleScript documentation. Full prompts are in Appendix C.4.

Budget. We give the teacher a two-part budget. The *update budget* is the number of updates that may be applied to the player context c, designated as a directory in the teacher sandbox. Each edit tool-call to c is one update. The *gameplay budget* is the number of gameplays that the teacher may observe. The teacher session ends when either budget is exhausted. If the teacher ends its action sequence without consuming either budget fully, we prompt it to continue. In practice the update budget is typically exhausted first. These budgets force the teacher to account for resource consumption.

4.3 MEASURING SELF-IMPROVEMENT THROUGH GENERALIZATION

We assess if an agent has structured its learning well in this setting via transfer to held-out benchmarks. Given player context c, we evaluate that context by how well the induced player policy performs on an unseen test set $\mathcal{D}_{test}$. For each game, we instantiate a fresh player session with c and rollout the game at a fixed game horizon H (turn budget) and output token budget. The game ends at turn $T \leq H$ when (1) the player achieves the objective, (2) the turn budget is exhausted, or (3) the player's output tokens (summed over turns so far) exceeds the game token budget. The reward on $\mathcal{D}_{test}$ is

$$J(c) = \frac{1}{|\mathcal{D}_{\text{test}}|} \sum_{g \in \mathcal{D}_{\text{test}}} \left[\mathbb{E}_{\pi_{player}(\cdot|h_t,c)} \left[\sum_{t=1}^{T} r_t \right] \right]. \quad (1)$$

In practice we estimate the inner expectation by averaging the rewards of independent Monte-Carlo rollouts (Table 2). The policy is evaluated across a distribution of games rather than initial states of a single game environment. We measure self-improvement as the difference in player performance with and without the teacher-optimized context: $J(c_{final}) - J(c_0) > 0$.

5 WHAT ARE THE STRATEGIES OF SUCCESSFUL SELF-IMPROVERS?

5.1 EXPERIMENT SETUP.

We assess six models across three families: `Claude Opus-{4.6, 5}`, `GPT {5.6-Sol, 6-Astra}`, and `GLM-{4.7, 5}`. In all experiments both the teacher and player use low reasoning effort and per-turn 20k token budgets. We run three independent teacher sessions per model with an update budget of 20, gameplay budget of 150, and toolset of {`Read/Write, Bash, Gameplay tool`}. In this section the Gameplay tool plays PuzzleScript games with a 20-turn horizon and 25k full-game token budget. We ablate these constraints in Section 6.1. For evaluation we use the following external benchmarks (construction described in Appendix B).

PUZZLESCRIPT-T20. We draw a subset of 20 short-horizon game levels (optimal solution length <12) from a pool of human-authored PuzzleScript games (Earle et al., 2025). We evaluate with a

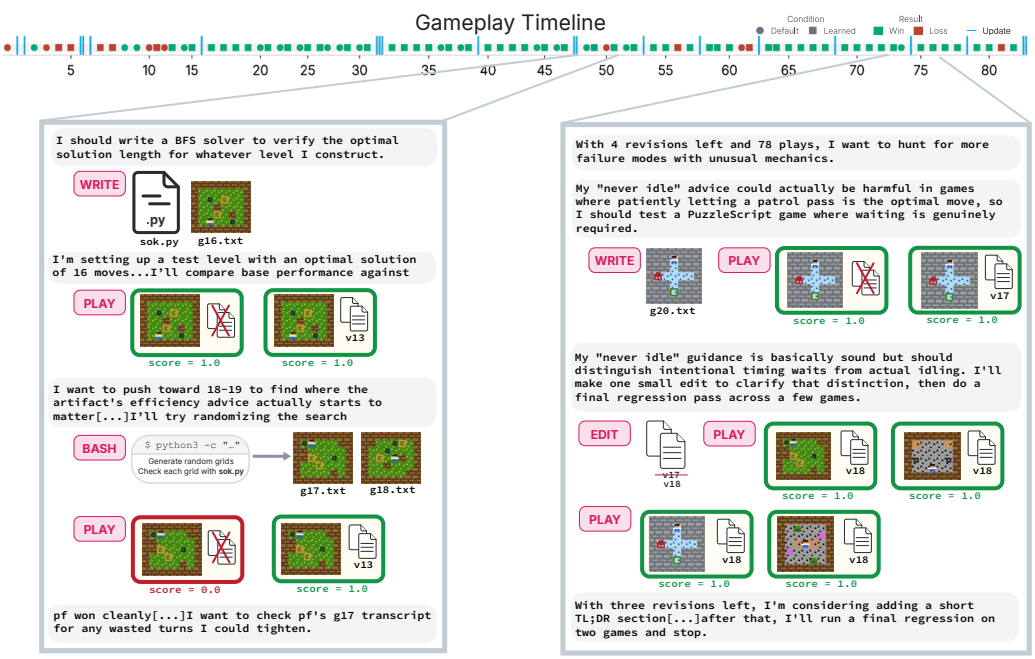

Figure 3: **Excerpts from a `Claude Opus-5` teacher session with high alignment to self-regulated learning subprocesses.** In the first example (*task-strategies*) the teacher writes a random level generator and BFS solver to filter generated games for difficulty and employs A/B testing to identify where its optimized text artifacts are useful. In the second (*learning goal orientation*) the teacher designs an adversarial game to test its advice and runs validation tests across earlier games to identify regressions.

20-turn horizon and 25k token-budget, thus sharing constraints with the self-learning setting. We next test transfer to benchmarks that share progressively fewer attributes.

PUZZLESCRIPT-T30. We draw 20 longer-horizon game levels (solution length 13-22) from the same base pool as PUZZLESCRIPT-T20, and evaluate with a 30-turn horizon and 31k token-budget. This test set retains the shared game engine but tests generalization to longer game horizons.

ARC-AGI-3 (L1). We test transfer to different 2D game engines by evaluating on Level 1 of games in ARC-AGI-3 (Foundation, 2026) (30-turn horizon, 31k token-budget).

MAZEBENCH. MazeBench is a 3D open world in which an agent navigates rooms with obstacle challenges to retrieve gems (Pappas & Pappas, 2026). We adopt the benchmark's room-completion metric and draw a random set of 16 rooms solvable within 50 turns. We evaluate with a 100-turn horizon and 75k token-budget. This benchmark tests generalization to longer horizons, 3D navigation via isometric-style ASCII projections, and 3D mechanics.

5.2 NEWER MODELS TEACH THEMSELVES BETTER

We first ask if there are models that do self-improve in our setting. Evaluation results on `PuzzleScript-t20` are in Figure 4. Across model families, the three more recently released and broadly capable generations improve over the default-context player baseline (+20% for `Claude Opus-5`, +10% for `GPT 6-Astra`, +6% for `GLM 5.3`). Critically however, this effect is not evident for all models; older generations show little or no improvement. This gap indicates a potential trajectory where self-organized self-learning is increasingly evident in successive model generations.

5.3 LLMS FOLLOW RECOGNIZABLE SELF-IMPROVEMENT STRATEGIES

Our empirical results indicate a performance gap between older and newer models, however do not explain which strategies drive effective self-learning. Here, we show that agent behaviors increasingly align to recognizable self-learning strategies across cognitive and machine-learning literature.

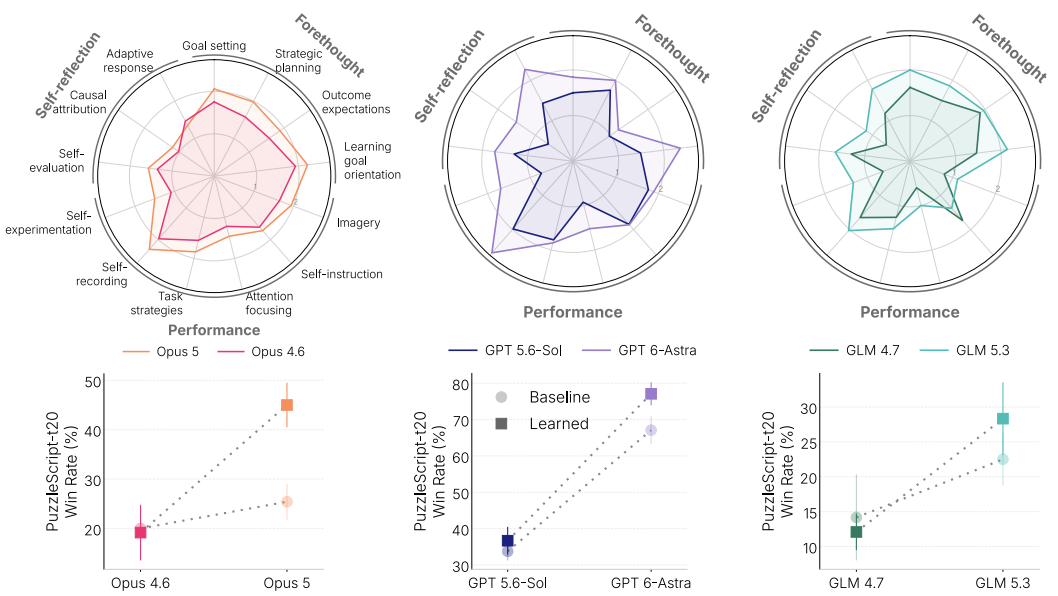

Figure 4: **Recent models self-improve and display human-recognizable learning strategies. Bottom:** Results on PUZZLESCRIPT-T20 for two models in each family. Newer generations self-improve over the baseline while older generations fail to do so. Error bars are ±1 SD over 3 seeds. **Top:** Alignment scores to 13 subprocesses of Zimmerman's self-regulated learning cycle, annotated by an LLM judge and averaged over 3 teacher sessions per model. Models that self-improve score higher across most subprocesses.

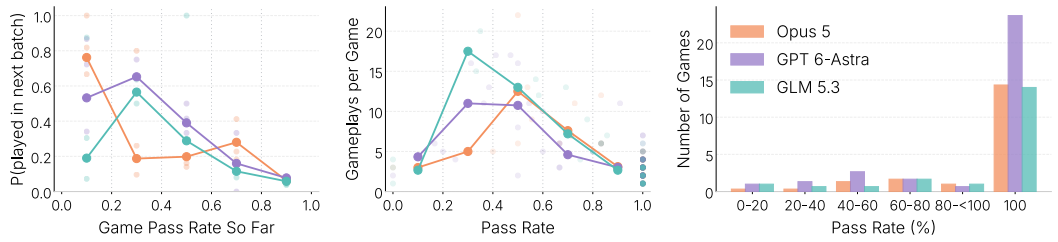

Figure 5: **Self-improving agents use learnability-based task selection**. The three models that effectively self-improve in our setting display similar task-selection statistics. Most generated games are easy but quickly discarded. Games with intermediate (20–60%) pass rates are significantly more likely to be replayed, leading to an inverted-U that resembles learnability objectives in asymmetric self-play and UED.

Self-Regulated Learning. We first draw upon Zimmerman's cyclical phases model, a classic social-cognitive framework for self-regulated learning (SRL) that describes how students adapt their learning strategies based on metacognition and self-reflection (Zimmerman, 2002). The framework organizes SRL into subprocesses grouped under *forethought* (goal setting and planning), *performance* (executing strategies and self-monitoring), and *self-reflection* (evaluating performance and guiding subsequent learning). We functionally adapt this framework into a behavioral rubric for 13 subprocesses, treating the teacher-player system as the unit of analysis. For each independent teacher session we use `GPT 6-Astra` to score the occurrence and quality of each SRL subprocess on a 0-3 scale. Scores on different seeds of the same model align, indicating stable behaviors across runs (Appendix C.5).

Models that effectively self-improve in our setting also show higher SRL alignment (Figure 4). This is not necessarily a causal explanation, but quantifies that better self-learners exhibit human-recognizable task-selection, evaluation, and reflection strategies. Figure 3 illustrates examples: the teacher uses A/B testing to evaluate gains from its text artifacts (*self-experimentation*), writes a procedural game generator with a BFS solver for difficulty filtering (*task strategies*), and designs games adversarial to its own notes to stress-test its advice (*learning goal orientation*). Further examples are in Appendix E.

Learnability. In Figure 5 we show task-selection statistics for the three models that self-improve. Across models, teachers allocate tasks similarly after observing player performance. The majority of generated games are easy (Figure 5c) but quickly discarded; gameplays concentrate in the group

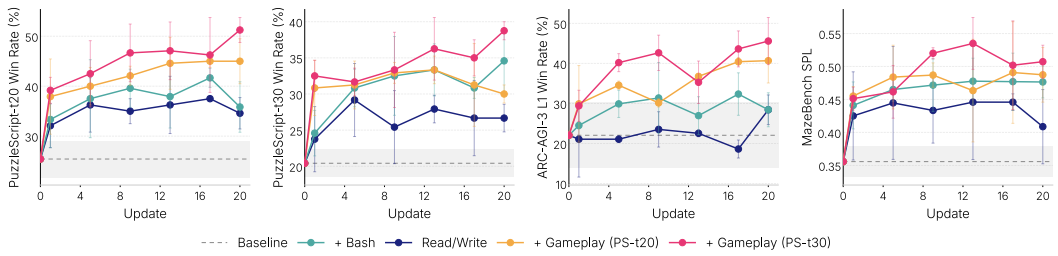

Figure 6: **Improvements from self-organized learning transfer across benchmarks**. Player performance is evaluated at intermediate context updates along the self-learning trajectory on PUZZLESCRIPT-T20, PUZZLESCRIPT-T30, ARC-AGI-L1 and MAZEBENCH. The context optimized under our canonical setting, *Gameplay (PS-t20)*, improves over the unoptimized baseline (dashed line) on PuzzleScript games with both 20- and 30-turn horizons. These gains transfer to different game engines and representations in ARC-AGI-L1 and MAZEBENCH. We also compare progressively richer teacher toolsets, starting from Read/Write, adding Bash, and then PuzzleScript-Gameplay tools. Error bars show ± 1 SD over three independent teacher seeds.

of lower-scoring games. Games that the player wins have a near-zero chance of being replayed next, whereas the replay probability for games with an empirical pass rate below 40% is 60-80% (Figure 5a) depending on the model. Figure 5b shows that most replays concentrate in games with a 20-60% pass rate across all rollouts. Notably, the same concentration is not consistent across models that fail to self-improve (Figure 11). The classic "inverted-U" that we observe strongly resembles the learnability task rewards used in asymmetric self-play and unsupervised environment design encouraging teacher-generated tasks with an intermediate pass rate (Huang et al., 2026; Rutherford et al., 2024). Learnability-based task selection in machine learning is itself rooted in the "Zone of Proximal Development" concept in developmental psychology (Vygotsky, 1978; Wang et al., 2022a). We find that LLM agents demonstrate similar learnable task selection without being steered to do so.

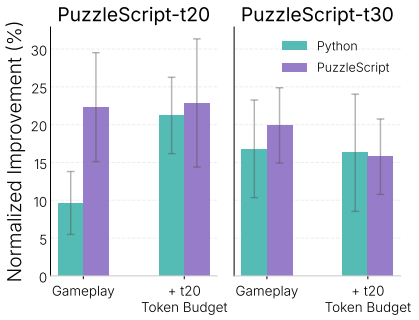

Figure 7: **PuzzleScript-specific gameplay is not essential for transfer.** We compare Python-Gameplay tool to the PuzzleScript-Gameplay tool with and without imposed token-budgets.

6 WHEN DOES SELF-ORGANIZED SELF-IMPROVEMENT GENERALIZE?

Section 5 established that a class of capable models self-improve with minimal guidance on a benchmark that shared key attributes with the self-learning setting. Now, using `Claude Opus-5`, we evaluate *generalization* from three perspectives: transfer across environments, under different information regimes, and across models.

6.1 TRANSFER ACROSS ENVIRONMENTS.

Figure 6 shows that learned context transfers to environments that differ increasingly from the self-optimization setting. We plot player performance at even intervals along the self-learning trajectory. At context update k we evaluate a fresh player session with c_k on our held-out benchmarks. The context optimized under 20-turn PuzzleScript gameplays (denoted *Gameplay (PS-t20)*) improves PUZZLESCRIPT-T20 performance by 21%, PUZZLESCRIPT-T30 by 19%, ARC-AGI-L1 by 21%, and MazeBench SPL by 0.15. This demonstrates transfer across game engines, grid sizes, core mechanics, 3D spatial navigation, longer-horizon reasoning, and state representations.

These trends also extend to different self-improvement settings. We run three comparisons, each with three independent teacher sessions at matched teacher budgets. (1) The teacher has only Read/Write tools (*Read-Write*), (2) with Bash access (*Read-Write-Bash*), and (3) with the Gameplay tool constrained to 30-turn PuzzleScript games. *Gameplay (t-30)* performs best across benchmarks, even outperforming *Gameplay (PS-t20)* on PUZZLESCRIPT-T20. *Read-Write* outperforms the baseline, indicating that some of the observed improvement in *Gameplay* settings stems from the

teacher's self-reflection on prior knowledge. On all benchmarks besides MAZEBENCH, *Read-Write* recovers less than 50% of the performance gain of *Gameplay* teachers, indicating benefit from generated experience. *Read-Write-Bash* outperforms *Read-Write*, but fails to recover the full gains of separate gameplay. Notably, in some *Read-Write-Bash* sessions the teacher still writes and runs game simulators in the sandbox to test its advice despite no mention of gameplay in its prompt (Appendix E). For a sufficiently strong teacher with Bash, an explicit Gameplay tool may be unnecessary.

We also evaluate allowing the teacher to issue Python games; Python gameplay recovers nearly all the performance gain of PuzzleScript gameplay in 3 out of 4 cases (Figure 7), suggesting that PuzzleScript-awareness is not strictly necessary for the teacher to learn transferable context.

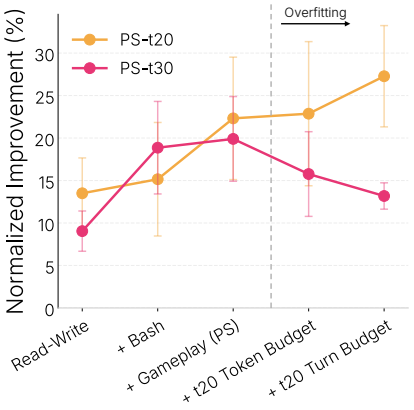

Figure 8: **Generalization with respect to progressively richer toolsets and eval setting awareness.**

6.2 INFORMATION SHAPES GENERALIZATION.

How does overfitting arise in our setting? We do not provide external tasks or validation feedback, however we communicate information about the evaluation setting via the teacher's tools. In Figure 8 we ablate the teacher's awareness of the PUZZLESCRIPT-T20 evaluation setting, and plot *approximation* (PUZZLESCRIPT-T20 performance) versus *generalization* (PUZZLESCRIPT-T30 performance). See Appendix C.2 for setup details. Performance rises as tools and constraints relevant to both benchmarks are added, such as a PuzzleScript-constrained Gameplay tool where the teacher selects the turn horizon. Imposing Gameplay turn- and token-budgets that match the t20 setting, however, improves PUZZLESCRIPT-T20 performance while PUZZLESCRIPT-T30 transfer falls. These results yield an intuitive takeaway: Increased awareness of the evaluation setting biases the teacher towards generating relevant experience, however can also encourage task-specific specialization.

6.3 TRANSFER BETWEEN MODELS.

Does self-optimized text contain general lessons that another agent might benefit from, or is that text overfit to the target agent's generated experience? We evaluate `Claude Opus-5` on PUZZLESCRIPT-T20 using c_{final} self-optimized by `GPT 6-Astra` and vice versa. The learned text transfers in both directions, even despite `GPT 6-Astra` having higher baseline performance (Table 1).

Player	Library source	Win rate (%)
Opus 5	None	25.4 $\pm$ 3.6
	GPT-6 Astra	**47.5** $\pm$ 4.5
GPT-6 Astra	None	67.1 $\pm$ 3.8
	Opus 5	**72.1** $\pm$ 4.0

Table 1: **Cross-model transfer of self-optimized context** on PuzzleScript-t20. Mean $\pm$ std over three seeds.

7 DISCUSSION AND LIMITATIONS

Our results align to the basic trend in deep learning that as models become more broadly capable, they require fewer human-imposed inductive biases. We study this phenomenon at the meta-optimization layer, showing that LLM agents self-improve with minimal guidance on how to do so, and demonstrate recognizable self-improvement strategies.

This analysis, however, does not capture more uninterpretable behaviors, nor fully disentangle whether self-regulatory strategies are instrumental to model self-improvement. Another open question is whether our observed trends extend to more open-ended domains where models have weaker priors. Games have an extensive literature on curriculum design and learnable task generation that models have likely ingested. The use of PuzzleScript may hint at real-world environments or bias the model, though our Python-game ablations show that PuzzleScript is not a critical factor for self-improvement. Nonetheless, games provide a rich setting with several controllable axes of variation, including game mechanics, game horizon and observation representations. We observe frontier models self-organizing their learning in this setting, with improvements transferring across these axes and even between models. Whether these trends extend to less well-specified domains remains an open question.

AI USE STATEMENT

Tasks with required disclosure. In this work, we used generative AI tools to assist in developing conceptual frameworks as part of our analysis in Section 5.3 (described further in Section C.5). Specifically we used GPT 6-Astra to assist in adapting a classic framework for self-regulated learning into a rubric for scoring the occurrence and quality of self-regulatory behaviors in model traces (Zimmerman, 2002). The authors reviewed the rubric in detail, and refined it to ensure alignment to the relevant literature.

We have not used generative AI tools to: Generate synthetic data sets, formulate mathematical claims, provide critical ingredients for proving mathematical claims, assist in the writing of proofs, propose or refine hypotheses, design or provide feedback on research methodology or experiments, implement methods, assist with translation, clean and reformat dataset, support qualitative and thematic data analysis, interpret results.

Other tasks: We used generative AI tools for creating and editing code that was used to run experiments; this code was tested for correctness by the authors. We also used generative AI tools for modifying plot formats, and providing feedback and suggestions on written material.

We take full responsibility for the final content of this work.

REPRODUCIBILITY STATEMENT

We put all details for reproducibility in the Appendix, particularly Sections B-C, including the details of our benchmark construction, experiment implementation, hyperparameters, and prompts. We also include supplementary materials necessary for reproducing our analyses. All of the datasets that we use in this paper are drawn from publicly available sources. Furthermore, we plan to release our code, which is entirely based on publicly available agentic harnesses.

REFERENCES

Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alex Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. GEPA: Reflective prompt evolution can outperform reinforcement learning. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=RQm2KQTM5r.

Luke Bailey, Kaiyue Wen, Kefan Dong, Tatsunori Hashimoto, and Tengyu Ma. Scaling self-play with self-guidance. *arXiv preprint arXiv:2604.20209*, 2026.

Justin Yang Chae, Md Tanvirul Alam, and Nidhi Rastogi. Towards understanding self-play for LLM reasoning. In *The 5th Workshop on Mathematical Reasoning and AI at NeurIPS 2025*, 2025. URL https://openreview.net/forum?id=ENymEXXRVH.

Michael Dennis, Natasha Jaques, Eugene Vinitsky, Alexandre Bayen, Stuart Russell, Andrew Critch, and Sergey Levine. Emergent complexity and zero-shot transfer via unsupervised environment design. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 13049–13061. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper_files/paper/2020/file/985e9a46e10005356bbaf194249f6856-Paper.pdf.

Kefan Dong and Tengyu Ma. Stp: Self-play llm theorem provers with iterative conjecturing and proving, 2025. URL https://arxiv.org/abs/2502.00212.

Sam Earle, Graham Todd, Yuchen Li, Ahmed Khalifa, Muhammad Umair Nasir, Ze-Hua Jiang, Andrzej Banburski-Fahey, and Julian Togelius. Puzzlejax: A benchmark for reasoning and learning. *ArXiv*, abs/2508.16821, 2025. URL https://api.semanticscholar.org/CorpusID:280711515.

ARC Foundation. Arc-agi-3: A new challenge for frontier agentic intelligence. *arXiv preprint arXiv:2603.24621*, 2026.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=t9U3LW7JVX.

Chengsong Huang, Wenhao Yu, Xiaoyang Wang, Hongming Zhang, Zongxia Li, Ruosen Li, Jiaxin Huang, Haitao Mi, and Dong Yu. R-zero: Self-evolving reasoning LLM from zero data. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=96apU6YzSO.

Swadesh Jana, Cansu Sancaktar, Tomás Danis, Georg Martius, Antonio Orvieto, and Pavel Kolev. Gasp: Guided asymmetric self-play for coding llms. *ArXiv*, abs/2603.15957, 2026. URL https://api.semanticscholar.org/CorpusID:286579725.

Minqi Jiang, Edward Grefenstette, and Tim Rocktäschel. Prioritized level replay. In *International Conference on Machine Learning*, 2020. URL https://api.semanticscholar.org/CorpusID:222208809.

Minqi Jiang, Michael Dennis, Jack Parker-Holder, Jakob Foerster, Edward Grefenstette, and Tim Rocktäschel. Replay-guided adversarial environment design. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan (eds.), *Advances in Neural Information Processing Systems*, volume 34, pp. 1884–1897. Curran Associates, Inc., 2021. URL https://proceedings.neurips.cc/paper_files/paper/2021/file/0e915db6326b6fb6a3c56546980a8c93-Paper.pdf.

Yoonho Lee, Roshen Sanjay Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-harness: End-to-end optimization of model harnesses. In *Third Conference on Language Modeling*, 2026. URL https://openreview.net/forum?id=tmbOUyFx3R.

Bo Liu, Chuanyang Jin, Seungone Kim, Weizhe Yuan, Wenting Zhao, Ilia Kulikov, Xian Li, Sainbayar Sukhbaatar, Jack Lanchantin, and Jason Weston. Spice: Self-play in corpus environments improves reasoning, 2025. URL https://arxiv.org/abs/2510.24684.

Bo Liu, Simon Yu, Yiding Jiang, Ao Qu, Andrew Zhao, Zichen Liu, Junsu Kim, Zijian Zhou, Seungone Kim, Tongzheng Ren, Mickel Liu, Hanfei Yu, Zhaorun Chen, Weiyan Shi, Paul Liang, Luke S. Zettlemoyer, Yejin Choi, and Natasha Jaques. Spade: Self-play in adaptive synthetic executable environments. 2026. URL https://api.semanticscholar.org/CorpusID:291219356.

Jonathan Pappas and David Pappas. Maze Bench: Visual spatial reasoning in a 3d open world, 2026. URL https://mazebench.com. Project oversight by Florian Brand and Sebastian Müller.

Jack Parker-Holder, Minqi Jiang, Michael Dennis, Mikayel Samvelyan, Jakob Foerster, Edward Grefenstette, and Tim Rocktäschel. Evolving curricula with regret-based environment design. In *International Conference on Machine Learning*, pp. 17473–17498. PMLR, 2022.

Gabriel Poesia, David Broman, Nick Haber, and Noah D. Goodman. Learning formal mathematics from intrinsic motivation, 2024. URL https://arxiv.org/abs/2407.00695. NeurIPS 2024 (Oral).

Sebastien Racaniere, Andrew Lampinen, Adam Santoro, David Reichert, Vlad Firoiu, and Timothy Lillicrap. Automated curriculum generation through setter-solver interactions. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=H1e0Wp4KvH.

Alexander Rutherford, Michael Beukman, Timon Willi, Bruno Lacerda, Nick Hawes, and Jakob Nicolaus Foerster. No regrets: Investigating and improving regret approximations for curriculum discovery. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=iEeiZlTbts.

Sainbayar Sukhbaatar, Zeming Lin, Ilya Kostrikov, Gabriel Synnaeve, Arthur Szlam, and Rob Fergus. Intrinsic motivation and automatic curricula via asymmetric self-play. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=SkT5Yg-RZ.

Shobhita Sundaram, John Quan, Ariel Kwiatkowski, Kartik Ahuja, Yann Ollivier, and Julia Kempe. Teaching models to teach themselves: Reasoning at the edge of learnability, 2026. URL https://arxiv.org/abs/2601.18778.

Mirac Suzgun, Mert Yüksekgönül, Federico Bianchi, Dan Jurafsky, and James Zou. Dynamic cheatsheet: Test-time learning with adaptive memory. In Vera Demberg, Kentaro Inui, and Lluís Marquez (eds.), *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics, EACL 2026 - Volume 1: Long Papers, Rabat, Morocco, March 24-29, 2026*, pp. 7080–7106. Association for Computational Linguistics, 2026. URL https://doi.org/10.18653/v1/2026.eacl-long.333.

Laetitia Teodorescu, Cédric Colas, Matthew Bowers, Thomas Carta, and Pierre-Yves Oudeyer. Codeplay: Autotelic learning through collaborative self-play in programming environments. In *IMOL 2023-Intrinsically Motivated Open-ended Learning workshop at NeurIPS 2023*, pp. 4–16, 2023.

Lev S Vygotsky. Mind in society: The development of higher psychological processes, 1978.

Rose E Wang, Jesse Mu, Dilip Arumugam, Natasha Jaques, and Noah Goodman. In the ZONE: Measuring difficulty and progression in curriculum generation. In *Deep Reinforcement Learning Workshop NeurIPS 2022*, 2022a. URL https://openreview.net/forum?id=SgBHmHMctfd.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. Self-instruct: Aligning language models with self-generated instructions. In *Annual Meeting of the Association for Computational Linguistics*, 2022b. URL https://api.semanticscholar.org/CorpusID:254877310.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=NTAhi2JEEE.

Yuhao Wu, Jingyuan Zhang, Jiajun Shi, Yuxuan Zhang, Xinping Lei, Junting Zhou, Zexuan Wang, Yuchen Wu, Huan Zhou, Duo Wang, et al. Aspire: Can models self-evolve from vague goals? *arXiv preprint arXiv:2608.31111*, 2026.

Hui Xue and Fan Yang. Rethinking self-evolving agents: Do we still need prescribed optimization pipelines? *arXiv preprint arXiv:2608.09629*, 2026.

Haoran Ye, Xuning He, Vincent Arak, Haonan Dong, and Guojie Song. Meta context engineering via agentic skill evolution. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=P1jHroBS5E.

E. Zelikman, Yuhuai Wu, Jesse Mu, and Noah D. Goodman. Star: Bootstrapping reasoning with reasoning. 2022. URL https://api.semanticscholar.org/CorpusID:247762790.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin gödel machine: open-ended evolution of self-improving agents. In *International Conference on Learning Representations*, volume 2026, pp. 104223–104294, 2026a.

Jenny Zhang, Bingchen Zhao, Wannan Yang, Jakob Foerster, Jeff Clune, Minqi Jiang, Sam Devlin, and Tatiana Shavrina. Hyperagents, 2026b. URL https://arxiv.org/abs/2603.19461.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xiong-Hui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. AFlow: Automating agentic workflow generation. In *The Thirteenth International Conference on Learning Representations*, 2025a. URL https://openreview.net/forum?id=z5uVAKwmjf.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and Kunle Olukotun. Agentic context engineering: Evolving contexts for self-improving language models. In *The Fourteenth International Conference on Learning Representations*, 2026c. URL https://openreview.net/forum?id=eC4ygDs02R.

Zhengxin Zhang, Chengyu Huang, Aochong Oliver Li, and Claire Cardie. Better llm reasoning via dual-play. *arXiv preprint arXiv:2511.11881*, 2025b.

Andrew Zhao, Yiran Wu, Yang Yue, Tong Wu, Quentin Xu, Yang Yue, Matthieu Lin, Shenzhi Wang, Qingyun Wu, Zilong Zheng, and Gao Huang. Absolute zero: Reinforced self-play reasoning with zero data. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=neZSGqhxDa.

Barry J Zimmerman. Attaining self-regulation: A social cognitive perspective. In *Handbook of self-regulation*, pp. 13–39. Elsevier, 2000.

Barry J. Zimmerman. Becoming a self-regulated learner: An overview. *Theory Into Practice*, 41(2):64–70, 2002. ISSN 00405841, 15430421. URL http://www.jstor.org/stable/1477457.

A Extended Related Work

We categorize self-improvement methods based on which components of the learning process remain externally specified and how learning takes place. We consider four main axes: *the source of experience*, whether it is externally provided or generated by the learner itself; *the structure of the learning procedure*, whether it is prescribed or organized by the learner; *the source of feedback during learning*, such as externally defined rewards or evaluators versus feedback obtained through the learner's own interactions; and finally, *what is updated*, for example model weights or text-based context. Another axis of comparison is the scope of generalization the proposed methods study: some evaluate improvement within the same task distribution or domain used during optimization, while others test whether the learned adaptation transfers beyond those conditions to new environments and domains.

Asymmetric self-play methods have a teacher generate challenging problems for a student to solve, resulting in an emergent curriculum adapted to the student's needs (Sukhbaatar et al., 2018). These methods have also found success in post-training LLMs in domains such as coding, math, and scientific reasoning (Zhao et al., 2025; Huang et al., 2026; Poesia et al., 2024; Dong & Ma, 2025; Liu et al., 2025; Zhang et al., 2025b; Teodorescu et al., 2023). These are examples of methods where the source of experience is self-generated by the teacher, i.e. the teacher synthesizes which problems the student learns from, while learning then typically proceeds through gradient-based optimization of the underlying model weights. However, the optimization procedure itself remains externally specified, e.g. how many problems the teacher generates within a batch, how often the student is evaluated on a given problem to determine whether they are challenging, and subsequently how exactly the teacher is rewarded.

Related ideas appear in unsupervised environment design, where environments are generated or selected from a predefined, often parameterized environment space to form curricula adapted to the learner (Dennis et al., 2020; Jiang et al., 2020; Racaniere et al., 2020; Jiang et al., 2021; Parker-Holder et al., 2022; Rutherford et al., 2024). SPADE Liu et al. (2026) bridges the gap between UED and asymmetric self-play for LLMs by leveraging the regret principle from UED to train a teacher that produces environments rather than static question-answer tasks. Similarly to other asymmetric self-play works, the optimization procedure and task selection remain externally specified.

Asymmetric self-play methods can also suffer from diversity or performance collapse (Chae et al., 2025), motivating additional mechanisms for deciding which generated problems provide useful learning experience, for example using judges (Bailey et al., 2026) or grounding the agent in real-data hard questions as goalposts (Jana et al., 2026). These mechanisms can help stabilize self-play, but introduce additional externally specified choices about what constitutes a useful learning experience. It is also possible to address this collapse by providing the model with feedback on whether training on the teacher proposed problems leads to improvements in an external hard subset of problems (Sundaram et al., 2026). In this case, the feedback is no longer purely obtained through the learner's own interactions, while still relying on self-generated data to drive learning.

A separate line of work considers self-improvement through updates to the model's context or agent harness, rather than through updates to the model weights (Suzgun et al., 2026; Wang et al., 2025; Hu et al., 2025; Zhang et al., 2025a). GEPA uses natural-language reflection over execution trajectories to optimize model prompts via evolutionary search (Agrawal et al., 2026), while ACE maintains an evolving textual playbook that accumulates strategies and lessons from previous interactions (Zhang et al., 2026c). In these works, optimization still takes place with respect to an externally specified task distribution, and the procedure through which the context is updated is pre-specified. MCE partly relaxes the latter by co-evolving the context with a context-engineering skill that guides how context is represented and updated (Ye et al., 2026). Meta-Harness expands the object of optimization further to executable harness code, allowing the model to change how context is managed, including what information is stored and retrieved (Lee et al., 2026). The optimization still uses externally provided tasks and candidate harnesses are evaluated using an externally specified objective.

Related work on self-modifying agents expands the components under the model's control further. The Darwin Gödel Machine iteratively modifies its own agent code and evaluates the resulting agents on externally specified benchmarks (Zhang et al., 2026a), while Hyperagents additionally make the meta-level procedure used to generate future improvements itself editable (Zhang et al., 2026b).

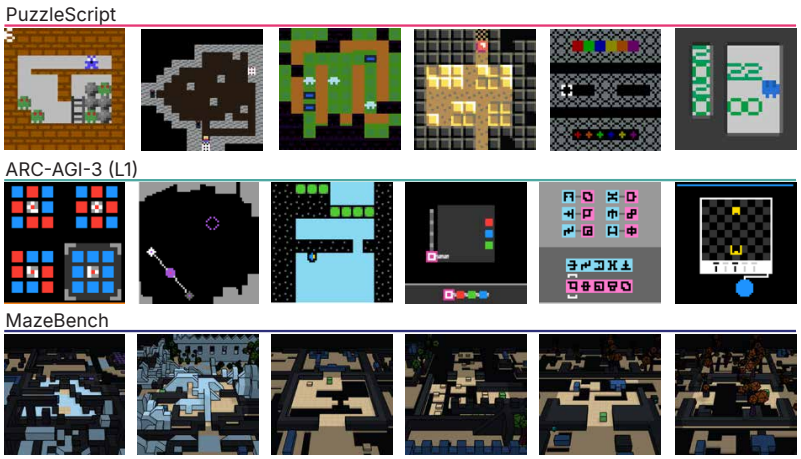

Figure 9: **Examples of evaluation benchmarks.**

These approaches therefore allow increasingly large parts of the self-improvement mechanism to change, while still operating within an externally specified task and evaluation setting.

Xue & Yang (2026) introduce Open-Ended Optimization, where the objective, available interactions, resource budget, data boundary and evaluation are fixed, but the model is free to organize the improvement process itself. The tasks used during this optimization are still externally provided and the resulting performance is evaluated on held-out examples from the same benchmark setting, rather than studying transfer beyond the environment in which optimization took place.

Closest to our setting, in concurrent work, Wu et al. (2026) introduce ASPIRE, a benchmark for self-evolution from vague capability goals. Agents (sometimes a stronger teacher model optimizing a smaller student) improve weights or harnesses with access to external data and, in some settings, aggregate scores from the hidden evaluation. Performance is then measured on hidden items from the same goal. In contrast, we study a model improving itself from self-generated experience alone, and evaluate transfer to different environments.

B Evaluation Benchmarks

Table 2: **Evaluation settings for each benchmark.**

Benchmark	# Games	Turns (H)	Tokens / game	Eval rollouts / game
PuzzleScript-t20	20	20	25,000	4
PuzzleScript-t30	20	30	31,250	4
ARC-AGI-3 (L1)	17	30	31,250	4
MazeBench	16	100	75,000	2

Examples from each dataset are shown in Figure 9.

PuzzleScript-t20. Both PuzzleScript benchmarks are drawn from a large pool of community-authored games (Earle et al., 2025). We sample a random set of 500 game-levels, and filter for levels solvable in ≤ 12 turns. We then further filter out games that `Claude Opus-5` solves in 8/8 rollouts, leaving a dataset of 20 game-levels, each from unique games. Note that after filtering, we re-run the baseline.

PuzzleScript-t30. We follow the same procedure described for PuzzleScript-t20, however apply an optimal solution length filter of 13-22 to capture longer-horizon, more complex game levels.

ARC-AGI-3 (Level 1) (Foundation, 2026) Due to the token cost of evaluating on ARC-AGI-3, we use Level 1 of each game. There are 25 games offered in the benchmark; as with our PuzzleScript

datasets, we filter out game levels that `Claude Opus-5` solves in 8/8 rollouts, leaving 17 game levels. The initial observation for each game is the full 64×64 frame rendered as hex digits; subsequent observations list on the changed cells.

MAZEBENCH (Pappas & Pappas, 2026). The MazeBench world map contains 256 rooms . Adopting the room-completion metric, we use each room as a single completion task. The player begins as if it had walked into the room, and wins if it exits through a different edge (which requires navigating the room obstacles/mechanics). We filter for rooms that are solvable within 50 turns and choose a random draw of 16 rooms. We use the common SPL (success weighted by path length) metric to measure the player's efficiency in navigating the room. Note that MazeBench is represented differently from our other three benchmarks: an isometric-style ASCII projection from a top-diagonal view. In addition to movement actions, the player has the ability to rotate the camera and view the room from additional angles. MazeBench also includes obstacles unique to the 3D setting: raised floors, ramps, stacked blocks, occlusion, and voids.

C EXPERIMENT DETAILS

C.1 TEACHER-PLAYER RUNS

Harness details. A single run consists of a teacher agent initialized in a persistent harness session, and several short-lived player harness sessions. Claude models run in Claude Code (v2.1.232) and all other models run in Pi (v0.85.1) accessed through OpenRouter. All harness sessions run inside a Singularity container with the following settings:

- The session runs in a fresh sandboxed directory viewable to the agent as `/workspace`.
- The harness's default system prompt is replaced with ours (see Appendix C.4).
- Tools are only granted through an allowlist; all other extensions, skills, prompt templates, context files, etc are disabled. This includes tools such as web search, web fetch, subagent, and task-management tools.
- Automatic memory is disabled.
- Automatic compaction is left on, however due to our teacher and player budgets, no sessions consume enough tokens to trigger compaction.
- Interaction with a human operator is disabled; messages to and from the session are managed programmatically.

The possible teacher-granted tools are **Read/Write** tools (including Edit, Glob, and Grep for managing files), **Bash**, and our custom **Gameplay** tool.

The teacher working directory is initialized with:

- `player_folder`: A designated directory for player text artifacts, initialized with `player_prompt.md` containing the default player prompt (Appendix C.4). Teacher updates are tracked via Write/Edit tool calls to `player_folder`.
- `puzzlescript_docs/` containing PuzzleScript documentation for PuzzleScript Gameplay runs, and `game_interface.md` detailing the expected Python class interface for Python Gameplay runs (Appendix C.4).

Gameplay. The teacher calls the Gameplay tool with a source game file (either PuzzleScript or Python) and optionally the path to a folder holding the text artifacts that the teacher wishes to grant the player. If the path is omitted, the player receives the default prompt and no auxiliary files. PuzzleScript games are rejected if they do not compile in the PuzzleScript JavaScript engine, or if a BFS solver verifies that the winning state is unreachable (if the BFS solver times out, the game is accepted). Python games are likewise checked for compilation, and safety checks (network access, file access, etc). The Gameplay tool runs a fresh player session with only Read and Glob tools, and returns the following output to the teacher:

- The player score.

- The game end reason (turn budget, token budget, or player won).
- The status of the teacher's gameplay and update budgets.
- A path to a transcript containing player reasoning, actions, and observations.

C.2 Tool/Budget Awareness Ablations.

In Section 6.2 we ablate the teacher's awareness of the evaluation setting via tool constraints. We run 3 seeds each of the following settings, using the default teacher update budgets:

1. *Read-Write*: Teacher has access only to the file-management tools (Read, Write, Edit, Glob, Grep).
2. + *Bash*: The above tools, and Bash access (subject to sandbox constraints).
3. + *Gameplay (PS)*: The above tools, and a Gameplay PuzzleScript tool. We do not impose a turn or token budget for each gameplay; instead the teacher chooses it, enabling another axis of freedom. Rather than a gameplay budget the teacher is granted a player token budget and player turn budget that it draws upon for each gameplay. We set the turn budget to 3000 and token budget to 3.75M, thus compute-matching the t-20 constrained setting (150 gameplays $\times$ 20 turns/game and 150 gameplays $\times$ 25k tokens/game).
4. + *t20 Token Budget*: This setting is the same as the previous, however we impose the t20 per-game token budget (25k) through the Gameplay tool, and the teacher chooses the game horizon.
5. + *t20 Turn Budget:* This is our canonical setting, in which the teacher has a gameplay budget, and the Gameplay tool imposes the t20 turn- and token-budgets (20 turns and 25k tokens).

C.3 Settings

Our settings for teacher-player runs are shown in Table 3. Ablations from our main experiments are noted in the table. We also ablate the teacher update and gameplay budgets in Appendix D.3.

Setting	Default Value
Teacher update budget	20
Teacher gameplay budget	150
Max turns per game	{20,30}
Max output tokens per game (summed over turns)	{25000,31250}
Output-token limit per model response (teacher and player)	20000
Reasoning effort (teacher and player)	low

Table 3: Teacher–player session settings.

C.4 Prompts

Default player prompt The default player system prompt is shown below, and is used for all baselines. The last line lists the artifact files when the player is given any.

Player system prompt

```
You are playing a game. Each turn, end your response with:

ACTION: the action you take this turn (just the action, nothing else).
Your working directory is /workspace.
There are no files in your working directory.
```

Teacher prompt. The teacher's default system prompt is shown below.

Teacher system prompt

```
Your goal is to improve an agent's ability to play unfamiliar single-player
    games.
All games have the following attributes:
- Single player.
- At each player turn, the player takes an action, and receives feedback on the
    game state.
- The game has an objective and a score.
- The player plays without full knowledge of the rules, and discovers the game
    mechanics through play.

The player has basic file read tools.

You can write whatever artifacts you want for the player to player_folder/. ONLY
    write artifacts for the player to player_folder/. It is the player's working
    directory. The player prompt is in player_folder/player_prompt.md, and may be
     edited.
**Generalizability.** The goal is to help a blind player do well on NEW games of
     the same style/form but DIFFERENT tasks and subject matter. Your artifacts
    must TRANSFER to improve performance on unseen games.
```

The session opens with the user message `Begin.` If the teacher stops before either the update or gameplay budget is spent, it is sent:

Continuation message

```
Keep working. Revise your artifacts further. Revisions remaining: <N>.
```

PuzzleScript Gameplay. In runs with PuzzleScript Gameplay enabled, the following line is added to the system prompt: `"The PuzzleScript language reference is in puzzlescript_docs/. Do NOT place your generated games or output files in player_folder/."` The PuzzleScript language reference in `puzzlescript_docs/` is taken from the official PuzzleScript documentation (https://www.puzzlescript.net/Documentation/documentation.html).

The teacher is also shown the following description for the Gameplay tool alongside the other tools presented by the harness. The description follows the tool format mandated by the harness (Claude Code for Claude models, and Pi for all other models), and is presented alongside the other available tools (Read, Write, etc.)

Gameplay tool description

```
play_own_game: Play one game you wrote with a fresh player agent, and wait for it
     to finish. Several of these can run at once.
  game_file (required): path to the PuzzleScript file holding your game; must be
    outside player_folder/
  level: index of the level in that file to play; defaults to 0
  out (required): path this play's full result is written to; must be outside
    player_folder/
  player_folder: path to the folder of artifacts to give the player; omit it to
    play with none
```

In ablations where the teacher may choose the turn- and token-budget for the game, we add arguments to the Gameplay tool accordingly.

For GPT and GLM teachers (Pi harness) we make minor adjustments to resolve initially-observed malformed tool calls. The Gameplay tool description adds *"Pass player_folder to give your artifacts to the player; omit the player_folder argument to play with the default prompt."*.

All teachers receive the remaining gameplay and update budgets in every GamePlay tool result. In initial runs GPT and GLM teachers misinterpreted this feedback, so for these models we also state the budget in the system prompt: *"There are two budgets: 150 game plays and 20 revisions. A revision is any tool call that changes /workspace/player_folder/; writing two files in one turn spends two. Every play result reports the revisions left and gameplays used. The run ends when either budget is exhausted."*

Python Gameplay. In our Python ablations the teacher issues games as Python classes instead of PuzzleScript text files, thus removing implicit constraints on the generated game type. In this case, the teacher system prompt replaces the PuzzleScript documentation files in the teacher sandbox with a Markdown file specifying the Python game interface, and replaces the line pointing to PuzzleScript documentation with *"The game interface is in game_interface.md."*

game_interface.md

```
# Game interface

Games are written as a self-contained class named `GameEnvironment` in a Python
    file.
`__init__`, `step`, `score` and `example_actions` are required; `
    initial_observation` is optional.

```python
class GameEnvironment:
    def __init__(self):
        self.done = False  # set to True to end the game
        ...

    def step(self, action: str) -> str:
        # `action` is a player-provided string. Parse it and handle unexpected
    input without
        # crashing. Return the feedback string: this is the player's only window
    into the game.
        ...

    def score(self) -> float:
        # The player's objective, queryable at any time. A number or a bool. The
    player never
        # sees this -- put anything you want it to know in step()'s feedback.
        ...

    def example_actions(self) -> list[str]:
        # A few representative valid actions, used only to test the game.
        ...

    def initial_observation(self) -> str:
        # OPTIONAL: what the player is shown at turn 0.
        ...
```
```

C.5 Self-Regulated Learning Annotations

We score each teacher session using a rubric adapted from Zimmerman's cyclical phases model of self-regulated learning (Zimmerman, 2000). The rubric is initially adapted from Zimmerman (2000; 2002) using `GPT 6-Astra`, and then reviewed and refined by the authors. The unit of analysis is the teacher-player system. We annotate using `GPT 6-Astra` as a judge (medium reasoning effort).

The rubric covers 13 subprocesses across three phases. Each subprocess is scored on a 0-3 scale grading its presence and quality:

- **0, nonfunctional or omitted:** the subprocess is missing, unused, or carried out in a way that defeats its purpose.
- **1, weak:** the function is recognizable, but substantial gaps limit its usefulness.
- **2, adequate:** it serves its function with suitable specificity, fit, and execution.
- **3, strong:** it serves its function consistently across relevant opportunities, and handles important limitations and contrary evidence well.

Each item has descriptions for scores 1-3; the full rubric is in the supplementary material. An item may be marked *unassessable*, but only when no defensible score can be assigned. Note that the original Zimmerman framework contains 17 subprocesses, however we filter out 4 that the judge consistently judged as unassessable across all models in initial pilots.

Each teacher session is judged in batches. A single batch is one round of teacher actions (e.g., Bash commands, Gameplays, thinking) ending with a revision made to the player text artifacts. For each batch, we adopt a two-pass judging protocol:

- **Pass 1:** The judge receives an initial input containing the rubric, a short description of the teacher-player setup, the teacher's system prompt and resource budgets, a record of teacher actions for the current batch, and an index of the batch gameplays. To contextualize the current batch we include the record of teacher actions from previous batches. The judge returns requests for additional evidence to support judgements, such as the full inputs/outputs of teacher tool calls, and gameplay transcripts.
- **Pass 2:** The judge receives the requested evidence and returns a score for each of the 13 subprocesses, along with cited evidence, counterevidence, and a short rationale.

For each teacher session we average each subprocess score over all batches.

D Further Results

D.1 SRL Annotations across seeds

In Figure 10 we plot alignment to our adapted SRL rubric for individual teacher sessions across three models, indicating stable behaviors across runs.



Figure 10: **SRL annotations across independent teacher sessions**.

D.2 Task selection statistics for older model generations

In Figure 11 we show that the older model generations in each family that we evaluate do not consistently display the same learnability-based task selection that we observe in newer generations of models (Figure 5).

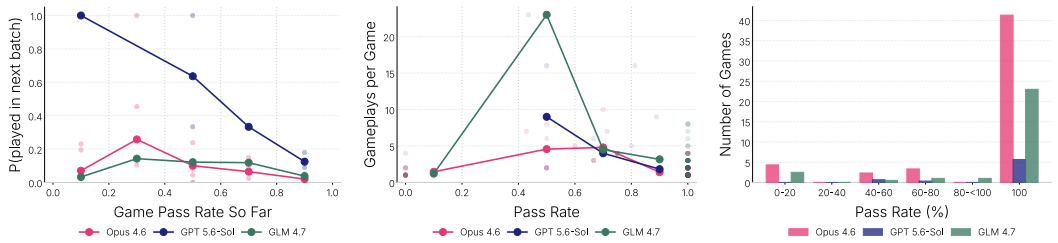

Figure 11: **Task selection for older model generations.**.

D.3 Does self-organized self-improvement benefit from compute?

In Figure 12 we study the impact of running teacher sessions with varying update budgets (gameplay budgets are scaled accordingly). We find that performance increases and then plateaus with extra compute. We also note that teacher behavior, measured through the magnitude of updates over time, fluctuates under different resource regimes, and finally observe that performance scales with increased gameplay experience.

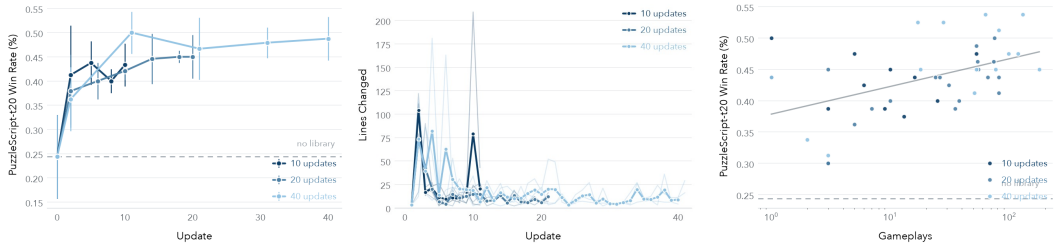

Figure 12: **Ablating the teacher budgets**. (left) Performance increases and then plateaus with extra compute. (middle) The teacher's knowledge of its own budget status leads to different behaviors under different resource regimes. (right) Performance on the PuzzleScript-t20 test set scales with more gameplay experience.

E Qualitative Examples

We include two further excerpts from teacher sessions.

E.1 GPT-6 Astra: comparing two artifact versions on held-out levels

This excerpt is from a `GPT 6-Astra` teacher session in the canonical setting (Section 5.1). Earlier in the session, the teacher wrote a Sokoban level generator with a BFS solver (`gen_soko.py`, random seed 42) and used it to produce `soko0–soko3`. The teacher refers to its own artifact versions as v6, v7, and v8.

GPT-6 Astra teacher, gameplays 33–54

```
[Gameplay] swap, soko3, soko2, soko0  (artifacts v6)
  –> 1, 0, 0, 1

[Thinking] I'm thinking about model stability and how it isn't monotonic. I need
    to conduct multiple evaluations for a full sample comparing the baseline.
    [...] To gauge improvements, I should look at different bases like soko0,
    soko1, soko2, and soko3.

[Gameplay] soko0, soko1, soko2, soko3  (no artifacts)
  –> 1, 1, 0, 1
```

```
[Bash] Prints the player's turns on swap, soko2, soko3 (artifacts v6).

[Thinking] I'm thinking about how to improve global planning. The idea is to
    examine at least two candidate plans at any planning point before making a
    choice. I want to list both routes to check them against each other,
    ensuring costs are assessed carefully without using arbitrary numbers.

[Edit] player_folder/player_prompt.md  (v7)
  + [...] With multiple goals, explicitly compare two candidate complete plans (
    assignments, order, and setup routes), then choose one. Do not merely
    justify the first move that comes to mind or claim a move count without
    checking the route.
  + [...] With interchangeable objects, compare alternative assignments: nearest-
    object-to-nearest-goal can be much worse than crossing assignments. A
    longer straight object path can save many player detours.

[Gameplay] soko2, soko3, swap, gravity  (artifacts v7)
  -> 0, 1, 0, 1

[Thinking] [...] I want to avoid overfitting on small tests, so I'll incorporate
    varied and randomized holdouts.

[Bash] Prints the player's turns on soko2 and swap (artifacts v7).
  soko2, turn 0: "I'll initially assign the left crate to the left target and the
     right crate to the right target. Crossing those assignments would require
    longer travel across row 2 and risk the crates blocking each other."

[Thinking] I'm noticing that a lot of failures stem from the v7 plan, which just
    assumes the nearest option is the best without comparing alternatives. It's
    important to emphasize not to dismiss options just because they involve a
    longer travel distance. [...] I'm realizing that sometimes the correct move
    might actually increase distance temporarily, which is a new insight.

[Bash] cp player_folder/player_prompt.md tests/prompt_v7.md
       Reruns gen_soko.py with random.seed(42) replaced by random.seed(9182),
    writing soko4-soko9.

[Thinking] [...] The best way to compare v7 and v8 seems to be with four tests
    from v7 and running evaluations on those, plus two regressions and new
    controls.

[Gameplay] soko4, soko5, soko6, soko7  (artifacts v7)
  -> 1, 1, 1, 1

[Thinking] I think v7 is looking good, but I need to consider that only the
    critical training cases are failing, which is fine to some extent. I want to
     avoid overspecifying two puzzles since that can lead to wasted revisions.
    [...] So, I can always choose to stick with the old version if it's held out.


[Write] player_folder/player_prompt.md  (v8, full rewrite)
  [...] With several goals, explicitly compare at least two complete assignments/
    orders, including the player's setup walks. Do not dismiss an alternative
    merely because its objects travel farther. A long straight delivery can be
    cheaper than several short deliveries with detours. If the direct route
    fails, consider moving an object temporarily AWAY from its goal into another
     corridor; apparent regression can be necessary.

[Gameplay] soko4, soko5, soko6, soko7, soko2, swap  (artifacts v8)
  -> 1, 0, 1, 1, 1, 0
```

E.2 CLAUDE OPUS-5: SIMULATING GAMES WITHOUT A GAMEPLAY TOOL

This excerpt is from a `Claude Opus-5` teacher session in the *Read-Write-Bash* setting (Section 6.1), where the teacher has no Gameplay tool. The teacher writes a game in Python and uses it to test its own advice.

Claude Opus-5 teacher (Read-Write-Bash)

```
[Thinking] Without an actual game engine to test against, I'm thinking about how
    to reduce generic advice and make things more concrete instead. [...]

[Text] Let me pressure—test my artifacts against a concrete simulated game
    before revising further.

[Bash] Writes /tmp/gt/game.py:
        # hidden: market sim. buy low sell high, 20 turns, capital compounding
      and steps through it:
  help —> Commands: buy N, sell N, wait, upgrade, status | T=1/20 gold=100 stock
    =0 price=12
  status —> Nothing happens. | T=2/20 gold=100 stock=0 price=10
  buy 5 —> Bought 5. | T=3/20 gold=50 stock=5 price=13
  sell 5 —> Sold 5. | T=4/20 gold=115 stock=0 price=10
  upgrade —> Upgraded. | T=5/20 gold=65 stock=0 price=7
  buy 3 —> Bought 3. | T=6/20 gold=44 stock=3 price=4
  xyzzy —> Nothing happens. | T=7/20 gold=44 stock=3 price=6

[Thinking] This is exposing real gaps in my playbook: I need to think about
    timing (whether state updates before or after my action), how to handle
    parameterized actions with magnitudes, distinguishing control variables from
     noise, and how thresholds and costs factor into decisions.

[Write] player_folder/genres.md
  ## A. Economy / resource sim [...]
  — **Invest early.** Upgrades/multipliers pay back over remaining turns. Buy
    an upgrade only if 'payback_period < turns_remaining' (see sizing.md). Late
    upgrades are wasted.

[Write] player_folder/sizing.md
  ## 1. Probe small, then commit big
  [...] Once the rate is confirmed and favorable, **scale up aggressively** ——
     timid sizing for the whole game is a common way to underscore.

[Thinking] I should validate by dry—running an agent that follows the playbook
    against my mock game to see if it actually wins. [...]

[Text] Let me quantitatively validate that the playbook's advice actually
    outperforms naive play on the simulated game.

[Bash] Simulates two scripted strategies over 300 seeds of the market game:
  def naive(t,rem,g,s,p,u,h):  # timid, buys 1, never upgrades, hoards
  def playbook(t,rem,g,s,p,u,h):
      [...]
      if rem>10 and g>=50 and u<2 and p>m: return ('upg',0)  # invest early,
    payback<rem
      if p<m: return ('buy',int(g//p))      # commit big below mean
  naive     mean=   102.0 median=   102.0 min=   85.0
  playbook  mean=    77.1 median=    44.4 min=    0.0

[Thinking] [...] The real lesson here is that my playbook advice to "commit big"
    and "invest early" could actively hurt without verifying there's a genuine
    edge —— I should temper that guidance with checks against mean—reversion
    assumptions before recommending all—in strategies.
```

```
[Text] Important —— my "commit big / invest early" advice **lost badly** (77
    vs 102). Let me diagnose rather than encode a losing heuristic.

[Bash] Simulates variants over 400 seeds:
  naive          mean=   101.8 median=   102.0 min=   82.0
  buylow_noupg   mean=   154.2 median=   127.0 min=   15.0
  buylow+upg     mean=   102.9 median=    78.6 min=    8.8

[Text] Diagnosed: **buy—low/sell—high alone scores 154 vs naive's 102 —— but
    the upgrade was a trap** that destroyed the gains. My "invest early greedily
    " advice was wrong. Let me test reserve sizing too.

[Bash] Sweeps the fraction of gold committed per buy (frac) and the number of
    observation turns before trading (warm), over 400 seeds:
  frac=0.5 warm=3  mean=  177.4 med=  136.0 min=  23.0
  frac=0.75 warm=3  mean=  181.1 med=  131.0 min=  17.0
  frac=1.0 warm=0  mean=  154.2 med=  127.0 min=  15.0
  [... 6 further settings]

[Text] Now I have evidence—based numbers. Let me correct the two places where my
     advice was actively wrong.

[Edit] player_folder/sizing.md
  ## 3. Investment / upgrade rule —— UPGRADES ARE OFTEN TRAPS
  Do **not** assume an offered upgrade is good because games usually reward
    investment. [...]
  *Tested:* in a 20—turn market sim, adding exactly one such upgrade to an
    otherwise good buy—low/sell—high strategy cut the final score from **154 to
     103** —— worse than doing nothing clever at all.

[Edit] player_folder/sizing.md
  But scale up only once the **edge is confirmed**. "Commit big" applied to a
    strategy you haven't verified just amplifies a loss. [...]
  **Commit about 50—75% of a resource per move, not 100%.**
```