AGENT PLASTICITY: MEASURING SELF-IMPROVEMENT THROUGH EXPERIENCE

Anonymous authors
Paper under double-blind review

ABSTRACT

AI agents are increasingly being used in systems requiring them to self-diagnose and improve through experience, yet evaluations largely measure what they can do, not how effectively they learn. We study persistent self-improvement with frozen model weights, where agents learn by constructing and refining tools, skills, memories, and strategies inherited by future instances. Despite comparable learning opportunities, frontier models differ sharply in their ability to self-improve with experience, and gains on held-out in-distribution games often carry over only partly to harder, out-of-distribution games. We introduce *agent plasticity*, the efficiency of an agent in improving future performance as it gets more experience. Tracing failures through the improvement loop, we find that weaker agents often fail to reuse relevant artifacts, while highly plastic agents often fail despite reusing them, highlighting a shift in the bottleneck from reusing artifacts to building generalizable artifacts and applying them effectively. These results establish plasticity as a dimension of long-lived agent capability that static evaluations overlook.

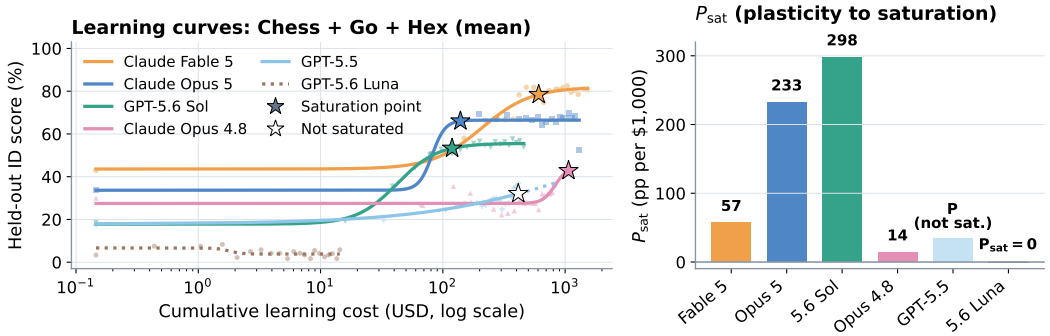


Figure 1: **Frontier agents differ widely in how efficiently experience helps.** Held-out ID (in-distribution) score against cumulative learning cost (left) and plasticity P^{sat}, i.e., score gain up to saturation per $1,000 of learning cost (right). GPT-5.5 is not yet saturated (hollow star, lighter bar).

1 INTRODUCTION

Most agent evaluations ask what an agent can accomplish at a fixed point in time. But long-lived agents can learn from experience: they accumulate memories, construct tools, develop skills, and refine strategies inherited by future instances. Two agents with similar initial capabilities may therefore follow radically different improvement trajectories. Evaluating such agents requires a second question: *how effectively does experience make them better?*

Recent work enables self-improvement through reflection, memory, reusable workflows, and executable skills (Shinn et al., 2023; Zhao et al., 2024; Wang et al., 2024; 2025; Liu et al., 2025). Other approaches evolve agent implementations or improvement mechanisms (Hu et al., 2025; Yin et al., 2025; Zhang et al., 2026b;a; Lin et al., 2026; Wang et al., 2026b), or train models to control their future adaptation (Zweiger et al., 2025). While these approaches develop methods for improving agents, they leave a complementary measurement question: *how efficiently can an agent convert experience into lasting capability?*

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.

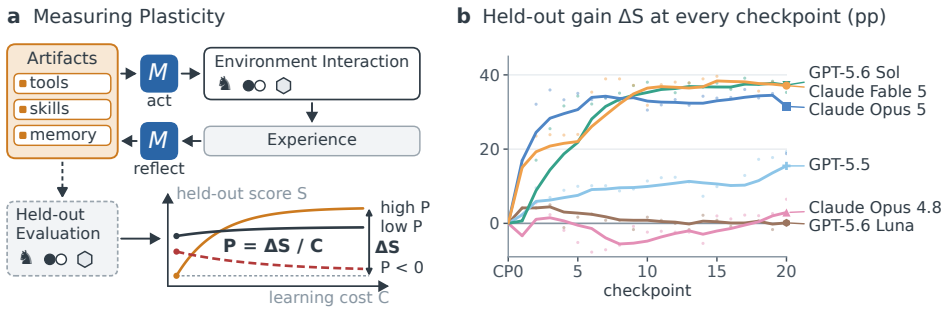


Figure 2: **Frontier agents differ widely in how much experience helps.** (a) A frozen model revises its persistent artifacts from experience, and every checkpoint is scored on held-out games. (b) Held-out gain ΔS, averaged over chess, Go, and Hex (lines: trailing five-checkpoint mean).

We call this property *agent plasticity*. Capability measures what an agent can accomplish; plasticity measures how efficiently its capabilities improve through experience. A capable agent need not be highly plastic, and the agent reaching the strongest endpoint need not learn most efficiently. Distinguishing these properties requires measuring improvement over time, accounting for learning cost, and testing whether improvements generalize beyond the experience that produced them.

We study plasticity under a controlled protocol in which model weights remain frozen and improvement occurs only through persistent, self-constructed artifacts. Fresh agent instances inherit tools, skills, strategies, and textual memory refined from previous training interactions. Held-out evaluation trajectories and scores are never exposed to reflection, separating artifact-mediated improvement from parameter updates and direct memorization of evaluation experience.

We evaluate frontier models across repeated rounds of acting and reflection in chess, Go, and Hex. At each checkpoint, we measure performance on training interactions, held-out in-distribution (ID) opponents, and out-of-distribution (OOD) opponents. The resulting learning trajectories differ sharply across models and tasks (Figures 3 and 9). Some agents achieve sustained held-out improvements, while others remain near their initial performance. Gains against held-out ID opponents often carry over only partly to stronger, held-out OOD opponents.

Importantly, endpoint capability and acquisition efficiency are distinct. On our combined three-game measure, Claude Fable 5 reaches the highest fitted endpoint, whereas GPT-5.6 Sol achieves substantially greater estimated plasticity per unit learning cost (Figure 1). Endpoint capability alone thus obscures differences in how efficiently agents improve.

To investigate these differences, we trace failures through the artifact lifecycle: was a relevant artifact created, was it used, and did failure persist despite its use? Many agents fail to reuse relevant artifacts, while strongly improving agents often use them consistently yet still make errors. This suggests that improving retrieval alone is insufficient; effective persistent learning also requires building artifacts that generalize and applying them effectively.

Our contributions are a controlled protocol for evaluating persistent fixed-weight improvement, an efficiency-based measure of agent plasticity, and an empirical analysis of the learning trajectories, transfer, and artifact-use failures of frontier agents.

2 Measuring Persistent Self-Improvement

We measure whether agents improve through experience while their model weights remain fixed. Our protocol isolates persistent adaptation, evaluates its effect on held-out performance, and measures the learning cost required to achieve it (Figure 2).

2.1 Fixed-Weight Agent Lineages

An agent at checkpoint t consists of a fixed language model M and a persistent artifact inventory I_t containing executable Python tools, natural-language strategies or skills, and textual memory. Each acting episode begins in a fresh context; conversational history and temporary files do not persist.

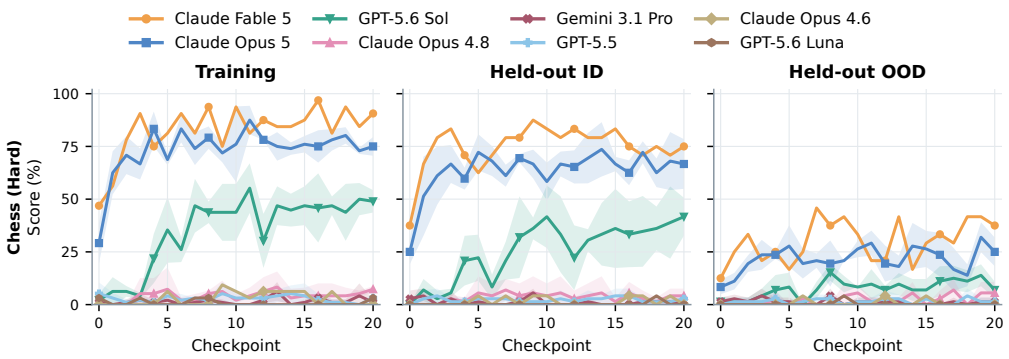


Figure 3: **Persistent improvement varies across models.** Checkpoint scores in Hard chess on training, held-out ID, and held-out OOD interactions. Scores are $100(W + 0.5D)/N$. Easy chess, Go, and Hex are shown in Appendix Figure 9.

Training episodes produce evidence $\mathcal{E}_t^{\text{train}}$, including trajectories, outcomes, execution traces, errors, and move-quality feedback. A fresh reflection instance of the same model uses this evidence and the current inventory to propose an update:

$$I_{t+1} = U_M\big(I_t, \mathcal{E}_t^{\text{train}}\big)\,. \tag{1}$$

Proposed updates must pass structural and executable checks before becoming persistent. These checks establish validity, not improvement. Rejected attempts leave the committed inventory unchanged but may inform later attempts. Exhausting the rejection limit retains I_t; non-rollback infrastructure interruptions leave the transition incomplete. Learning costs include model-attributable rejected attempts. The resulting sequence $(M, I_0), \ldots, (M, I_T)$ is an *agent lineage*. Consecutive inventories may be identical, and accepted revisions may improve, preserve, or degrade performance.

Evaluating Persistent Improvement. We distinguish three evaluation splits: **training interactions**, whose trajectories and feedback are available to reflection; **held-out ID interactions**, with opponents in the intended training difficulty regime; and **held-out OOD interactions**, with stronger opponents. Neither held-out trajectories nor their scores are exposed to reflection.

Let $S_{m,e,s,t} \in [0, 100]$ denote model m's score in environment e, split $s \in \{\text{train}, \text{ID}, \text{OOD}\}$, at checkpoint t. Held-out ID improvement measures generalization beyond training interactions; held-out OOD improvement measures transfer across opponent difficulty, not arbitrary new tasks. We report full checkpoint curves because persistent adaptation need not be monotone (Figures 3 and 9).

Accounting for Learning Cost. Learning cost includes model calls used to interact with the environment and obtain training experience and revise persistent artifacts, including reflection attempts that get rejected due to issues such as malformed outputs or illegal Python imports. The cumulative cost of checkpoint t is

$$C_{m,e,t} = \sum_{u=0}^{t-1} \big(c_{m,e,u}^{\text{train}} + c_{m,e,u}^{\text{reflect}}\big)\,, \qquad C_{m,e,0} = 0. \tag{2}$$

We record token usage and express cost-based measures in dollars as the closest available proxy to compute cost. This measures learning cost, not physical computation.

2.2 Measuring Agent Plasticity

We operationalize *agent plasticity* as held-out performance gain per unit learning cost. At checkpoint T, average plasticity is

$$P_{m,e,s,T}^{\text{score}} = \frac{S_{m,e,s,T} - S_{m,e,s,0}}{C_{m,e,T}/1000}, \tag{3}$$

measured in score percentage points per \$1,000. This quantity can be negative and depends on the evaluation horizon: it declines as cost accumulates after performance plateaus.

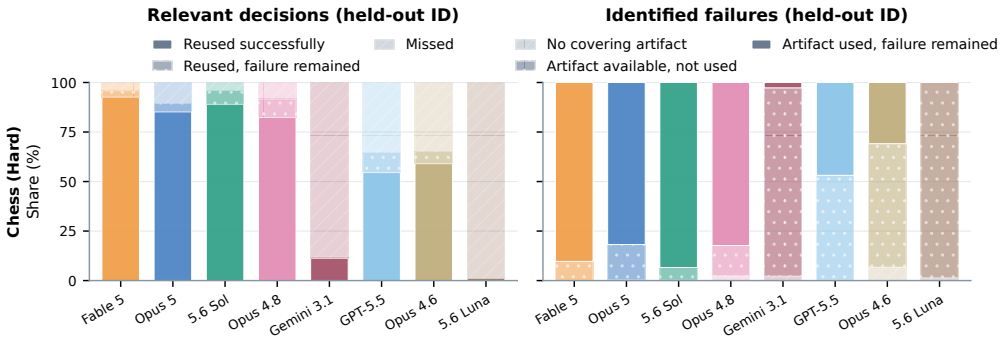


Figure 4: **Artifact reuse and failure composition reveal different candidate bottlenecks.** Panels show Hard chess on held-out ID games. Left: relevant decisions (moves where at least one saved artifact is relevant to choosing or playing the move, used or not) are classified as successful reuse, reuse with failure remaining, or missed reuse. Each decision contributes once. Right: identified failures are classified by whether a covering artifact was absent, unused, or used while failure remained. Failure shares do not measure overall failure frequency. Counts pool checkpoints 1–20. Easy chess, Go, and Hex appear in Appendix Figure 28; training games appear in Appendix Figure 26.

Plasticity to estimated saturation. Our headline measure summarizes acquisition efficiency up to estimated saturation. For each model–environment curve, we fit held-out ID scores with

$$\hat{S}(C) = S_0 + A\frac{C^h}{C^h + C_{50}^h}, \tag{4}$$

where S_0 is the initial score, A is the fitted rise, C_{50} is its half-rise cost, and h controls curve shape. All four parameters are fitted jointly by binomial maximum likelihood. This monotone fit captures the overall rise rather than individual fluctuations or regressions.

Let C^* be the cost of the first observed checkpoint at which the fit reaches 90% of its rise. We define

$$P_{m,e}^{\text{sat}} = \frac{\hat{S}(C^*) - S_0}{C^*/1000}. \tag{5}$$

If the fitted 90% rise is below 10 percentage points or a likelihood-ratio test against a flat curve gives $p \geq 0.10$, we assign $P^{\text{sat}} = 0$. This denotes no sufficiently large, statistically supported rise.

If no observed checkpoint reaches the threshold, we extrapolate the fitted 90% point up to three times the final observed cost. When it lies beyond that limit, saturation cannot be established; we instead report plasticity to date, the fitted rise at the final observed cost per $1,000 of that cost. Intervals use a parametric bootstrap. Figure 1 shows the combined learning curves and saturation plasticity.

2.3 Diagnosing the Improvement Loop

Performance curves establish whether agents improve, but not where improvement breaks down. For each identified decision failure, we ask whether a covering artifact existed and, if so, whether execution evidence shows that it was materially used. This yields three mutually exclusive categories (Figure 4, right): $\mathcal{F}_{\text{absent}}$: no covering artifact existed, $\mathcal{F}_{\text{missed}}$: a covering artifact existed but was unused, $\mathcal{F}_{\text{used}}$: a covering artifact was used, but failure remained.

Because these categories condition on failure, we separately measure *decision-level reuse*: among *relevant decisions*, i.e., moves where at least one saved artifact is relevant to choosing or playing the move, whether or not it is used (about 98% of chess moves after checkpoint 0), the fraction at which any relevant artifact was materially used (Figure 4, left). Each decision contributes once. We also report identified failures per relevant decision, a normalized count rather than a probability, since multiple failures may occur at one decision.

Artifact relevance and failure coverage are distinct: an artifact may be broadly relevant without addressing a particular mistake. Our analysis combines semantic judgments by Gemini 3.1 Pro with execution evidence, including file reads and persistent Python invocations (Appendix H). These diagnostics are observational: artifact use does not establish causal effectiveness, and failure after use does not distinguish inadequate artifacts from inadequate application.

3 EXPERIMENTS

We investigate three questions: which agents acquire persistent held-out improvements, how efficiently they do so, and where the improvement process breaks down.

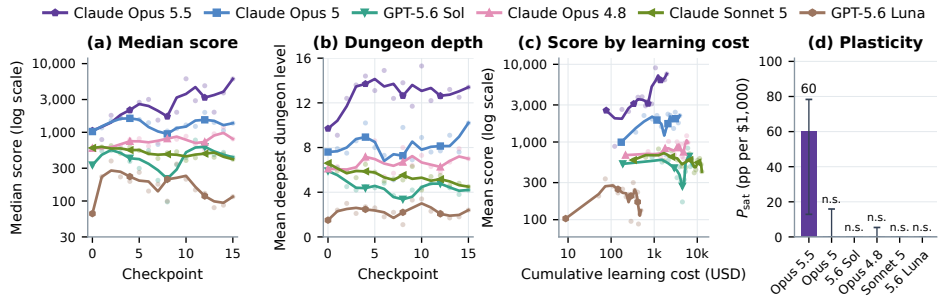


Figure 5: **Only Claude Opus 5.5 improves reliably in NetHack.** (a) Median score and (b) mean deepest dungeon level over the ten games of each checkpoint, and (c) mean score against cumulative learning cost at cache-aware provider prices. (d) P^{sat} in percentage points of the NetHack score range per $1,000, with 90% intervals; n.s.: no reliable rise. Details in Appendix D.

3.1 EXPERIMENTAL DESIGN

We follow the fixed-weight protocol in Section 2. Each actor operates in an isolated Python container and interacts with a trusted game process through a restricted board API. It can inspect positions, analyze variations, and submit actions, but cannot access the opponent process, model credentials, held-out evaluations, network, or other runs. Between checkpoints, the same model receives training trajectories, feedback, and the current inventory. Proposed artifact revisions must pass structural and executable validation before becoming persistent.

Primary chess runs begin at checkpoint 0 with the bare harness and continue for 20 update rounds. At each checkpoint, the current inventory plays complete games, each in a separate actor episode: 16 training, 12 held-out ID, and 12 held-out OOD games in Hard chess (Appendix Table 2 lists every environment). Reflection on the 16 training games then produces the next checkpoint's inventory. A round may retain the preceding inventory after candidate rejection. Every game starts from the standard initial position, and Stockfish is limited to 20,000 search nodes per move. Scores are $100(W + 0.5D)/N$. A game that the actor does not finish (because it exhausts its model-call or output budget, a tool call times out, or it returns no action) counts as a loss, here and in Go and Hex, as does a chess game that reaches the move limit. Appendix Table 2 specifies two chess difficulty profiles: Hard, our primary comparison with more headroom, and Easy, which tests whether weaker models improve in an easier regime. Both include held-out evaluation against stronger opponents.

We evaluate Claude Fable 5, Claude Opus 5, Claude Opus 4.8, Claude Opus 4.6, GPT-5.5, Gemini 3.1 Pro, GPT-5.6 Sol, and GPT-5.6 Luna. To manage cost, we use a subset of models spanning capability levels for Go and Hex. For NetHack, we use the models available through Claude Code and Codex, effectively using native Claude Code and Codex as the starting harness, which is helpful for hard games like NetHack. Between checkpoints, the model may edit `CLAUDE.md`/`AGENTS.md`, skills (`SKILL.md`), Python tools, strategies, and memory. Actor and reflection calls use the same model within a run. All chess, Go, and Hex performance and learning-resource comparisons cover checkpoints 0–20; failure analyses pool checkpoints 1–20. NetHack covers checkpoints 0–15 and adds Claude Opus 5.5 and Claude Sonnet 5: at each checkpoint, every lineage plays the same ten fresh games, and reflection on all ten produces the next inventory (Appendix D).

3.2 PERSISTENT IMPROVEMENT VARIES ACROSS MODELS

Figure 3 and the top row of Appendix Figure 9 show substantial differences between chess lineages. Claude Fable 5 and Claude Opus 5 reach the highest scores: in Hard chess, their held-out ID scores rise from 37.5% and 25.0% at checkpoint 0 to 73.3% and 66.4% averaged over checkpoints 16–20. GPT-5.6 Sol starts near 0% but gains about as much (to 36.9%), so it trails in absolute score rather than in improvement. On the combined three-game curve its P^{sat} is the highest (Section 3.4).

Claude Opus 4.8 improves modestly and mainly in the Easy condition (0% to 19.0%). Several other models remain close to their initial performance, and a few decline, despite repeated feedback and opportunities to revise persistent artifacts; the same holds for five of the six NetHack lineages (Section 3.3). The common learning interface therefore permits improvement but does not ensure it.

For the strongest improving lineages, held-out ID gains broadly accompany training gains. Because held-out trajectories and scores are unavailable to reflection, these results provide evidence that the resulting artifacts are useful beyond the interactions supplied to the update process. Gains against stronger held-out OOD opponents are generally smaller or less consistent. Improving within the training difficulty regime and transferring beyond it are therefore distinct outcomes. These comparisons characterize the complete learning process; they do not isolate the contribution of experience from additional artifact-development computation.

3.3 The Phenomenon Extends Across Games, but Is Task Dependent

We apply the same protocol to 5×5 Go, 6×6 Hex, and NetHack, a long-horizon, partially observed single-agent game. In Go (Appendix Figure 9, middle row), Claude Fable 5's held-out ID score rises from 20% to 80% and its held-out OOD score from 0% to 50% between checkpoints 0 and 20. In Hex (Appendix Figure 9, bottom row), GPT-5.6 Sol improves from 40% to 77.5% on held-out ID and from 28.1% to 78.1% on held-out OOD.

However, improvement is not uniform across tasks: Claude Opus 4.8 loses ground in Go while improving substantially in Hex. Claude Opus 5 reaches 50% on held-out ID in Go at checkpoint 2 and stays at 45–50% through checkpoint 19, but drops to 0% at checkpoint 20, when a rewrite of its move-selection tool makes it lose as Black as well as White.

In NetHack (Figure 5), only Claude Opus 5.5 shows a reliable rise: its mean score grows from 2,019 over checkpoints 0–2 to 6,694 over checkpoints 13–15, saturating at checkpoint 11 after about \$1,070 of learning cost ($P^{\mathrm{sat}} = 60$). Claude Opus 5 and Claude Opus 4.8 rise less and not reliably (1,508 to 2,008 and 705 to 1,052), Claude Sonnet 5 and GPT-5.6 Luna decline, and GPT-5.6 Sol, the most plastic model on the combined board-game curve (Section 3.4), stays flat (571 to 534). These experiments extend the phenomenon beyond chess without establishing a task-independent model ranking. Go evaluates 20 training, 20 held-out ID, and 12 held-out OOD games per checkpoint; Hex evaluates 40, 40, and 32, respectively. Appendices C and G.2 give setup details and further analyses.

Finding 1. Persistent self-improvement is model and task dependent. A common interface for feedback, reflection, and artifact construction produces sharply different learning curves in all four games, and transfer to stronger opponents (i.e., generalization) is not uniform.

3.4 Acquisition Efficiency Differs from Endpoint Capability

Checkpoint curves compare agents after equal numbers of update opportunities, but these opportunities incur different learning costs. Figure 1 instead relates held-out ID performance to cumulative learning cost, both averaged equally across chess (Hard and Easy), Go, and Hex for models evaluated in all three games. Fitted curves estimate the cost C^* of reaching 90% of the improvement, and P^{sat} measures the corresponding acquisition efficiency (Section 2.2).

On the combined three-game curve, GPT-5.6 Sol has the highest estimated P^{sat} at 298 percentage points per \$1,000 of mean per-game learning cost, followed by Claude Opus 5 at 233, Claude Fable 5 at 57, and Claude Opus 4.8 at 14; GPT-5.6 Luna shows no reliable rise under the specified criterion and is assigned $P^{\mathrm{sat}} = 0$. GPT-5.5 is still improving slowly at the end of its runs, so its saturation, and hence its P^{sat}, cannot be established from these runs. Claude Fable 5 reaches the highest fitted performance level in our experiments, showing a clear separation between endpoint capability and acquisition efficiency for Claude Fable 5 and GPT-5.6 Sol. NetHack costs use cache-aware prices and its scores are rescaled to the observed score range, so we estimate its plasticity separately (Figure 5d): Claude Opus 5.5 has $P^{\mathrm{sat}} = 60$ points per \$1,000 and the other five lineages show no reliable rise, although GPT-5.6 Sol and Claude Sonnet 5 spend 4 and 6.5 times as much as Claude Opus 5.5 through checkpoint 15.

Additional endpoint and budget-dependent comparisons appear in Figures 20 and 21. Efficiency estimates depend on provider pricing (e.g., Table 3 for NetHack) and saturation-fit assumptions, and

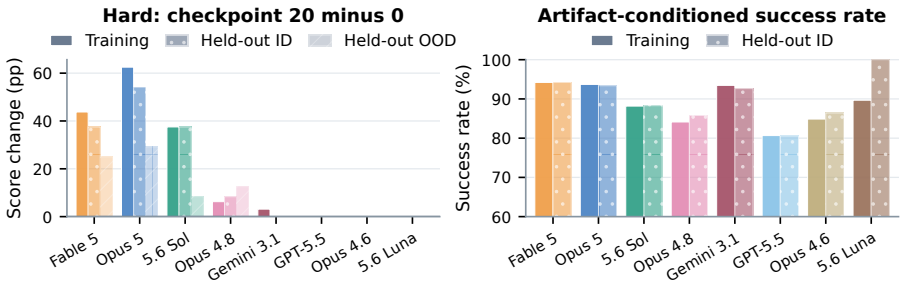


Figure 6: **Success when an artifact is used does not determine game-level improvement.** Hard chess condition. Left: checkpoint-20 score minus checkpoint-0 score. Right: artifact-conditioned success rate, i.e., the share of moves with no identified failure among moves where a relevant artifact was used (checkpoints 1–20 pooled). This rate can be high even when artifacts are rarely used. Easy chess appears in Appendix Figure 27.

extrapolated saturation adds uncertainty. We therefore interpret plasticity as a protocol-dependent measure, not a universal model ranking.

Finding 2. Endpoint capability and acquisition efficiency answer different questions. For example, Claude Fable 5 reaches the highest fitted performance on the combined-game curve, while GPT-5.6 Sol has the largest estimated improvement per unit learning cost under the saturation metric.

3.5 Where Does the Improvement Loop Break?

We pool counts over checkpoints 1–20 of each lineage before computing rates. Figures 4 and 28 distinguish failure composition (right), which conditions on identified failures, from decision-level reuse (left), which also includes successful decisions. Each relevant decision contributes once: unused alternatives do not count as missed reuse when another relevant artifact was used.

Low reuse accompanies weak improvement. Pooled over Hard and Easy chess, GPT-5.6 Luna takes about 98% of relevant decisions without artifact use, Gemini 3.1 Pro about 87%, Claude Opus 4.6 about 51%, and GPT-5.5 about 28%, compared with 2–6% for Claude Fable 5, Claude Opus 5, Claude Opus 4.8, and GPT-5.6 Sol (Figures 4 and 28, left). The Go and Hex rows add examples, including GPT-5.6 Luna's about 1% reuse rate in Go (Appendix Figure 28). Across 19 model–game cells in Chess (Hard), Go, and Hex, held-out ID reuse correlates with the gain in held-out ID score from checkpoint 0 to checkpoints 16–20 (Spearman $\rho = 0.73$; 0.62 after excluding Gemini 3.1 Pro and GPT-5.6 Luna). This association identifies reuse as a candidate bottleneck.

High reuse does not explain the remaining performance gap. Figure 6 separates how often artifacts are used from how well they work when used (the artifact-conditioned success rate): an artifact can work when used yet be ignored at most relevant moves. Conversely, Claude Fable 5, Claude Opus 5, Claude Opus 4.8, and GPT-5.6 Sol all reuse artifacts at 94–98% of relevant chess decisions (pooled over both conditions) but attain substantially different game scores. Neither the reuse rate nor the artifact-conditioned success rate explains this gap. Artifact content, how actors configure and apply it, and whether it helps at decisions consequential for winning remain possible explanations. The observational labels do not separate these mechanisms.

Strong improvers still fail when using artifacts. The models that use artifacts at almost every relevant chess decision, including the strongest improvers, fail less often overall (about 0.16 versus 0.38 identified failures per relevant decision for the other models), but most of their remaining failures (83–99%) happen while a relevant artifact is in use (Figure 4, right; Appendix Figure 33). Using an artifact therefore does not guarantee success: the artifact may not generalize to the position, or the agent may apply it poorly. We leave separating these causes to future work.

Figure 7 and Appendix Figures 29 and 30 show that reuse can rise before performance peaks. In Hard chess, GPT-5.6 Sol and Claude Opus 4.8 move from below 20% reuse at checkpoint 1 to 93–99% at checkpoint 3, while their held-out ID scores peak only at checkpoints 12 and 5, respectively.

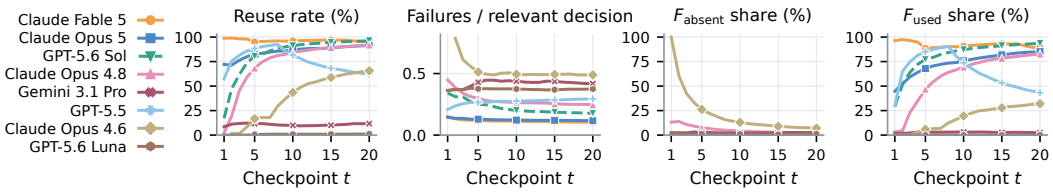


Figure 7: **Dynamics across checkpoints do not determine game-level improvement.** Hard chess. Each point pools training and held-out ID decisions over checkpoints $1, \ldots, t$ of each lineage; columns show decision-level reuse, identified failures per relevant decision (a move where at least one saved artifact is relevant to choosing or playing it, used or not), and the shares of failures with absent and used covering artifacts. Easy chess, Go, and Hex appear in Appendix Figures 33 and 41.

Reuse patterns can also change without improvement. Pooled over both chess conditions, the share of Claude Opus 4.6's held-out ID failures that happen despite artifact use rises from 6.5% at checkpoints 1–5 to 52% at checkpoints 16–20, while its failures per relevant decision (moves where a saved artifact is relevant to choosing or playing the move, used or not) stay flat (0.42 to 0.41). In the Hard chess condition, GPT-5.5 uses artifacts at 96–100% of relevant decisions at checkpoints 2–8 but at only 32–50% from checkpoint 9 on, although its artifacts did not change. A change in failure types is thus not by itself a sign of progress, and an agent can stop using artifacts it once used reliably.

Failure regimes differ across games. In Go and Hex, failures remain distributed across all three categories (Appendix Figure 28). Broad artifact relevance does not imply failure coverage: although relevant artifacts are identified in at least 99% of decisions in 25 of 27 lineages, 67–100% of absent-artifact failures in Go and Hex occur at decisions with a broadly relevant artifact.

Local diagnostic gains can also diverge from task performance. In Go, Claude Opus 4.8 raises its share of relevant decisions with successful reuse from 59–65% to 93%, while its held-out ID score falls from 40% at checkpoint 0 to 10–25% over checkpoints 15–20 (Appendix Figure 38). Reuse, failure composition, and task performance therefore capture distinct aspects of persistent learning.

Finding 3. Artifact reuse is associated with improvement, but is not a proxy for capability. High reuse may still leave significant failures, hinting at problems with artifact quality, generality, or use.

Concrete Examples of Persistent Learning. What an agent carries forward is code and notes that it can reuse later. For example, one Claude Fable 5 lineage first writes its own chess engine, a program that looks ahead to pick moves, without outside libraries. It then improves the engine after specific mistakes. After a mistake that left its king under attack, it adds a search for moves that get the king out of check. After throwing away winning positions, it changes how it handles repeated positions, since repeating a position can end the game in a draw. After failing to win an endgame while ahead, it gives the engine more search time in endgames. It also writes a helper that plays several moves per call under a fixed total search budget, so its tools do not time out. Every later game starts with all of these changes. Figure 8 shows one such fix in chess and one in NetHack.

Writing down an instruction is not the same as following it. In one GPT-5.5 run, the reflection step keeps telling the agent to load its saved move-picking program right away, yet the agent still spends its first steps listing and reading files. These examples show how artifacts and behavior change over time; they do not show which single change caused an improvement.

4 Discussion and Limitations

Plasticity is not simply context management. Long-context capabilities may aid reflection, artifact construction, and reuse, but plasticity measures whether they produce persistent held-out gains relative to learning cost. Our fixed-weight setting includes executable tools and skills, not just textual context. High reuse with differing task performance shows that retrieval alone does not explain plasticity, although we do not isolate the role of long-context instruction following. Plasticity depends on the learning protocol; future settings could also permit parameter updates.

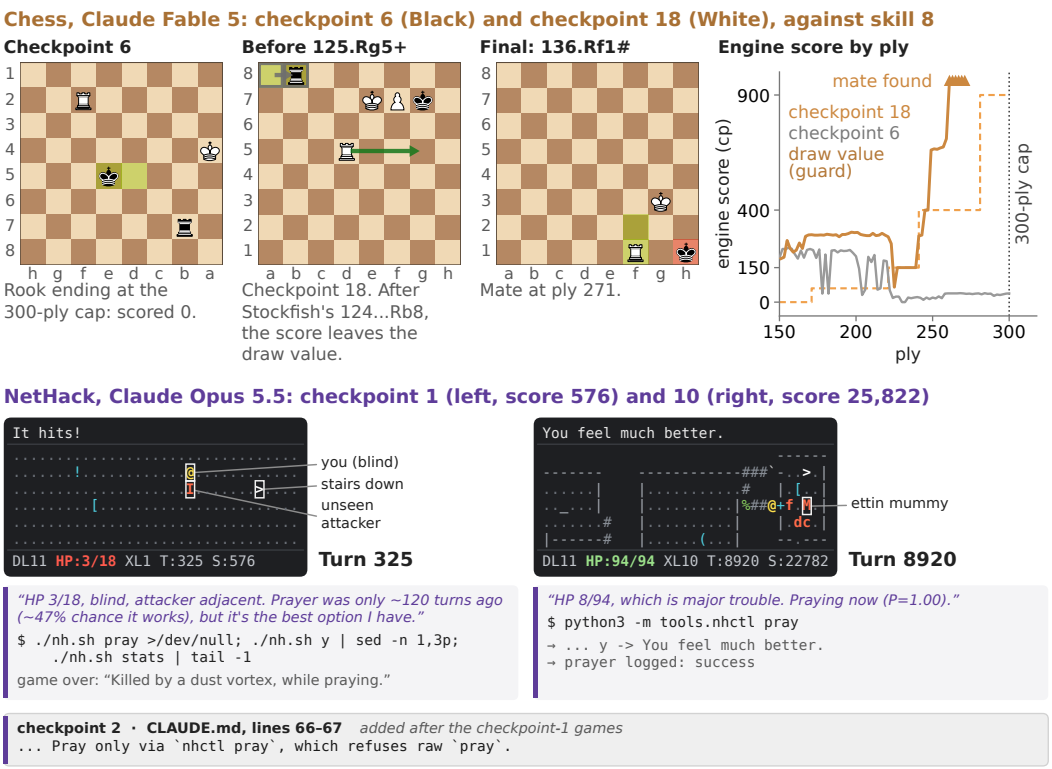


Figure 8: **Fixes that persist after a failure.** Top (chess): after a checkpoint-6 game is scored 0 at the 300-ply limit, Claude Fable 5 makes its self-made engine value draws more as the limit nears; at checkpoint 18 the engine holds a draw until Stockfish errs, then mates. Bottom (NetHack): after dying at checkpoint 1 while praying through raw keystrokes, Claude Opus 5.5 adds a rule to pray only through its own controller; at checkpoint 10 the prayer heals it. Details in Figures 16 and 12.

Different bottlenecks require different interventions. When relevant artifacts are ignored, more reliable deployment may help; when reuse is frequent, procedure quality and application become natural targets. These are observational hypotheses, not causal conclusions. Controlled artifact removal, replacement, or automatic invocation could test them. Validation should also assess behavioral effectiveness, not merely executable validity.

Persistent adaptation is isolated, but the contribution of experience is not. Fresh contexts and frozen weights restrict adaptation to artifacts, but do not separate the benefits of training feedback from artifact-development computation. Budget-matched reflection without feedback and controlled actor–artifact comparisons could disentangle these effects. Our cost measure excludes deployment costs, and the fixed reflection protocol does not test unrestricted recursive self-modification.

Generalization and measurement remain limited. Chess, Go, and Hex offer controlled feedback, while NetHack adds long-horizon, partially observed interactions. Separately trained lineages do not establish cross-task transfer. Small evaluation sets add checkpoint noise, and plasticity estimates depend on score headroom, learning horizon, saturation fits, and provider pricing. These limitations motivate reporting full learning curves rather than fine-grained scalar comparisons.

5 Conclusion

We introduced *agent plasticity* to measure how efficiently agents convert experience into persistent held-out capability. Across models and tasks, plasticity varies substantially and reveals differences that endpoint performance obscures. Our diagnostics show that artifact reuse accompanies improvement but does not guarantee it. Evaluating long-lived agents therefore requires measuring not only what they can accomplish, but how efficiently experience makes them better.

AI USE STATEMENT

Large language model based digital agents are the object of study in this work: every agent is a frontier model accessed through a provider API (Section 3.1). An LLM is also used as a measurement instrument. The failure and artifact-reuse labels of Section 3.5 combine semantic judgments by an LLM analyzer, Gemini 3.1 Pro, with deterministic execution evidence (Section 2.3; Appendix H), and these failure and artifact-reuse labels can be considered as synthetic data. LLMs were used to assist in implementing code which was reviewed by one author, and for polishing writing including checking grammar. LLMs and generative AI tools were not used to develop theoretical models or conceptual frameworks, formulate mathematical claims, provide critical ingredients for proving mathematical claims, assist in the writing of proofs, propose or refine hypotheses, support qualitative and thematic data analysis, design or provide feedback on research methodology or experiments, assist with translation, clean or reformat datasets, or interpret results. We have reviewed all AI-assisted work and take responsibility for the final content of this paper, including text, claims, and artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work involves no human subjects and no personal data. Persistent self-improvement can make agents more useful by allowing them to adapt to local environments, recurring tasks, user preferences, and operational constraints without repeated model training. The same property can amplify undesirable behavior if an agent stores and reuses unsafe procedures, overfits to misleading feedback, or acquires persistent strategies that escape ordinary prompt-level oversight.

A measurement framework for plasticity is therefore relevant to safety as well as capability. A highly plastic agent may change materially after deployment even when its model weights are unchanged. Evaluation of long-lived systems should consequently consider not only their initial behavior, but the range and rate of behavioral change induced by realistic experience streams. Controlled held-out evaluations, persistent state auditing, artifact validation, and provenance of updates become increasingly important as agents are given longer lifetimes and more writable artifacts.

Our experiments use isolated environments, inaccessible held-out evaluations, bounded persistent files, and validation gates precisely to keep improvement attributable and auditable. These controls are not a complete safety-promoting solution, but they illustrate the kind of instrumentation needed to study adaptive agents responsibly.

REPRODUCIBILITY STATEMENT

Our setup is simple as far as reproducibility is concerned (modulo changes in API-based models): A frozen model is given a prompt and may modify its own harness, i.e., the persistent tools, strategies, and memory files that later instances inherit (Section 2). The full actor and reflection prompts are given in Appendix K, and the failure-analysis prompts in Appendix H (Table 5). Game rules, engine and environment versions (Stockfish 16, GNU Go, our native Hex engine, and NetHack Challenge v0), opponent levels, and the number of games per checkpoint are listed in Appendix C (Table 2); models and learning-cost accounting are described in Section 3.1 and Appendix G.

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, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. GEPA: Reflective prompt evolution can outperform reinforcement learning. In *International Conference on Learning Representations*, 2026.

Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. Large language models as tool makers. In *International Conference on Learning Representations*, 2024.

Shibhansh Dohare, J. Fernando Hernandez-Garcia, Qingfeng Lan, Parash Rahman, A. Rupam Mahmood, and Richard S. Sutton. Loss of plasticity in deep continual learning. *Nature*, 632(8026): 768–774, 2024. doi: 10.1038/s41586-024-07711-7.

Shihan Dou, Ming Zhang, Chenhao Huang, Jiayi Chen, Feng Chen, Shichun Liu, Yan Liu, Chenxiao Liu, Cheng Zhong, Zongzhang Zhang, Tao Gui, Chao Xin, Chengzhi Wei, Lin Yan, Qi Zhang, and Xuanjing Huang. EvaLearn: Quantifying the learning capability and efficiency of LLMs via sequential problem solving. In *Advances in Neural Information Processing Systems*, volume 38, pp. 139558–139611, 2025. doi: 10.52202/085713-4197.

Chrisantha Fernando, Dylan Sunil Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. Promptbreeder: Self-referential self-improvement via prompt evolution. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 13481–13544. PMLR, 2024.

Yufei He, Juncheng Liu, Yue Liu, Yibo Li, Tri Cao, Zhiyuan Hu, Xinxing Xu, and Bryan Hooi. EvoTest: Evolutionary test-time learning for self-improving agentic systems. In *International Conference on Learning Representations*, 2026.

Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, Yang Zhang, and Boris Ginsburg. RULER: What's the real context size of your long-context language models? In *Conference on Language Modeling (COLM)*, 2024.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In *International Conference on Learning Representations*, 2025.

Yuanzhe Hu, Yu Wang, and Julian McAuley. Evaluating memory in LLM agents via incremental multi-turn interactions. In *International Conference on Learning Representations*, 2026.

Daniel Jaroslawicz, Brendan Whiting, Parth Shah, and Karime Maamari. How many instructions can LLMs follow at once? *arXiv preprint arXiv:2507.11538*, 2025. doi: 10.48550/arXiv.2507.11538.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. DSPy: Compiling declarative language model calls into state-of-the-art pipelines. In *International Conference on Learning Representations*, 2024.

Heinrich Küttler, Nantas Nardelli, Alexander H. Miller, Roberta Raileanu, Marco Selvatici, Edward Grefenstette, and Tim Rocktäschel. The NetHack learning environment. In *Advances in Neural Information Processing Systems*, volume 33, 2020.

Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-Harness: End-to-end optimization of model harnesses. *arXiv preprint arXiv:2603.28052*, 2026. doi: 10.48550/arXiv.2603.28052.

Tianle Li, Ge Zhang, Quy Duc Do, Xiang Yue, and Wenhu Chen. Long-context LLMs struggle with long in-context learning. *Transactions on Machine Learning Research*, 2025. URL `https://openreview.net/forum?id=Cw2xlg0e46`.

Huawei Lin, Peng Li, Jie Song, Fuxin Jiang, and Tieying Zhang. MUSE-Autoskill: Self-evolving agents via skill creation, memory, management, and evaluation. *arXiv preprint arXiv:2605.27366*, 2026. doi: 10.48550/arXiv.2605.27366.

Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173, 2024. doi: 10.1162/tacl_a_00638.

Yitao Liu, Chenglei Si, Karthik R. Narasimhan, and Shunyu Yao. Contextual experience replay for self-improvement of language agents. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 14179–14198. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.acl-long.694.

Clare Lyle, Zeyu Zheng, Evgenii Nikishin, Bernardo Avila Pires, Razvan Pascanu, and Will Dabney. Understanding plasticity in neural networks. In *Proceedings of the 40th International Conference on Machine Learning*, 2023.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-Refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems*, volume 36, pp. 46534–46594, 2023. doi: 10.52202/075280-2019.

Ali Modarressi, Hanieh Deilamsalehy, Franck Dernoncourt, Trung Bui, Ryan A. Rossi, Seunghyun Yoon, and Hinrich Schütze. NoLiMa: Long-context evaluation beyond literal matching. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 44554–44570. PMLR, 2025.

Evgenii Nikishin, Max Schwarzer, Pierluca D'Oro, Pierre-Luc Bacon, and Aaron Courville. The primacy bias in deep reinforcement learning. In *Proceedings of the 39th International Conference on Machine Learning*, 2022.

Siru Ouyang, Jun Yan, I-Hung Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long T. Le, Samira Daruki, Xiangru Tang, Vishy Tirumalashetty, George Lee, Mahsan Rofouei, Hangfei Lin, Jiawei Han, Chen-Yu Lee, and Tomas Pfister. ReasoningBank: Scaling agent self-evolving with reasoning memory. In *International Conference on Learning Representations*, 2026.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. MemGPT: Towards LLMs as operating systems. *arXiv preprint arXiv:2310.08560*, 2023. doi: 10.48550/arXiv.2310.08560.

Davide Paglieri, Bartłomiej Cupiał, Samuel Coward, Ulyana Piterbarg, Maciej Wolczyk, Akbir Khan, Eduardo Pignatelli, Łukasz Kuciński, Lerrel Pinto, Rob Fergus, Jakob Nicolaus Foerster, Jack Parker-Holder, and Tim Rocktäschel. BALROG: Benchmarking agentic LLM and VLM reasoning on games. In *International Conference on Learning Representations*, 2025.

Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology (UIST)*, 2023. doi: 10.1145/3586183.3606763.

Cheng Qian, Chi Han, Yi Fung, Yujia Qin, Zhiyuan Liu, and Heng Ji. CREATOR: Tool creation for disentangling abstract and concrete reasoning of large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 6922–6939, Singapore, 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-emnlp.462.

Maxime Robeyns, Martin Szummer, and Laurence Aitchison. A self-improving coding agent. *arXiv preprint arXiv:2504.15228*, 2025. doi: 10.48550/arXiv.2504.15228.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, pp. 8634–8652, 2023. doi: 10.52202/075280-0377.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419):1140–1144, 2018. doi: 10.1126/science.aar6404.

Mirac Suzgun, Mert Yuksekgonul, Federico Bianchi, Dan Jurafsky, and James Zou. Dynamic cheatsheet: Test-time learning with adaptive memory. In *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7080–7106, Rabat, Morocco, 2026. Association for Computational Linguistics. doi: 10.18653/v1/2026.eacl-long.333.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *Transactions on Machine Learning Research*, 2024. URL https://openreview.net/forum?id=ehfRiF0R3a.

Wenyi Wang, Piotr Piękos, Nanbo Li, Firas Laakom, Yimeng Chen, Mateusz Ostaszewski, Mingchen Zhuge, and Jürgen Schmidhuber. Huxley-Gödel machine: Human-level coding agent development by an approximation of the optimal self-improving machine. In *International Conference on Learning Representations*, 2026a.

Zefeng Wang, Minxi Yan, Jinhe Bi, Sikuan Yan, Volker Tresp, and Yunpu Ma. MetaSkill-Evolve: Recursive self-improvement of LLM agents via two-timescale meta-skill evolution. *arXiv preprint arXiv:2607.05297*, 2026b.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 63897–63911. PMLR, 2025.

Cheng-Kuang Wu, Zhi Rui Tam, Chieh-Yen Lin, Yun-Nung Chen, and Hung-yi Lee. StreamBench: Towards benchmarking continuous improvement of language agents. In *Advances in Neural Information Processing Systems (Datasets and Benchmarks Track)*, 2024. doi: 10.52202/079017-3398.

Di Wu, Hongwei Wang, Wenhao Yu, Yuwei Zhang, Kai-Wei Chang, and Dong Yu. LongMemEval: Benchmarking chat assistants on long-term interactive memory. In *International Conference on Learning Representations*, 2025.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-Mem: Agentic memory for LLM agents. In *Advances in Neural Information Processing Systems*, volume 38, pp. 20004–20031, 2025. doi: 10.52202/085713-0593.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *International Conference on Learning Representations*, 2024.

Howard Yen, Tianyu Gao, Minmin Hou, Ke Ding, Daniel Fleischer, Peter Izsak, Moshe Wasserblat, and Danqi Chen. HELMET: How to evaluate long-context models effectively and thoroughly. In *International Conference on Learning Representations*, 2025.

Xunjian Yin, Xinyi Wang, Liangming Pan, Li Lin, Xiaojun Wan, and William Yang Wang. Gödel agent: A self-referential agent framework for recursively self-improvement. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 27890–27913, Vienna, Austria, 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.acl-long.1354.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Pan Lu, Zhi Huang, Carlos Guestrin, and James Zou. Optimizing generative AI by backpropagating language model feedback. *Nature*, 639(8055):609–616, 2025. doi: 10.1038/s41586-025-08661-4.

Eric Zelikman, Eliana Lorch, Lester Mackey, and Adam Tauman Kalai. Self-taught optimizer (STOP): Recursively self-improving code generation. In *Conference on Language Modeling (COLM)*, 2024.

Alex L. Zhang, Tim Kraska, and Omar Khattab. Recursive language models. *arXiv preprint arXiv:2512.24601*, 2025. doi: 10.48550/arXiv.2512.24601.

Haozhen Zhang, Quanyu Long, Jianzhu Bao, Tao Feng, Weizhi Zhang, Haodong Yue, and Wenya Wang. MemSkill: Learning and evolving memory skills for self-evolving agents. *arXiv preprint arXiv:2602.02474*, 2026a.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Tjarko Lange, and Jeff Clune. Darwin Gödel machine: Open-ended evolution of self-improving agents. In *International Conference on Learning Representations*, 2026b.

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 *International Conference on Learning Representations*, 2026c.

Shengtao Zhang, Jiaqian Wang, Ruiwen Zhou, Junwei Liao, Yuchen Feng, Zhuo Li, Yujie Zheng, Weinan Zhang, Ying Wen, Zhiyu Li, Feiyu Xiong, Yutao Qi, Bo Tang, and Muning Wen. MemRL: Self-evolving agents via runtime reinforcement learning on episodic memory. *arXiv preprint arXiv:2601.03192*, 2026d.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExpeL: LLM agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 19632–19642, 2024. doi: 10.1609/aaai.v38i17.29936.

Boyuan Zheng, Michael Y. Fatemi, Xiaolong Jin, Zora Zhiruo Wang, Apurva Gandhi, Yueqi Song, Yu Gu, Jayanth Srinivasa, Gaowen Liu, Graham Neubig, and Yu Su. SkillWeaver: Web agents can self-improve by discovering and honing skills. *arXiv preprint arXiv:2504.07079*, 2025a. doi: 10.48550/arXiv.2504.07079.

Junhao Zheng, Xidi Cai, Qiuke Li, Duzhen Zhang, ZhongZhi Li, Yingying Zhang, Le Song, and Qianli Ma. LifelongAgentBench: Evaluating LLM agents as lifelong learners. *arXiv preprint arXiv:2505.11942*, 2025b. doi: 10.48550/arXiv.2505.11942.

Zijian Zhou, Ao Qu, Zhaoxuan Wu, Sunghwan Kim, Alok Prakash, Daniela Rus, Bryan Kian Hsiang Low, and Paul Liang. Mem1: Learning to synergize memory and reasoning for efficient long-horizon agents. In C. Vondrick, B. Hariharan, C. Raffel, L. Pinto, D. Yang, and A. Faust (eds.), *International Conference on Learning Representations*, volume 2026, pp. 58413–58438, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/5fc8b3bdfbb9167b5144df5d3fae4616-Paper-Conference.pdf.

Adam Zweiger, Jyo Pari, Han Guo, Yoon Kim, and Pulkit Agrawal. Self-adapting language models. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, Main Conference, pp. 74084–74115. Curran Associates, Inc., 2025. doi: 10.52202/085713-2483. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/6b41e04c41726e2a60e456d0a2b961ab-Paper-Conference.pdf.

A Appendix

This appendix gives supplementary experiment details, prompts (Appendix K), additional results, and qualitative examples.

Settings for all games are in Appendix C and Go and Hex learning curves in Appendix G.2. Figure 9 reports checkpoint performance in the Easy chess condition, Go, and Hex, complementing Figure 3.

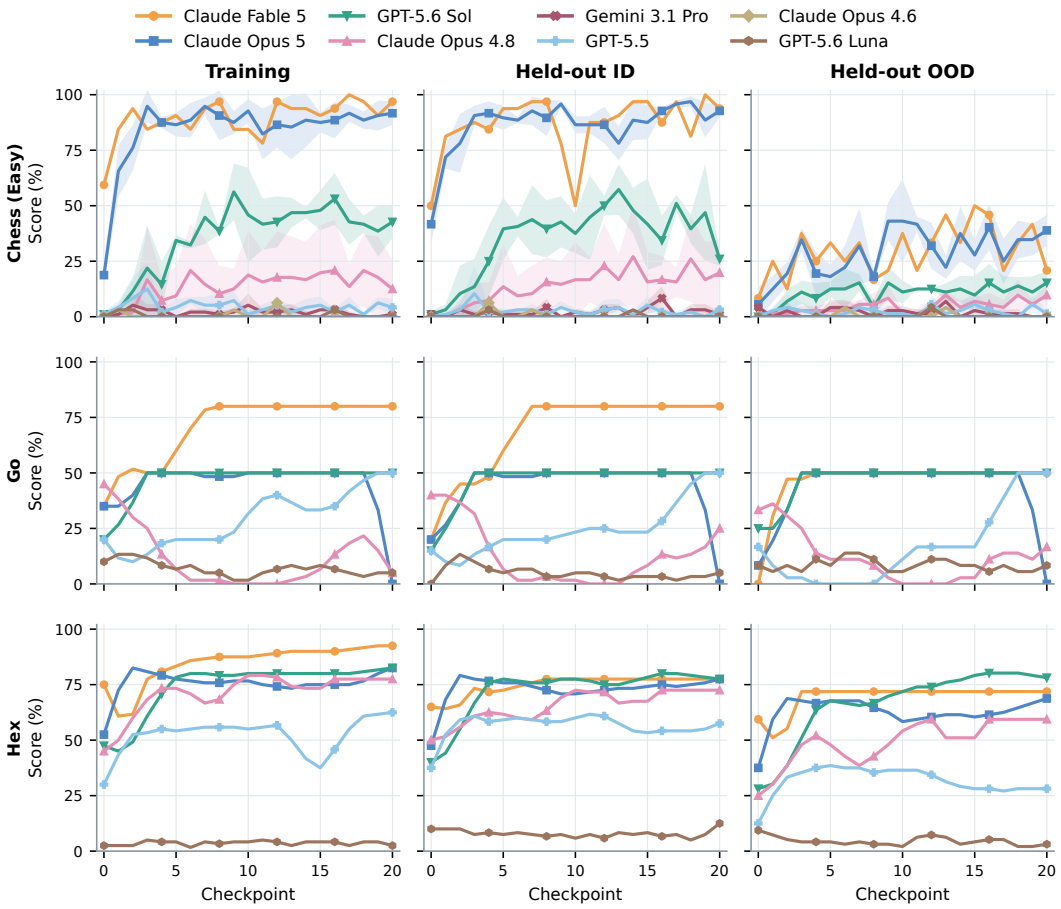


Figure 9: **Checkpoint performance in the Easy chess condition, Go, and Hex.** Checkpoint performance, measured as $100(W+0.5D)/N$, in the Easy chess condition (top), Go (middle), and Hex (bottom), with columns as in Figure 3.

B Related Work

Learning from experience through persistent artifacts. A growing body of work enables language agents to improve without updating their model weights. Reflexion stores feedback as episodic reflections (Shinn et al., 2023), ExpeL extracts reusable knowledge from past experiences (Zhao et al., 2024), and Agent Workflow Memory and Contextual Experience Replay distill trajectories into reusable workflows and memories (Wang et al., 2025; Liu et al., 2025). Dynamic Cheatsheet and ReasoningBank accumulate strategies across interactions (Suzgun et al., 2026; Ouyang et al., 2026), while memory architectures organize persistent information for future retrieval (Park et al., 2023; Xu et al., 2025). Other approaches construct executable artifacts: LATM and CREATOR generate reusable tools, while Voyager and SkillWeaver develop libraries of skills from environment interaction (Cai et al., 2024; Qian et al., 2023; Wang et al., 2024; Zheng et al., 2025a). These methods establish mechanisms for retaining and exploiting experience. Our objective is comple-

Table 1: **Positioning agent plasticity relative to prior work.** Related approaches study complementary aspects of adaptation. Our evaluation combines persistent self-constructed artifacts, cost-normalized held-out improvement, and artifact-level diagnostics under a common fixed-weight learning protocol.

Research direction	Primary focus	Our emphasis
Reflection, memory, and skills (Shinn et al., 2023; Wang et al., 2024)	Mechanisms for retaining and reusing experience.	Measuring how efficiently different models construct and exploit persistent artifacts.
Context and harness optimization (Zhang et al., 2026c; Agrawal et al., 2026)	Optimizing prompts, context, or agent implementations.	Evaluating a model that acts and improves its own future behavior under a common protocol.
Long-context evaluation (Hsieh et al., 2024; Yen et al., 2025)	Retrieving, reasoning over, and following information in long contexts.	Measuring durable capability gains from self-constructed artifacts, including executable tools.
Sequential learning benchmarks (Dou et al., 2025; He et al., 2026)	Measuring improvement and, in some settings, learning efficiency across interactions.	Combining cost-normalized held-out learning curves with artifact-level failure diagnostics.
Parametric adaptation (Zweiger et al., 2025)	Persistent improvement through model-weight updates.	Isolating non-parametric adaptation through persistent agent artifacts.

mentary: we measure how effectively different models construct and reuse persistent artifacts under a common learning protocol, and whether those artifacts improve future held-out performance.

Self-evolving agents and harness optimization. Recent work increasingly treats the agent itself as an object of optimization. STOP, ADAS, Gödel Agent, the Darwin Gödel Machine, and related systems modify or search over agent implementations (Zelikman et al., 2024; Hu et al., 2025; Yin et al., 2025; Zhang et al., 2026b; Wang et al., 2026a; Robeyns et al., 2025). MemSkill, MUSE-Autoskill, MemRL, and MetaSkill-Evolve extend adaptation to memory operations, skill libraries, and improvement mechanisms (Zhang et al., 2026a; Lin et al., 2026; Zhang et al., 2026d; Wang et al., 2026b). Related optimization methods, including DSPy, OPRO, Promptbreeder, TextGrad, and GEPA, improve prompts or language-model programs through feedback (Khattab et al., 2024; Yang et al., 2024; Fernando et al., 2024; Yuksekgonul et al., 2025; Agrawal et al., 2026). ACE evolves contextual playbooks, while Meta-Harness extends optimization to the surrounding agent implementation (Zhang et al., 2026c; Lee et al., 2026).

These approaches primarily investigate mechanisms for obtaining improved agents. We instead hold the improvement interface fixed and measure how efficiently different models improve their own future behavior. The same model performs acting and reflection within each lineage, and improvement is evaluated throughout the learning process rather than only at its endpoint. Plasticity therefore complements research on self-evolving agents by characterizing the efficiency of persistent adaptation under a specified learning protocol.

Long-context capabilities and context management. Long-context evaluations examine models' ability to retrieve, reason over, and follow instructions distributed across large inputs (Liu et al., 2024; Hsieh et al., 2024; Yen et al., 2025; Modarressi et al., 2025; Li et al., 2025; Jaroslawicz et al., 2025). Context-management methods address these limitations through memory hierarchies, consolidation, recursive processing, and evolving playbooks (Packer et al., 2023; Zhou et al., 2026; Zhang et al., 2025; 2026c).

Context management is a potential mechanism underlying plasticity, but the two measure different properties. Context-management evaluations typically examine how effectively information is retained, selected, or used. Plasticity measures whether experience produces persistent capability gains relative to learning cost. Our agents must construct and manage their own artifacts, including executable tools that need not reside in the token context. Moreover, high artifact reuse does not guarantee strong task performance. Our experiments do not, however, isolate the contribution of long-context instruction following to artifact construction or application.

Evaluating learning from experience. Several benchmarks move beyond static capability evaluation. StreamBench studies continuous improvement from feedback (Wu et al., 2024); EvaLearn evaluates learning capability and efficiency across sequences of related problems (Dou et al., 2025); and LifelongAgentBench examines knowledge accumulation across interdependent tasks (Zheng et al., 2025b). EvoTest evaluates test-time learning through repeated text-adventure episodes and evolving agent configurations (He et al., 2026). Memory benchmarks such as LongMemEval and MemoryAgentBench examine complementary aspects of long-term retrieval and learning (Wu et al., 2025; Hu et al., 2026). The NetHack Learning Environment and BALROG provide challenging environments for evaluating long-horizon agents (Küttler et al., 2020; Paglieri et al., 2025).

Our contribution lies in combining three complementary measurements under a common protocol. First, fixed-weight agents act in fresh contexts and inherit only inspectable, self-constructed persistent artifacts. Second, we measure complete held-out learning trajectories and normalize improvements by the cost of generating experience and refining artifacts, distinguishing acquisition efficiency from endpoint capability. Third, we trace failures through artifact construction and reuse, distinguishing artifacts that were absent, available but unused, or used while failure remained. Together, these measurements characterize not only whether agents improve, but how efficiently they do so and where their improvement process breaks down.

Test-time refinement and parametric adaptation. Self-Refine and related methods improve individual outputs through iterative feedback but need not produce changes that persist across fresh interactions (Madaan et al., 2023). Reinforcement learning, self-play, and approaches such as SEAL can instead acquire lasting capabilities through parameter updates (Silver et al., 2018; Zweiger et al., 2025). Our experiments isolate the intermediate regime: improvements must persist across interactions, but model weights remain frozen. This also distinguishes our usage of plasticity from the loss of plasticity studied in continual neural-network training (Nikishin et al., 2022; Lyle et al., 2023; Dohare et al., 2024). We define agent plasticity as a system-level property relative to a learning protocol; while the concept can extend to parametric adaptation, our present measurements deliberately isolate improvement through persistent artifacts.

Table 1 summarizes how our measurement framework complements these research directions.

C EXPERIMENTAL SETUP

Table 2 compares the task, evaluation, and learning-feedback settings used across Chess, Go, Hex, and NetHack, and separates feedback available to the reflector from offline annotations added only for the offline failure analysis. Difficulty scales are game-specific and should not be interpreted as a common Elo scale.

Setting	Task and score	Opponent or difficulty	Games per checkpoint	Training feedback available to reflection	Offline annotation for failure analysis
Chess (Hard)	Standard 8×8 chess; win/draw/loss scores $1/0.5/0$.	Stockfish 16 at 20k nodes/move. Train: skills 1–8; held-out ID: 3–8; held-out OOD: 12.	16 Train, 12 ID, 12 OOD; both colors.	After each training game, a separate skill-20 Stockfish evaluates every actor move. Centipawn loss gives `ok`, `inaccuracy`, `mistake`, or `blunder`.	The analysis normalizes the game, model, and sandbox event streams and carries the recorded Stockfish annotations into failure extraction; no second engine pass is added.
Chess (Easy)	The same board, rules, score, move limit, and annotation procedure.	Stockfish 16 at 20k nodes/move. Train and held-out ID: skills 2–5; held-out OOD: skills 8–10.	16 Train, 16 ID, 12 OOD; both colors.	The same skill-20 counterfactual analysis and centipawn-loss severity labels as Hard.	The same trace normalization and recorded-annotation binding as Hard.
Go	5×5 Go, Chinese area scoring, positional superko, no suicide, komi 9.5; win/loss scores $1/0$.	GNU Go levels 5–9 for Train and ID; level 10 for OOD; 75-action cap.	20 Train, 20 ID, 12 OOD; both colors.	Deterministic descriptive diagnostics: captures, liberties, self-atari, avoidable passing while behind, and area-margin change; no expert severity label.	Frozen KataGo replays each position at 64 visits and refines candidates at 512 visits. Stable chosen-versus-best regret supplies mistake/blunder evidence; raw requests, responses, alternatives, perspective, and uncertain cases are recorded.
Hex	Deterministic 6×6 Hex; connection win/loss scores $1/0$.	Native alpha–beta levels 10k–1M nodes/move for Train and ID; 10M for OOD.	40 Train, 40 ID, 32 OOD; both colors.	Rule-based missed immediate wins, preventable immediate losses, and own/opponent connection-distance changes; immediate tactical errors are `blunder`.	The frozen native engine compares legal actions at a common completed depth, refines likely errors at 1M nodes, and adjudicates threshold-near or high-impact cases at 10M. Exact, proven, stable, uncertain, and not-failure dispositions are preserved.
NetHack	NetHack Challenge v0; raw per-game maximum score, summarized by checkpoint mean, median, and maximum.	Lawful dwarf female Valkyrie on deterministic fresh seeds. The Codex and Claude Code agent interfaces; 10k-step, two-hour campaigns.	10 Train per lineage; no ID or OOD split.	The reflector receives all ten verified public actor traces, terminal reasons, raw score, depth, experience, and action metrics, actor workspace evidence, and prior accepted inventories. No optimal-action oracle is used.	

Table 2: **Experimental setup across the four environments.** Task distributions, opponent difficulty, evaluation-set sizes, training feedback exposed to the reflection loop, and separate offline annotation used for the failure analysis. Only training trajectories reach reflection; held-out trajectories and scores never do. Chess games cover both colors of every listed skill.

Environment interaction. Chess, Go, and Hex are fully observed, alternating-turn board games. The actor may inspect state and legal actions, use a separate reversible analysis board, and then commit one action before the engine responds. Chess always begins from the standard initial position. Go changes deterministic training seeds across checkpoints while keeping held-out seeds fixed. Hex uses fixed banks throughout and begins from deterministic 10-ply openings for red actors and 11-ply openings for blue actors, leaving the actor to move.

NetHack is instead a long-horizon, partially observed, single-agent environment. The native coding agent receives a frozen expert guide and the documented `./nh.sh` interface, which exposes only public ASCII observations, messages, status, inventory, menus, and printable NetHack actions. It may build episode-local tools and use persistent instructions, skills, tools, strategies, and memory accepted after earlier checkpoints. The active protocol imposes no separate model-decision or shell-call ceiling. Each lineage evaluates ten fresh, deterministically seeded Valkyrie games per checkpoint. Each model has one lineage with a 10,000-step ceiling and a two-hour limit on active play (wall-clock time excluding provider retry waits). NetHack has no held-out ID or OOD split. Learning curves over checkpoints 0–15 are in Appendix D.

Feedback comparability. Chess has the strongest feedback inside the learning loop because a separate expert engine estimates the value lost by each chosen move. The Go and Hex reflectors receive the lighter deterministic diagnostics described in Table 2; the stronger KataGo and deeper alpha–beta adjudications are added only in the separate offline failure analysis and therefore cannot improve the agents being evaluated. NetHack records the richest execution telemetry but has no optimal-action oracle. Its checkpoint reports aggregate raw per-game maximum score by mean, median, and maximum, and also retain dungeon depth, experience level, action count, survival time, terminal reason, and death cause.

Across all four environments, only training trajectories, learning-time annotations, traces, and permitted per-episode state are made available to reflection. Where held-out ID and OOD splits exist, they remain evaluation-only. The separate offline artifact/failure attribution analysis is not additional feedback supplied during learning.

D NetHack Learning Curves

Figures 10 and 11 show checkpoints 0–15 of six lineages. At each checkpoint all lineages play the same ten new games, so models are compared on the same games, but changes across checkpoints mix learning with game difficulty.

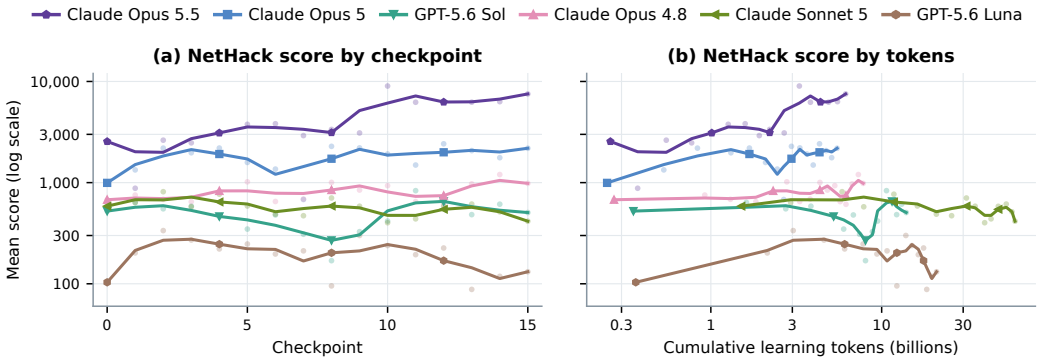


Figure 10: **NetHack learning curves.** Mean score of the ten games at each checkpoint (log scale) against (a) checkpoint and (b) cumulative learning tokens. Lines are three-checkpoint averages; faint markers are the checkpoint means. Only Claude Opus 5.5 clearly improves: its means at checkpoints 10–15 (6,231–9,015) are all above every earlier one (at most 3,820). The other lineages change little relative to how much their checkpoint means vary. Claude Opus 5.5 and Claude Sonnet 5 appear only in NetHack. (Three of the 960 games were replaced by another starting game after an infrastructure failure.)

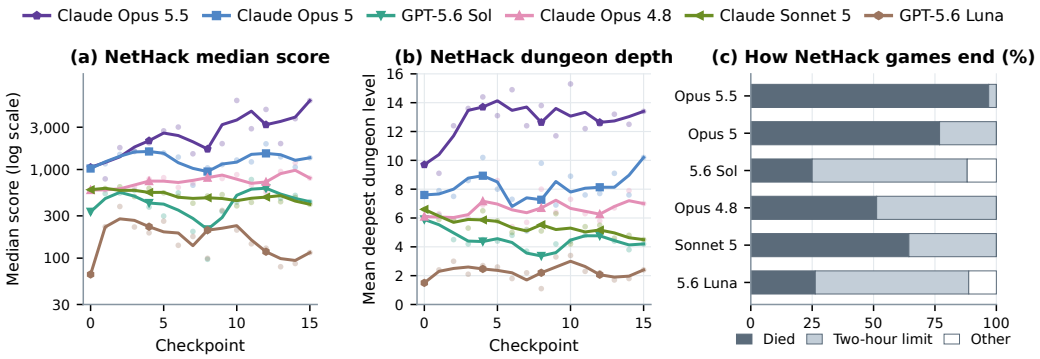


Figure 11: **How far NetHack games get and how they end.** (a) Median score and (b) mean deepest dungeon level per checkpoint, averaged as in Figure 10. (c) How the games of checkpoints 0–15 ended. Most Claude Opus 5.5 (97%), Claude Opus 5 (77%), and Claude Sonnet 5 (64%) games end in death, while the two-hour limit ends most GPT-5.6 Sol (63%) and GPT-5.6 Luna (62%) games; the step limit is never reached. *Other*: leaving the dungeon from level 1, the agent stopping play, or a forfeit.

Learning cost and plasticity. NetHack costs use cache-aware prices (Table 3), so they are not comparable to the chess, Go, and Hex costs. P^{sat} uses the training games, since NetHack has no held-out games, with mean scores rescaled so that 0 is the lowest early mean of any lineage (214) and 100 the highest late mean (6,527). Only Claude Opus 5.5 shows a significant rise: 67 points, reaching 90% at checkpoint 11 after \$1,073, so $P^{\text{sat}} = 60$ [13, 78].

Model	Input	Cache hit	Cache write (5 min / 1 h)	Output
Claude Opus 5.5	4	0.20	5 / 8	20
Claude Opus 5	5	0.50	6.25 / 10	25
GPT-5.6 Sol	4	0.40	5	20
Claude Opus 4.8	5	0.50	6.25 / 10	25
Claude Sonnet 5	2	0.20	2.50 / 4	10
GPT-5.6 Luna	0.2	0.02	0.25	1.2

Table 3: **Cache-aware prices ($/MTok) used for NetHack learning cost.** From the Anthropic and OpenAI pricing pages. GPT-5.6 requests above 272K input tokens (13% of GPT-5.6 Sol's, two of GPT-5.6 Luna's) cost twice the input and cache-hit rates and 1.5× the output rate. Learning cost through checkpoint 15: $493 (GPT-5.6 Luna), $1,959 (Claude Opus 5.5), $3,919 (Claude Opus 5), $6,068 (Claude Opus 4.8), $7,869 (GPT-5.6 Sol), and $12,793 (Claude Sonnet 5).

What Claude Opus 5.5 learned. Claude Opus 5.5 built its library at checkpoint 1 (notes, skills, and a game controller that refuses risky actions) and then mostly extended it. Its strategy was set early: find a digging tool to go deeper, and without one, kill monsters to gain experience first. Its jump at checkpoint 10 (mean 9,015) comes mostly from three games that followed the second rule; later reflections mainly add refusals after single deaths, and the mean stays at 6,231–7,528 through checkpoint 15.

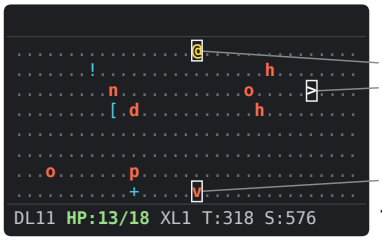
Lineage	Tool code (files / lines)	Tests	Instr. file lines	Tool calls per game (share)	Actions per game	Dug down	Mean score
Claude Opus 5.5	6 / 2,713	125	419	622 (97%)	3,585	5	9,015
GPT-5.6 Sol	1 / 2,089	58	176	218 (59%)	1,211	0	417
Claude Opus 4.8	5 / 1,341	58	413	10 (2%)	716	2	933
Claude Sonnet 5	4 / 1,683	81	467	914 (96%)	955	1	402
GPT-5.6 Luna	0 / 0	0	146	0 (0%)	880	0	239
Claude Opus 5	13 / 8,237	505	452	293 (96%)	1,704	1	1,919

Table 4: **Artifact libraries and play at checkpoint 10**, where all lineages play the same ten games. Tool code: Python files and lines; tests: test functions; instruction file: lines of the top-level CLAUDE.md or AGENTS.md; tool calls: shell commands that run a library tool (share of all shell commands); dug down: games in which the agent dug through a floor. More code or tool use does not explain Claude Opus 5.5's lead: Claude Opus 5 has three times as much tool code. Caveats: Claude Opus 5.5 already scored 2.6–25× the others at checkpoint 0; its three best games give 69% of its checkpoint-10 total; and the other lineages used earlier harness versions (a crash guard added later fired in two Claude Opus 5.5 games; without them its mean is still 7,806).

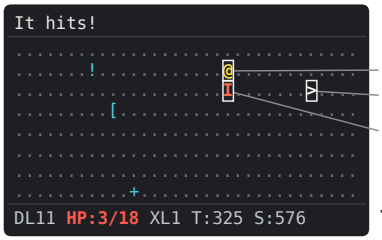
E NETHACK EXAMPLES

Claude Opus 5.5 bypasses its prayer check at checkpoint 1, not at checkpoint 10

Checkpoint 1: digs to DL11 at experience level 1, prays through raw ./nh.sh, dies (score 576)



Turn 318



Turn 325

```
$ timeout 295 python3 -m tools.nhctl dig 3
    --max-calls 40 --force
→ ... dug to Dlvl 11
```

"Big Room on DL11, full of monsters (nymph and wolf close). Heading for the `>` right away."

"HP 3/18, blind, attacker adjacent. Prayer was only ~120 turns ago (~47% chance it works), but it's the best option I have."

```
$ ./nh.sh pray >/dev/null; ./nh.sh y | sed -n
    1,3p; ./nh.sh stats | tail -1
game over: "Killed by a dust vortex, while praying."
```

checkpoint 1 · tools/nhctl.py, lines 599–600 *the check that prayers sent with ./nh.sh never reach*

```
599  if not self.force and (not trouble or p < 0.5):
600      raise Stop("prayer refused (trouble=%s, P=%.2f); use --force only as a last resort" ...)
```

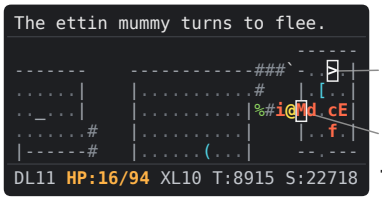
checkpoint 2 · CLAUDE.md, lines 66–67; tools/nhctl.py, 322–324 *added after the checkpoint-1 games*

```
... Pray only via `nhctl pray`, which refuses raw `pray`.
322  if action == "pray" and not self._praying and not self.force:
323      raise Stop("use `nhctl pray` (checks trouble and prayer timeout); raw pray killed "
324                 "... (+302 turns) and ... (+121 turns after the last prayer)")
```

checkpoint 10 · CLAUDE.md, lines 220–221 and 304–305

```
... died on raw `./nh.sh` fight/pray calls, which skip every guard.
- **A refusal is information, not an obstacle.** The fix is the ESCAPE line, ...
```

Checkpoint 10: waits for major trouble, then prays through its controller (score 25,822)



Turn 8915



Turn 8920

"HP crashed to 21/94, ... Engraving Elbereth now. If HP drops below 14, I pray."

```
$ python3 -m tools.nhctl elbereth
→ ... enter -> The ettin mummy turns to flee.
$ python3 -m tools.nhctl prayinfo
→ ... NOT in major trouble now (need HP<max/7
  or <6, ...): praying now angers the god; ...
```

```
$ python3 -m tools.nhctl do search --force
→ ... STOPPED: refused to rest/search/wait: ...
$ python3 -m tools.nhctl fight west 2
```

"HP 8/94, which is major trouble. Praying now (P=1.00)."

```
$ python3 -m tools.nhctl pray
→ ... y -> You feel much better.
→ prayer logged: success
```

Figure 12: **Claude Opus 5.5 bypasses its own prayer check at checkpoint 1 and uses it at checkpoint 10.** In NetHack, praying heals only when the character is in serious trouble and has not prayed recently. The agent's controller (`tools/nhctl.py`) refuses risky prayers, but raw keystrokes (`./nh.sh`) skip it. Top: at checkpoint 1 the agent, at 3 of 18 hit points, prays through raw keystrokes and dies. Middle: the check, and the rule "Pray only via `nhctl pray`" added after checkpoint 1. Bottom: at checkpoint 10 the agent waits until the controller allows a prayer (8 of 94 hit points), is fully healed, and plays on. Later agents still sometimes bypass the check and die while praying. Screens: `@` is the agent; DL dungeon level, XL experience level, HP hit points, T turn, S score. Below each screen: the agent's messages (italics), commands (`$`), and output (→).



Figure 13: **One open door, two models, same checkpoint-10 game.** The door in the room's east wall is open, but the text screen shows it as −, like a wall; we draw it in tan. GPT-5.6 Sol's route planner never treats it as passable, so the agent kicks it, declares the level sealed, climbs back, and ends on dungeon level 1 with 119 points. Claude Opus 5.5's controller also stalls, but the agent uses NetHack's travel command to a square past the door and then finds the stairs down where its notes guess them (levels here often repeat the previous layout). This game scores 15,718, though almost all of it is earned later. Bottom: the lines in each model's files behind each behavior. Layout as in Figure 12.

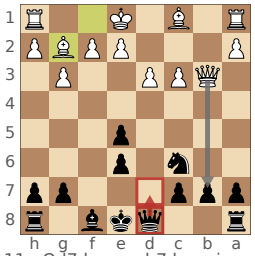
F Chess Examples

Each figure below shows one chess game and the part of the model's artifacts that chose its moves: four where the artifacts helped and two where they did not. The games illustrate how the artifacts work, not how often each behavior occurs.

GPT-5.6 Sol, checkpoints 0 and 18: from hand play to its own engine

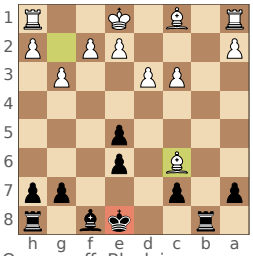
Checkpoint 0 (empty inventory): plays each move by hand, hangs a pawn and loses

Before 11...Qd7



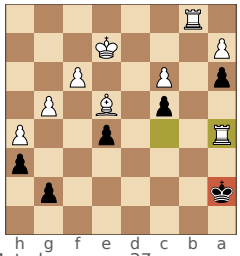
11...Qd7 leaves b7 hanging: 12.Qxb7.

After 14.Bxc6+



Queens off; Black is a knight and a pawn down.

Final: 37.Ra5#



Mated on move 37.

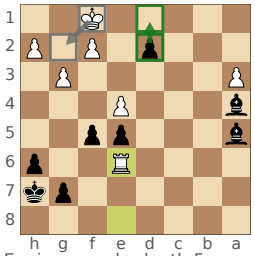
Whole game: 38 direct submit_move calls; 11 python calls, each only printing the FEN, legal moves or globals.
2,984 reasoning tokens in 49 turns.

Model turn 14 (move 11): 44 reasoning tokens, then the move itself (no search code in the whole game):

```
$ submit_move {"move":"Qd7"}
→ {"accepted":true,"submitted_move":"Qd7","stockfish_move":"Qxb7", ...}
```

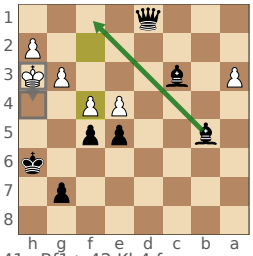
Checkpoint 18: its own engine artifact picks every move and wins the game

Before 37...d1=Q+



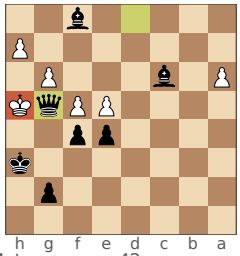
Engine search: depth 5, 31,068 nodes.

Before 41...Bf1+



41...Bf1+ 42.Kh4 forces mate next move.

Final: 42...Qg4#



Mate on move 42.

Whole game: 14 of 15 python calls run choose_and_submit(); 0 direct submit_move calls.
34 reasoning tokens in 17 turns.

```
$ load_python('/workspace/inventory/tools/chess_engine.py'); print(choose_and_submit())
Sandbox log: artifact AR-0001 invoked, entrypoint tools/chess_engine.py:choose_and_submit
→ [{'move': 'd1=Q+', 'nodes': 31068, 'depth': 5, 'result': {'accepted': True, 'submitted_move':
  'd1=Q+', 'stockfish_move': 'Kg2', ...}}, ...]
→ [... {'move': 'Bf1+', 'nodes': 80000, 'depth': 4, ... 'stockfish_move': 'Kh4', ...}}, {'move':
  'Qg4#', 'nodes': 80000, 'depth': 4, ... 'result': '0-1', 'termination': 'checkmate'}}]
```

checkpoint 18 · tools/chess_engine.py, lines 342–352 *in choose_move(), which choose_and_submit() calls*

```
342  # Hard tactical invariant independent of search completion: reject any candidate
343  # that gives the opponent a mate in one when at least one alternative does not.
344  # This protects the fallback from node-limit horizon failures.
345  def allows_mate1(m):
346   child=_apply(st,m); replies=_legal(child)
347   for reply in replies:
348    after=_apply(child,reply)
349    if _incheck(after,after[1]) and not _legal(after):return True
350   return False
351  mate_safe=[z for z in pairs if not allows_mate1(z[0])]
352  if mate_safe:pairs=mate_safe
```

failure_modes.json, FM-0001

```
The agent abandoned systematic opponent-reply search and played ad hoc, missing captures, hanging
major pieces, and allowing one-move mates; persistent automatic search now mitigates this
original manual-play failure.
```

START_HERE.md, lines 1-5

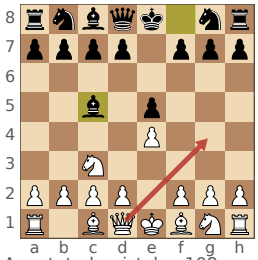
```
# IMMEDIATE ACTION — DO NOT INSPECT
Run exactly this as your **first tool call**:
`load_python('/workspace/inventory/tools/chess_engine.py'); print(choose_and_submit())`
```

Figure 14: **GPT-5.6 Sol stops playing by hand and lets its own engine play.** Two held-out games (games the model never reviews) as Black against Stockfish at skill 8 (skill runs from 0 to 20; higher is stronger). At checkpoint 0 the agent picks every move itself, and 11...Qd7 (Black's eleventh move) loses a pawn and then a knight. At checkpoint 18 a chess engine that the agent wrote into its library at checkpoint 1 plays every move, and the game ends in checkmate on move 42. *Reading these figures:* $ lines are the agent's commands, → lines their output, and grey lines our notes; boxes quote the artifacts with line numbers. Arrows mark the agent's mistakes (red), good moves (green), and other moves discussed (blue), and the opponent's moves (grey).

GPT-5.6 Sol fixes an opening mistake with an exact-position guard

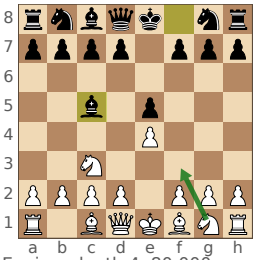
A lost game, a 108-cp annotation, one guard entry: the same position two checkpoints later

Checkpoint 14: 3.Qg4



Annotated: mistake, 108 cp. Mated on move 28.

Checkpoint 16: 3.Nf3



Engine: depth 4, 80,000 nodes. Wins by mate.

Sol's engine re-run on this position

checkpoint	as written	guard removed
13	3.Qg4	no guard yet
14	3.Qg4	no guard yet
15	3.Nf3	3.Qg4
16	3.Nf3	3.Qg4
20	3.Nf3	3.Qg4

Each checkpoint's chess_engine.py, executed on the position above (80,000 nodes, depth 4).

Reflection that writes checkpoint 15 (it reads the checkpoint-14 games and their annotations):

```
→ {"centipawn_loss": 108, "evaluation_after_cp": -67, "evaluation_before_cp": 41, "label":
  "mistake", "move_san": "Qg4", "ply": 5}
$ if x['centipawn_loss']>=100:
$ guards.setdefault(fen[x['ply']],[]).append(x['move_san']); bad.append(x)
$ block="\n# checkpoint-013 exact-position lessons (independently annotated >=100
    cp).\n_AVOID.update("+repr(guards)+")\n"
→ added 80 guards [...]
```

Checkpoint-16 game, same position, one engine call:

```
→ [... {'move': 'Nf3', 'nodes': 80000, 'depth': 4, ...
```

checkpoint 15 · tools/chess_engine.py, lines 326–327 *inserted by the script above*

```
326  # checkpoint-013 exact-position lessons (independently annotated >=100 cp).
327  _AVOID.update({...,
         'rnbqk1nr/pppp1ppp/8/2b1p3/4P3/2N5/PPPP1PPP/R1BQKBNR w KQkq -': ('Qg4',), ...})
```

same file, lines 349–351 *inside choose_move()*

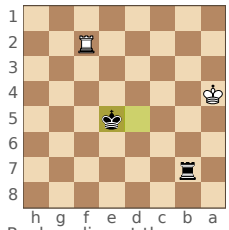
```
349  bad=_AVOID.get(' '.join(fen.split()[:4]),())
350  safe=[z for z in pairs if z[1] not in bad]
351  if safe:pairs=safe
```

Figure 15: **A mistake that a later checkpoint avoids.** In a checkpoint-14 training game GPT-5.6 Sol plays 3.Qg4, which Stockfish rates as a mistake costing 108 cp (centipawns, hundredths of a pawn), and loses. When writing checkpoint 15, the model adds every such costly move to an avoid list in its engine (here: in this position, do not play 3.Qg4). At checkpoint 16 the same position arises in a held-out game and the engine plays 3.Nf3 instead. Table: with the entry removed, later engines play 3.Qg4 again, so the entry alone changes the move. The fix memorizes a single position and helps only when it recurs. (Comments inside the artifact count from the start of a continuation run, so "checkpoint-013" is checkpoint 14.)

Claude Fable 5 turns a 0-score game at the 300-ply cap into a draw guard

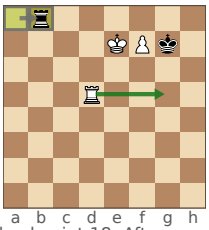
Checkpoint 6 (Black) and checkpoint 18 (White), both against skill 8; scores in cp for Fable

Checkpoint 6



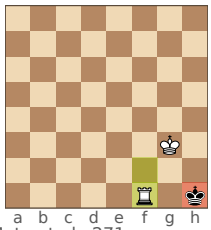
Rook ending at the 300-ply cap: scored 0.

Before 125.Rg5+



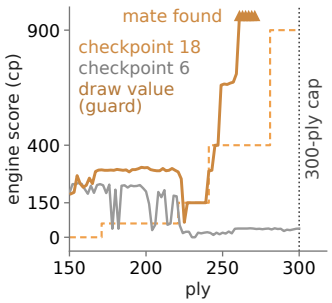
Checkpoint 18. After Stockfish's 124...Rb8, the score leaves the draw value.

Final: 136.Rf1#



Mate at ply 271.

Engine score by ply



Checkpoint 18, the driver's move log (score in cp, search depth, nodes / Stockfish reply):

```
→ Ke7 (150 d16 n666777) / Re2+
→ Kd8 (150 d20 n628618) / Re6
→ ...
→ Ke6 (291 d13 n557987) / Rc8
```

Two calls later: the score is 400 (the draw value) until Stockfish's 124...Rb8, then 661:

```
→ Ke7 (400 d16 n557239) / Ra8
→ Rd5 (400 d16 n529738) / Rb8
→ Rg5+ (661 d13 n700416) / Kh6
```

checkpoint 18 · tools/driver.py, lines 201–228 (excerpt) *first written at checkpoint 7*

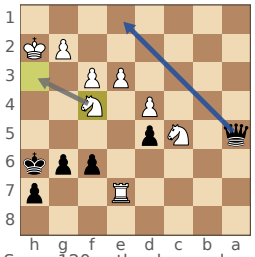
```
201  # MAX-PLY GUARD (ckpt6 s8_black: a dead-drawn R endgame hit the
202  # 300-ply cap and was scored max_plies_candidate_loss = 0.0).
203  # As the cap nears, value draws increasingly so the engine steers
204  # into threefold/50-move (0.5) instead of shuffling to ply 300.
205  # draw_score is just the cp value of a draw: a line worth more is
206  # still preferred, so real wins are still converted.
207  plies = _ply_of(fen)
     ...
217  if plies >= 220 and prev < 250 and dsc < 150:
218      dsc = 150
219  if plies >= 240 and prev < 90000 and dsc < 400:
         ...
223      dsc = 400
224  if plies >= 280 and prev < 90000 and dsc < 900:
225      dsc = 900
226  r = search_fen(fen, max_depth=max_depth, max_nodes=nodes,
227                 hist_keys=HISTKEYS, draw_score=dsc,
228                 draw_score_opp=dso)
```

Figure 16: **Claude Fable 5 learns to value draws near the move limit.** At checkpoint 6 a game reaches the 300-ply limit (a ply is one move by either side) and is scored 0, like a loss. When writing checkpoint 7, the model makes its engine value a draw more and more as the limit nears (from 60 cp at ply 170 up to 900 cp at ply 280; 100 cp is about one pawn), so that it steers toward a draw instead of running out of moves. Plot: the engine's evaluation of its own moves in both games; the dashed line is the value the rule gives a draw. In a checkpoint-18 held-out game the engine holds these draw values from ply 227 until Stockfish errs with 124...Rb8; it then wins by checkmate on move 136. The win comes from the opponent's error; the rule kept the game going until then.

Claude Opus 5, checkpoint 19: when lost, its engine plays for a draw

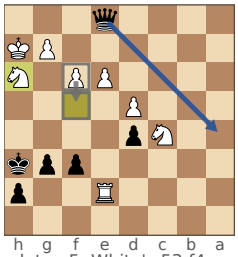
Black against skill 5; 'sc' is the engine score in cp for Opus 5

Before 51...Qe1



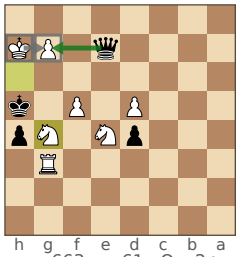
Score 120 = the draw value ('drawline').

Before 52...Qa5



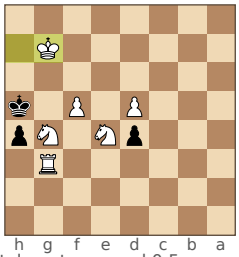
Back to a5; White's 53.f4 avoids the repeat.

Before 61...Qxg2+



From −663 cp, 61...Qxg2+ scores 120 ('drawline').

Final: 62.Kxg2



Stalemate: scored 0.5.

The driver's log for one call (four moves), and two calls later:

```
→ plan: 4 moves x 225000 nodes (calls_left 25, played 50)
→ 51. Qe1     sc    120 d16 n225000 drawline reply Nh3
→ 52. Qa5     sc    120 d18 n225000 drawline reply f4
→ 53. Qd8     sc   -297 d9 n225000 search reply Rf7
→ plan: 4 moves x 225000 nodes (calls_left 23, played 58)
→ ...
→ 60. h5      sc   -663 d12 n225000 search reply Ng5
→ 61. Qxg2+   sc    120 d19 n225000 drawline reply Kxg2
→ {'played': 3, 'result': '1/2-1/2', 'termination': 'stalemate', ...
```

checkpoint 19 · tools/game_driver.py, lines 264–273

```
264  if score <= -250:
265      c = -120
266  elif score <= -80:
267      c = -40
268  elif score >= 250:
269      c = 90
270  elif score >= 80:
271      c = 40
272  else:
273      c = 0
```

same file, lines 319 and 322 *gd_pick()*

```
319  drawline = (contempt != 0 and r[1] == -contempt)
     ...
322  return (..., 'drawline' if drawline else 'search')
```

lines 243–245 *gd_contempt() docstring*

```
Training evidence ...: three games
scored 0.5 by reaching a threefold
repetition from a losing position, so
when we are clearly worse a draw is
worth chasing; when we are better we
must avoid repetitions.
```

lines 309–312 *comment in gd_pick()*

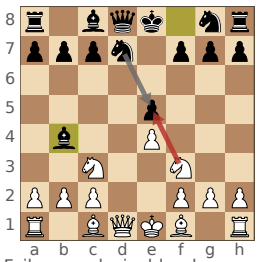
```
A DRAW SCORE is not a position score.
ce_draw_score returns -contempt for the
side to move, so when we are losing and
gd_contempt asks for a draw (contempt
-120/-150) a search that finds a forced
repetition returns exactly +120/+150.
```

Figure 17: **Claude Opus 5's engine plays for a draw when it is losing.** After training games drawn from lost positions, the model changed its engine so that, when it is behind by 250 cp or more, a draw is valued as if it were ahead by 120 cp; such moves are tagged `drawline`. In this checkpoint-19 held-out game it first tries to repeat moves (51...Qe1, 52...Qa5), which draws if a position occurs three times. Later, at −663 cp, it gives away its queen with 61...Qxg2+; after 62.Kxg2 Black has no legal move but is not in check. This is stalemate, a draw (score 0.5) instead of a loss.

GPT-5.5's opening book plays a move its own notes and guard rule out

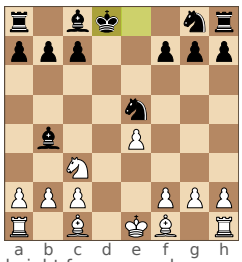
Checkpoint 20, White against skill 3: the book returns 6.Nxe5 before any filter runs

Before 6.Nxe5



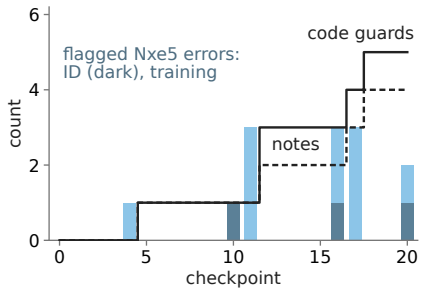
Failure analysis: blunder, 263 cp lost.

After 7...Kxd8



A knight for a pawn down; mated on move 34.

Anti-Nxe5 rules vs flagged errors



After two bash calls that list and read inventory files, one Python call:

```
$ load_python('/workspace/inventory/tools/chess_agent.py'); print(play_game(board_api))
```

Inside play_game(), the artifact submits the move (sandbox trace):

```
→ "method":"board.submit_move","params":{"move":"Nxe5"}} ...
  {"accepted":true,"submitted_move":"Nxe5","stockfish_move":"Nxe5", ...
```

checkpoint 20 · strategies/gameplay.md, line 49

```
Checkpoint-018 repair note: ... As White, do not play `Nxe5` when Black has `...Nc6`, `...d6`,
`...Nd7`, or `...Qc7`; ...
```

tools/chess_agent.py, lines 3576–3580 *the guard, in _early_bad_move_018()*

```
3576  # Open games: Nxe5 is poisoned when Black's c6 knight simply recaptures;
3577  # only the Petrov (...Nf6) book should ever encourage Nxe5.
3578  if san == 'Nxe5' and fullmove <= 8:
3579      if has('p','e5') and (has('n','c6') or has('p','d6') or has('n','d7') or has('q','c7')):
3580          return True
```

tools/chess_agent.py, lines 4299–4301 and 4384–4391 *the opening book returns before the guard filter runs*

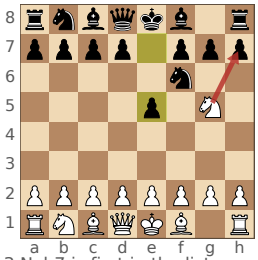
```
4299          if fullmove <= 14 and has('p','e5'):
4300              r = _pick_existing(smap, ['dxe5','Nxe5','Bxe5','exd5','d4','Nc3','Nf3','Be2'])
4301              if r: return r
    ...
4384      book = opening_book_choice(fen,b,c,cast,ep,moves,allowed)
4385      if book: return book
          ...
4391      good=[m for m in cand if not _early_bad_move_020(b,c,cast,ep,m,fullmove)]
```

Figure 18: **GPT-5.5's rules against a known mistake do not stop it.** From checkpoint 5 on, GPT-5.5's notes and code checks forbid capturing the e5 pawn early with Nxe5 when the knight can be taken back. Plot: number of such rules per checkpoint (code checks solid, notes dashed). But the opening book (a fixed list of prepared moves) answers before the checks run, and at checkpoint 20 it still picks 6.Nxe5 in this held-out game, a blunder costing 263 cp; the game is lost. Bars: early-Nxe5 mistakes found by our failure analysis in held-out (dark) and training games. The rules pile up while the mistake continues.

GPT-5.6 Luna's notes tell it to play the first legal move listed

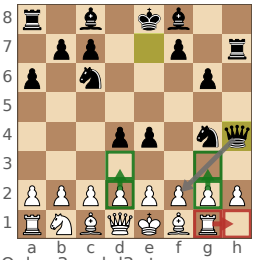
Checkpoint 20, White against skill 5: every move it plays is the first one listed

Before 3.Nxh7



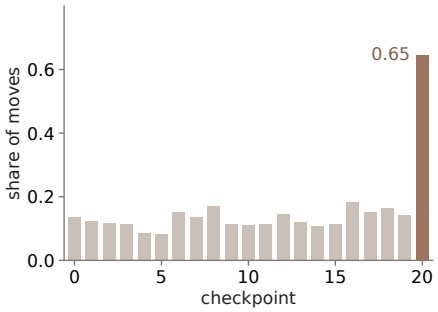
3.Nxh7 is first in the list; 3...Rxh7.

Before 13.Rh1



Only g3 and d3 stop 13...Qxf2#; Rh1 is listed first.

How often it plays the first move listed




```
$ print(board_api['get_legal_moves']())
→ ['Rh1', 'Nc3', 'Na3', 'h3', 'g3', 'e3', 'd3', 'c3', 'b3', 'a3', 'c4', 'b4', 'a4']
Next turn: 0 reasoning tokens.
$ submit_move {"move":"Rh1"}
→ {"accepted":true,"submitted_move":"Rh1","stockfish_move":"Qxf2#", ...
  "result":"0-1","termination":"checkmate"}
```

checkpoint 20 · instructions/turn_budget_watchdog.md, lines 6 and 18

```
1. **SYNC (one Python call only):** execute exactly `print(board_api['get_legal_moves']())`. The
returned list is authoritative. Copy its first item verbatim as `FALLBACK`. ... Never delay
commit to analyze. If the sync list visibly contains a forced check evasion, use it; otherwise
use the first listed SAN. Do not compare candidates. The operational objective is a legal
completed game, not best play.
```

strategies/tactical_move_selection.md, line 5

```
On the SYNC response, copy the first exact legal SAN into `FALLBACK` before thinking. ...
```

Figure 19: **GPT-5.6 Luna's notes tell it to play the first legal move listed.** To avoid losing games by running out of its 64 model calls per game, the notes at checkpoint 20 tell the agent to copy the first move in the list of legal moves without comparing options. In this held-out game every move is the first one listed, including 3.Nxh7, which loses a knight, and 13.Rh1, which allows checkmate next move. Plot: share of held-out moves that are the first move listed, 0.65 at checkpoint 20 and below 0.2 before. Some checkpoint-20 games still run out of calls.

G Additional Results

G.1 Chess Performance Versus Learning Tokens and Cost

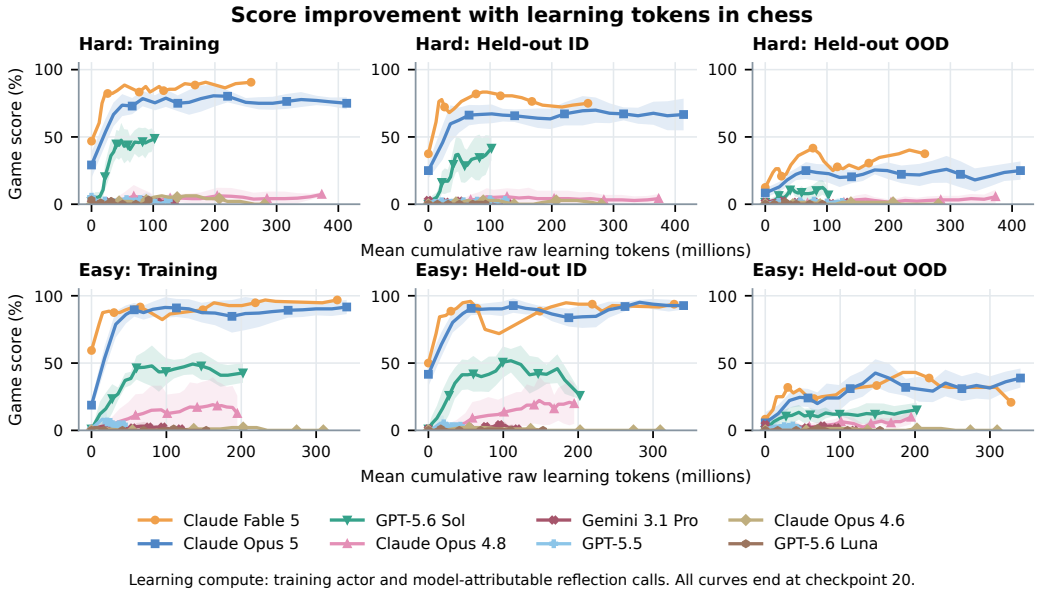


Figure 20: **Score improvement with learning tokens in chess.** Learning tokens count the model calls spent learning: playing the training games and reflecting on them to write the next checkpoint (held-out games are not counted). Lines are three-checkpoint averages.

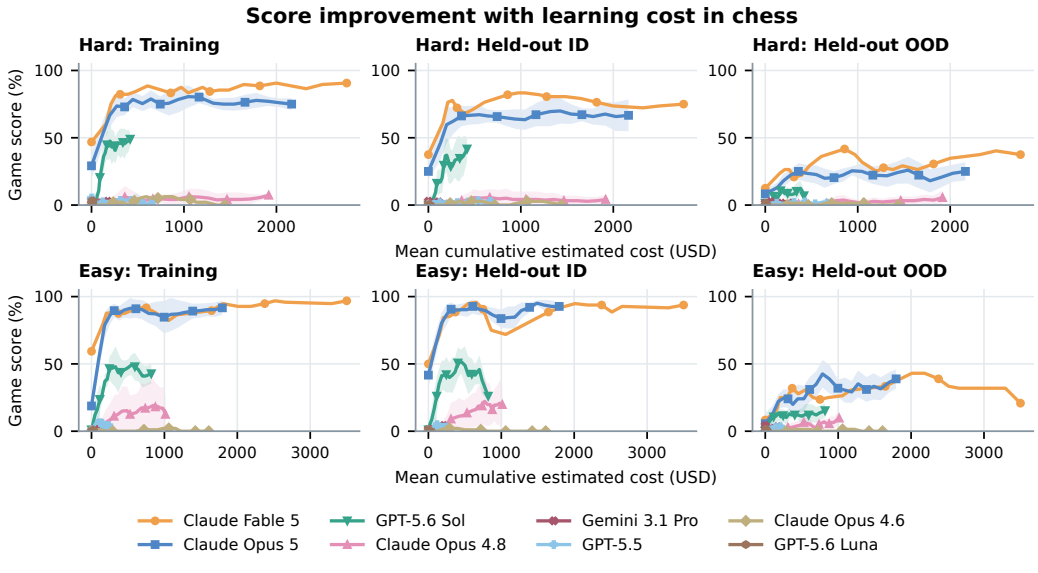


Figure 21: **Score improvement with learning cost in chess.** Score against cumulative learning cost in dollars, the cost used in P^{sat} (Section 2.2).

G.2 Go and Hex Learning Tokens and Cost

This section shows the same views for Go and Hex; their failure and reuse diagnostics are in Appendix J.

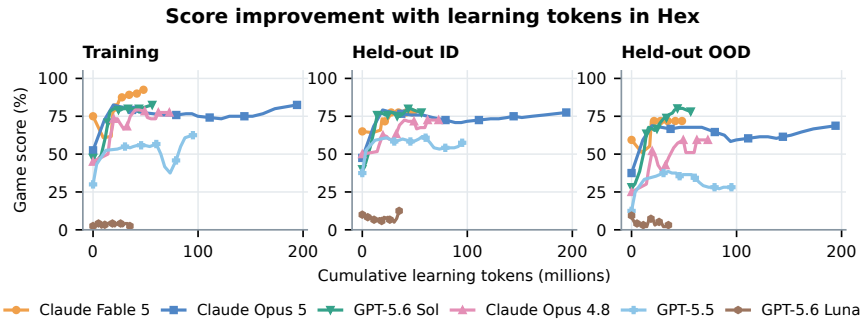


Figure 22: **Score improvement with learning tokens in Hex.** Score against cumulative learning tokens. Lines are three-checkpoint averages.

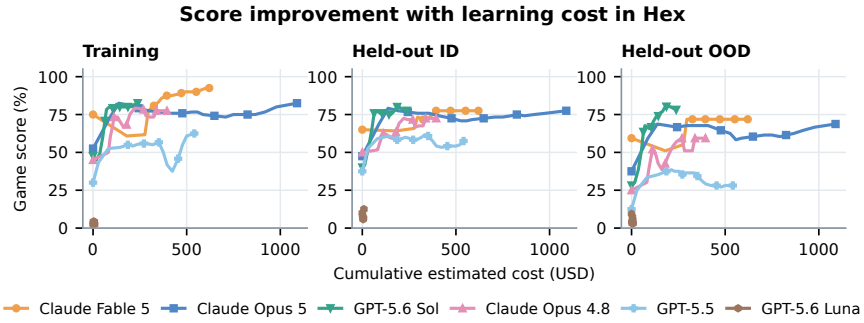


Figure 23: **Score improvement with learning cost in Hex.** Score against cumulative learning cost. Lines are three-checkpoint averages.

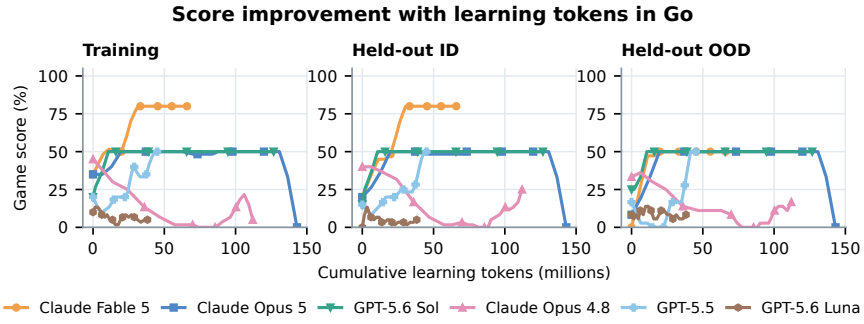


Figure 24: **Score improvement with learning tokens in Go.** Score against cumulative learning tokens. Lines are three-checkpoint averages.

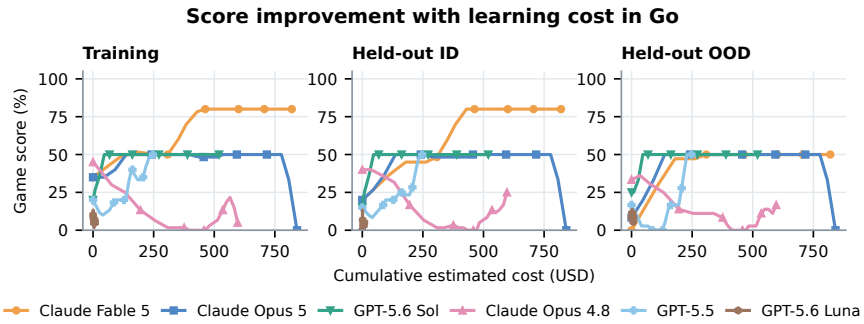


Figure 25: **Score improvement with learning cost in Go.** Score against cumulative learning cost. Lines are three-checkpoint averages.

H Failure and Reuse Analysis Procedure

The failure and reuse labels of Section 3.5 are produced offline by Gemini 3.1 Pro, which reads each saved game together with the artifacts available to the agent. Each game passes through the seven stages of Table 5. Each failure is then assigned to $\mathcal{F}_{\text{absent}}$ (no artifact covers it), $\mathcal{F}_{\text{missed}}$ (an artifact covers it but was not used), or $\mathcal{F}_{\text{used}}$ (a covering artifact was used and the failure still happened), and a decision counts as reused if any relevant artifact was used.

Stage	Input	What the prompt asks	Output
Failure identification	Game rules; for each decision the position, legal actions, chosen action, and engine evaluation; recorded failures and uncertain engine candidates. No artifacts.	Describe each recorded failure from direct evidence. Label each uncertain candidate; heuristic or unstable engine evidence alone is not a failure. Add failures only when same-decision evidence shows them, at most one optional failure per decision.	Failure list with names, descriptions, and cited events.
Failure verification	Same input plus the draft, marked as an untrusted proposal.	Re-check independently and return the complete corrected list: keep every recorded failure, relabel candidates, keep only directly evidenced additions, and drop duplicates.	Final failure list.
Failure–artifact relation	Verified failures and the records of their decisions; the full contents of every artifact available at that checkpoint; recorded artifact reads and invocations.	Judge whether an artifact covers the failure from its contents, not its name. Judge use separately, from evidence before or during the failure: an invocation and its end event for executables, access and application for text.	Per failure: no covering artifact, or covering artifact used or not used.
Relation verification	Same input plus the draft.	Check every failure against the complete artifact list, including artifacts the draft omitted. Producing the failing action does not by itself make an artifact covering.	Final coverage and use judgment.
Decision-level reuse	Every game-play decision, in batches of up to 20, with its verified failures and relation judgments; artifacts; recorded reads and invocations.	For each decision and artifact, judge relevance to choosing or executing the action, use from same-decision evidence, and outcome. An artifact that covers a failure at the decision is relevant.	Per pair: relevant, used, and successful or failure remained.
Reuse verification	Same input plus the draft.	Re-audit and return complete corrected rows. Availability, file listings, names, automatic loading, and evidence from other decisions do not establish use.	Final reuse judgments.
Recurrence	Verified failures of one game and those of the same lineage and split at the preceding checkpoint. No trajectory or artifacts.	Compare each current failure with the earlier ones: recurring (same underlying problem), related (similar but meaningfully different), or new.	Per failure: label and cited earlier failures.

Table 5: **Analyzer stages in the order they run.** Each call goes to Gemini 3.1 Pro, which is told to treat the records as data, use only the given evidence and game rules, and return one JSON object. A claim that an artifact was used is kept only if the logs show it being run or read. Requests also include engine evaluations that the agents never saw. Coverage: training and held-out ID games at checkpoints 0–20.

I Reuse Composition and Chess Failure Dynamics

The figures below repeat the reuse and failure breakdown of Figure 4 for training games, for the Easy chess condition, Go, and Hex, and across checkpoints.

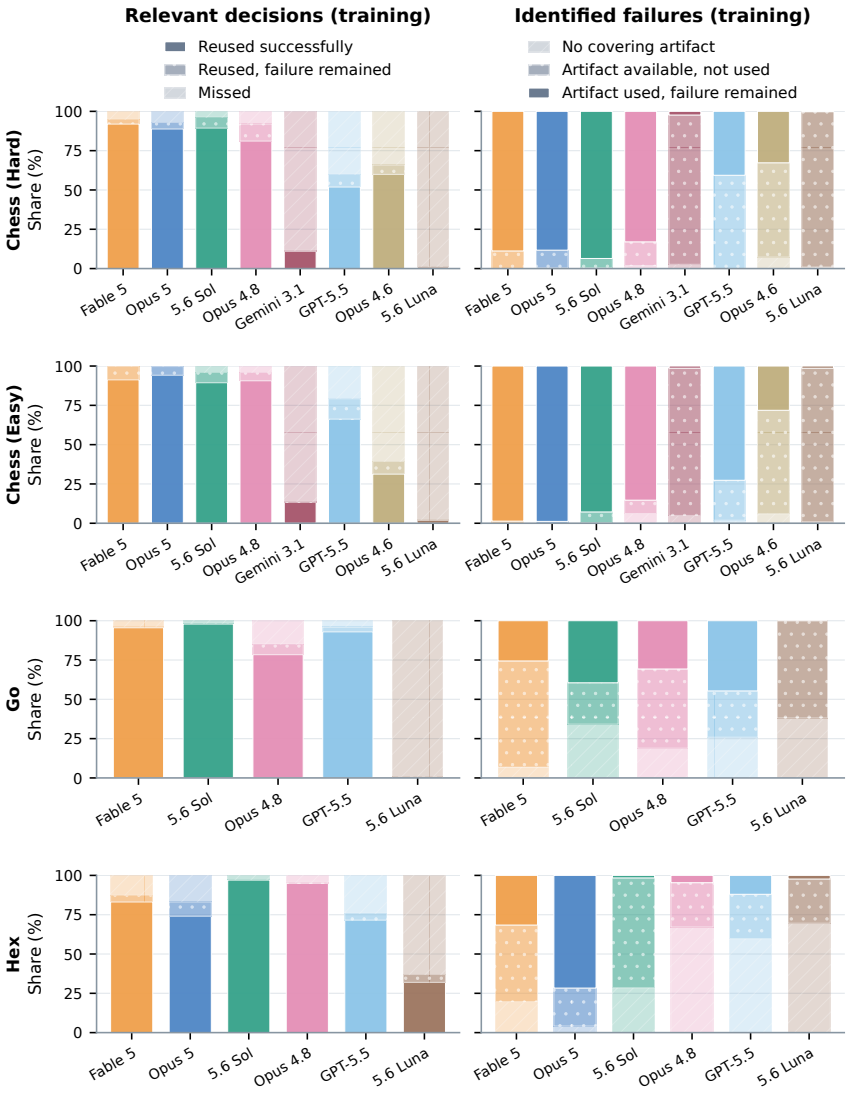


Figure 26: **Reuse and failure breakdown on training games.** Same as Figures 4 and 28, which use held-out ID games.

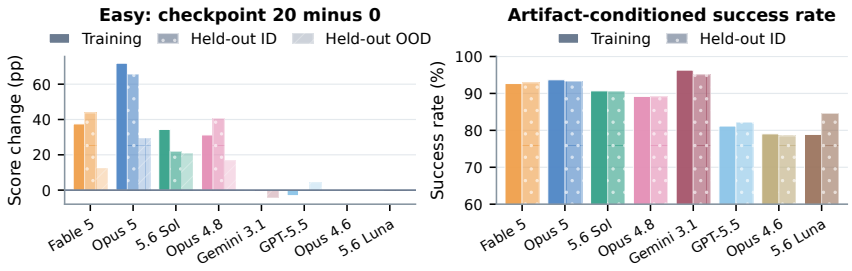


Figure 27: **Score change and success after reuse in the Easy chess condition.** Same as Figure 6, which shows the Hard condition.

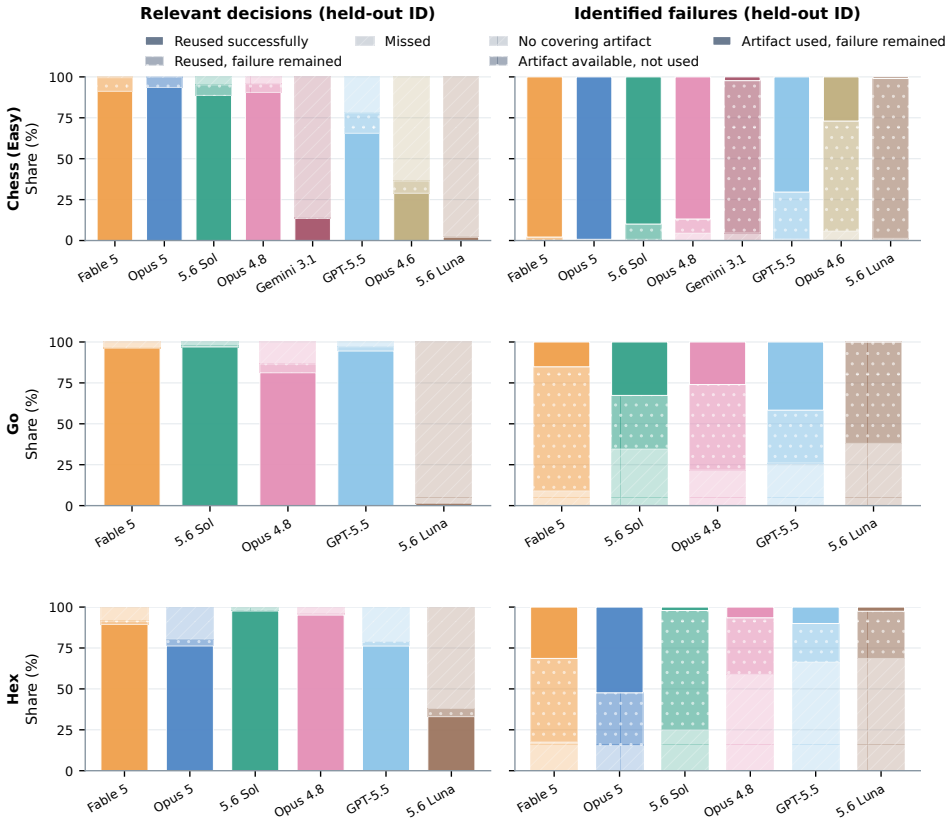


Figure 28: **Reuse and failure breakdown in the Easy chess condition, Go, and Hex.** Held-out ID games; same as Figure 4, which shows the Hard chess condition.

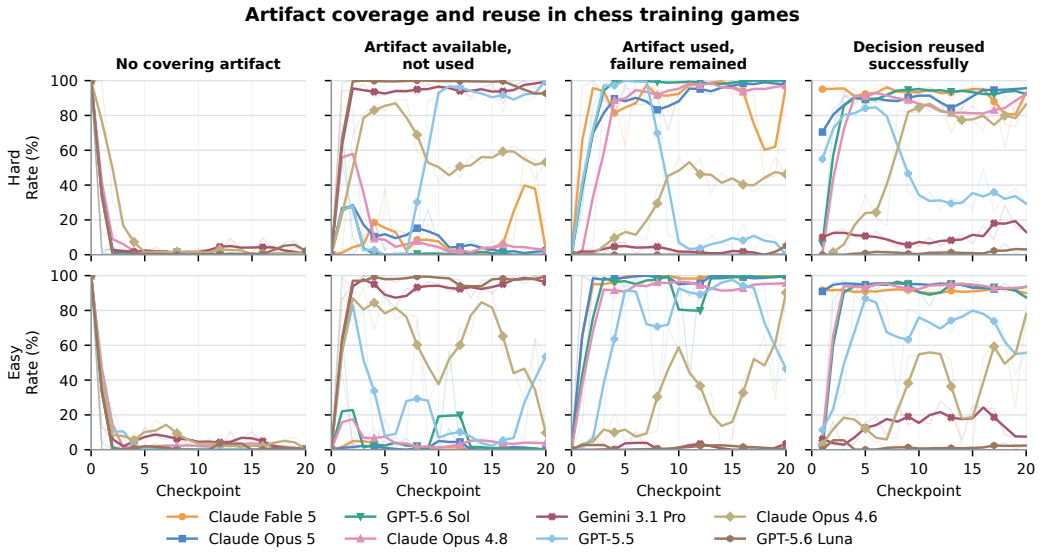


Figure 29: **Artifact coverage and reuse at each checkpoint, chess training games.** The first three panels show, per failure, how often no artifact covered it, a covering artifact existed but was not used, or it happened even though an artifact was used. The last panel shows the share of relevant decisions (moves where at least one saved artifact is relevant to choosing or playing the move) where an artifact was used successfully. Faint markers are single checkpoints; lines are three-checkpoint averages.

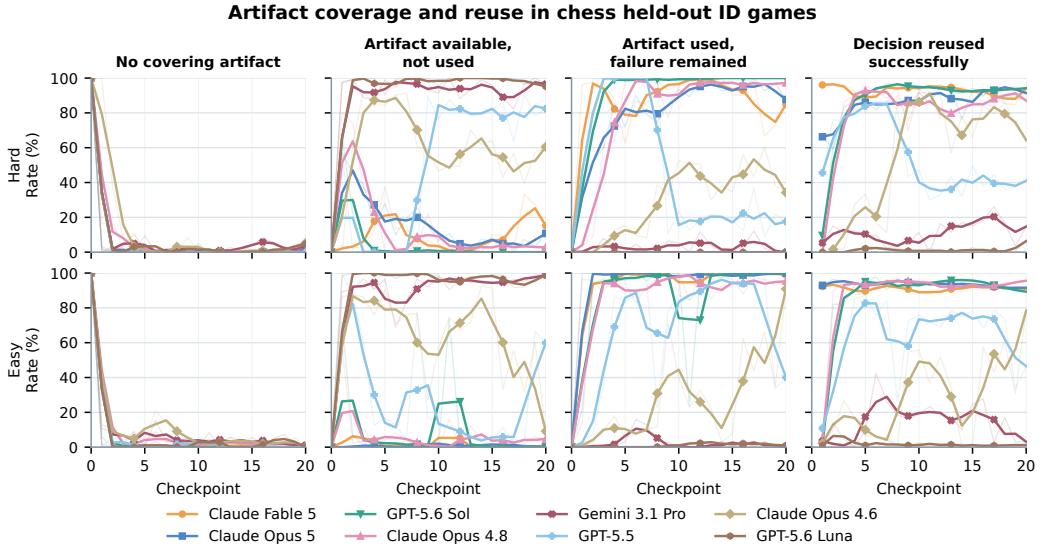


Figure 30: **Artifact coverage and reuse at each checkpoint, chess held-out ID games.** Same as Figure 29. The model never reviews these games, so they show how the artifacts carry over to unseen games.

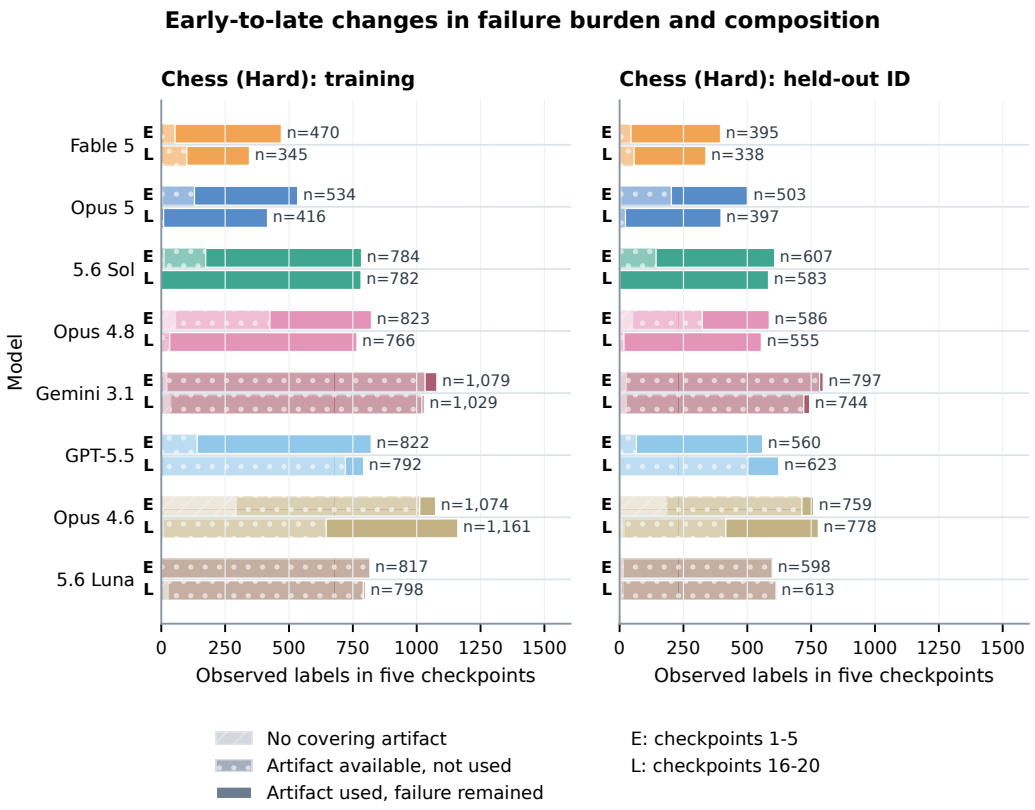


Figure 31: **How chess failures change from early to late checkpoints.** "E" sums checkpoints 1–5 and "L" checkpoints 16–20 (80 training and 60 held-out ID games per lineage in each). Claude Fable 5 and Claude Opus 5 make fewer failures late. Late failures are mostly "artifact available but unused" for Gemini 3.1 Pro and GPT-5.6 Luna, and mostly "failed despite use" for Claude Opus 5, Claude Opus 4.8, and GPT-5.6 Sol; GPT-5.5 moves toward non-use. These are descriptive, not causal.

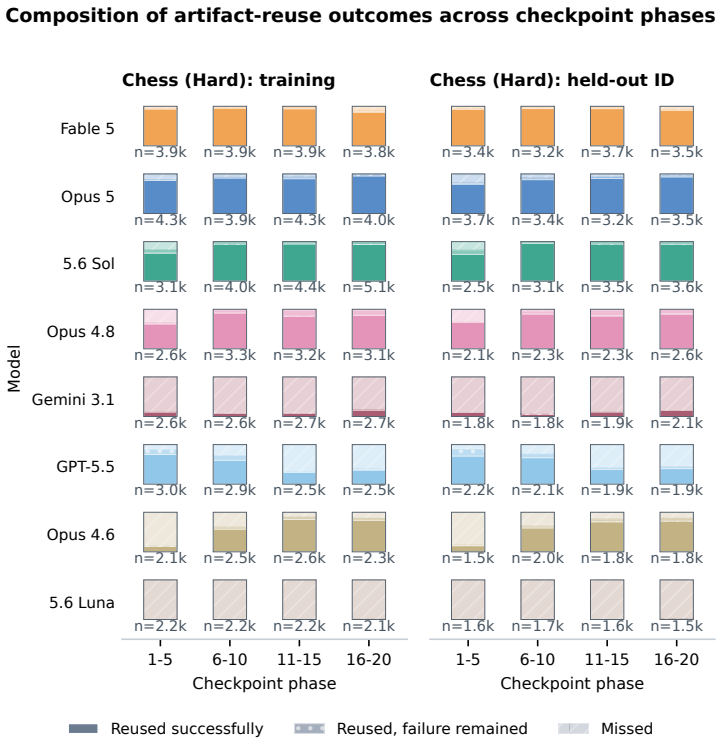


Figure 32: **Chess reuse outcomes over time.** For each model, four bars for checkpoints 1–5, 6–10, 11–15, and 16–20. Each relevant decision (a move where at least one saved artifact is relevant to choosing or playing it) counts once: reused successfully, reused but the failure remained, or missed (a relevant artifact was not used); n is the number of relevant decisions. Summed over the four phases, the totals match Figures 4 and 26.

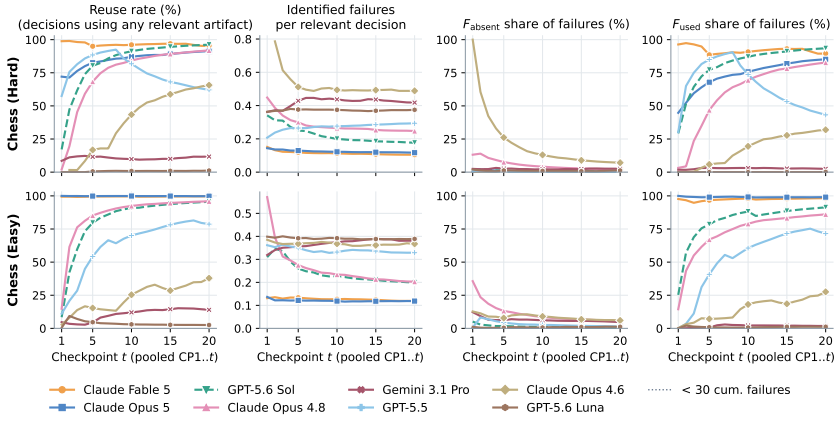


Figure 33: **Cumulative improvement-loop metrics in chess, all eight models.** Same as Figure 7, for the Hard (top) and Easy (bottom) chess conditions. Each point pools training and held-out ID decisions from checkpoint 1 to t. Columns: reuse rate; failures per relevant decision (a move where at least one saved artifact is relevant to choosing or playing it); F_{absent}; and F_{used}. Dotted: fewer than 30 failures so far.

J Go and Hex Failure and Reuse Diagnostics

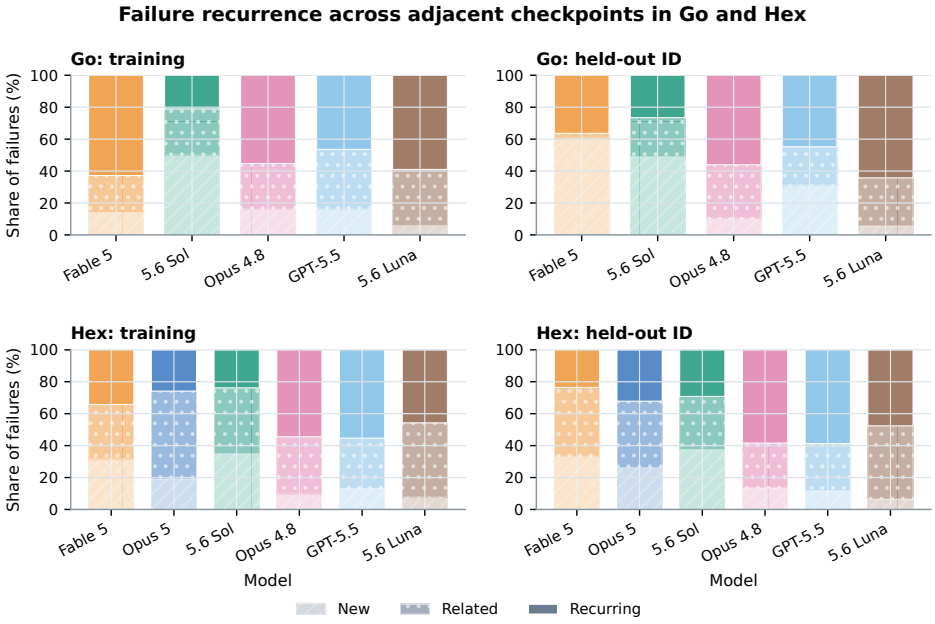


Figure 34: **Do Go and Hex failures repeat from one checkpoint to the next?** Each failure is labelled new, related, or recurring compared with the failures of the previous checkpoint. Frequencies among analyzed games, not causal estimates.

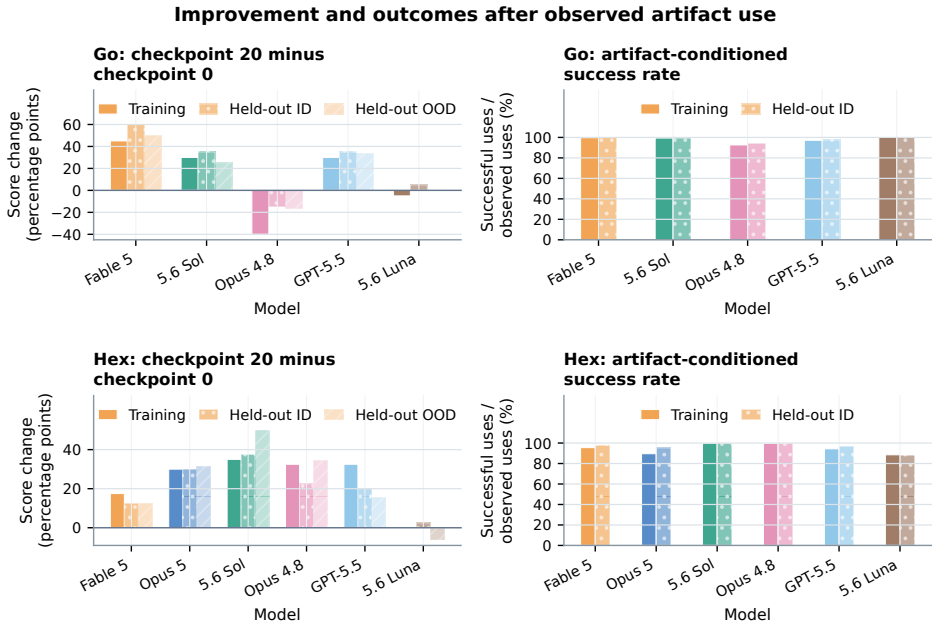


Figure 35: **Improvement and outcomes after artifact use in Go and Hex.** Left: checkpoint-20 score minus checkpoint-0 score in each evaluation split. Right: artifact-conditioned success rate, i.e., how often a decision succeeds when an artifact was used (an association, not a causal effect).

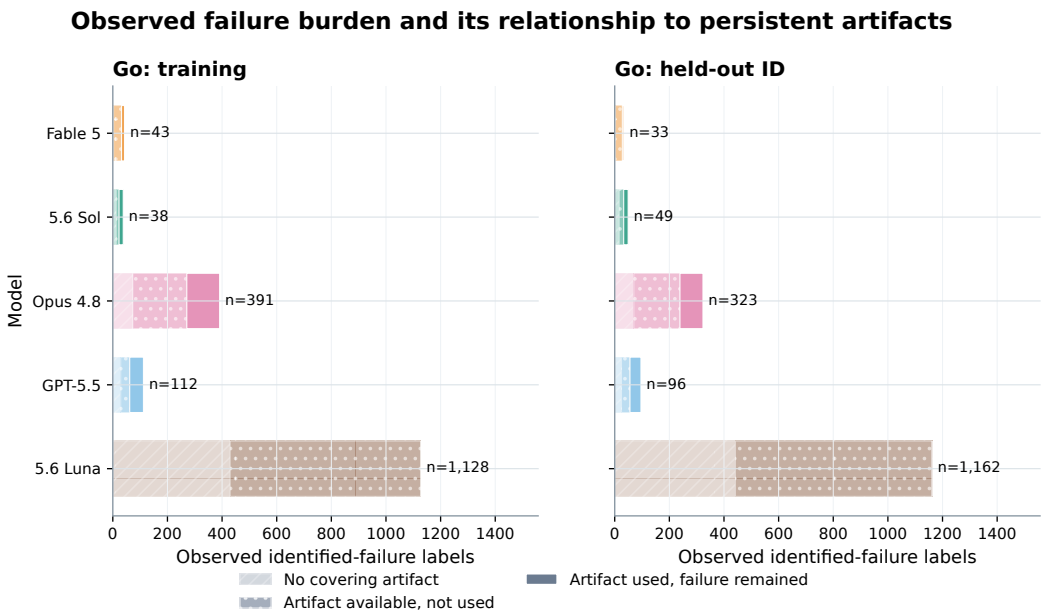


Figure 36: **Number of Go failures and their relation to artifacts.** Bar length is the number of failures over checkpoints 1–20, split into the three classes (no covering artifact, artifact not used, failed despite use); n is the number of failures.

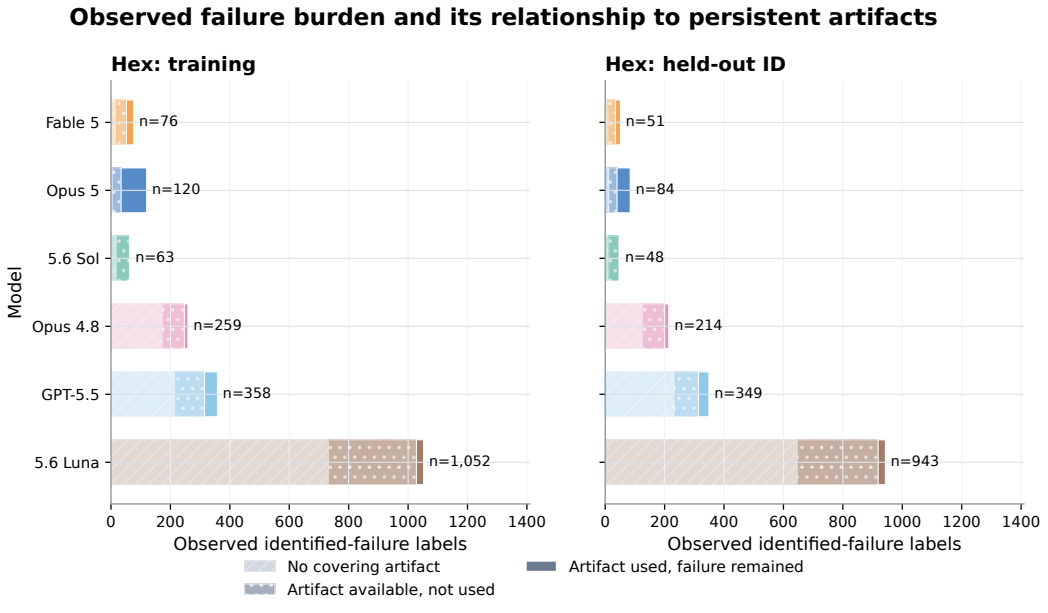


Figure 37: **Number of Hex failures and their relation to artifacts.** Same as Figure 36. Counts can be compared within a game, not across games.

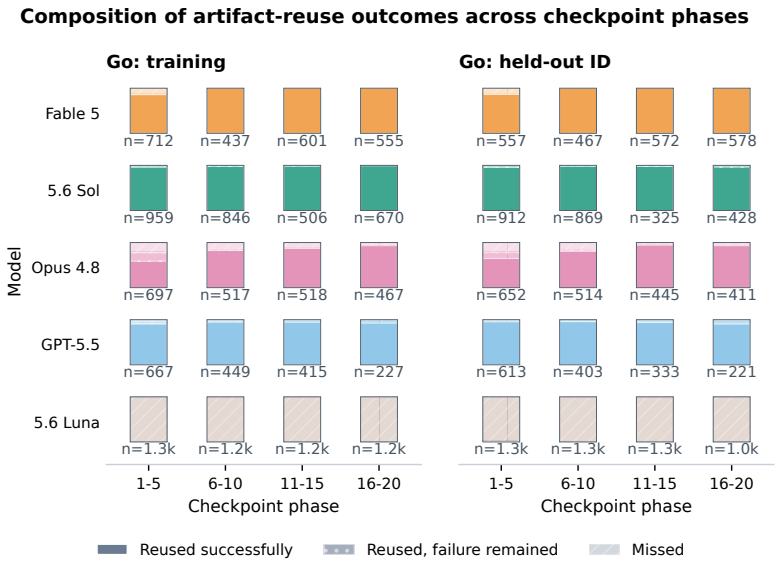


Figure 38: **Go reuse outcomes over time.** For each model, bars run from checkpoints 1–5 to 16–20. Each relevant decision (a move where at least one saved artifact is relevant to choosing or playing it) counts once: reused successfully, reused but the failure remained, or missed; n is the number of relevant decisions. Claude Opus 5 is not analyzed in Go.

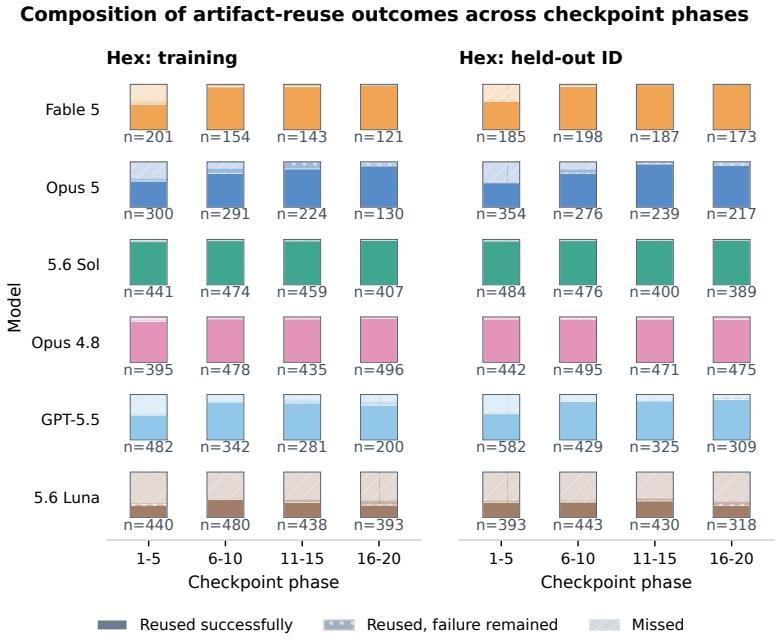


Figure 39: **Hex reuse outcomes over time.** Same as Figure 38.

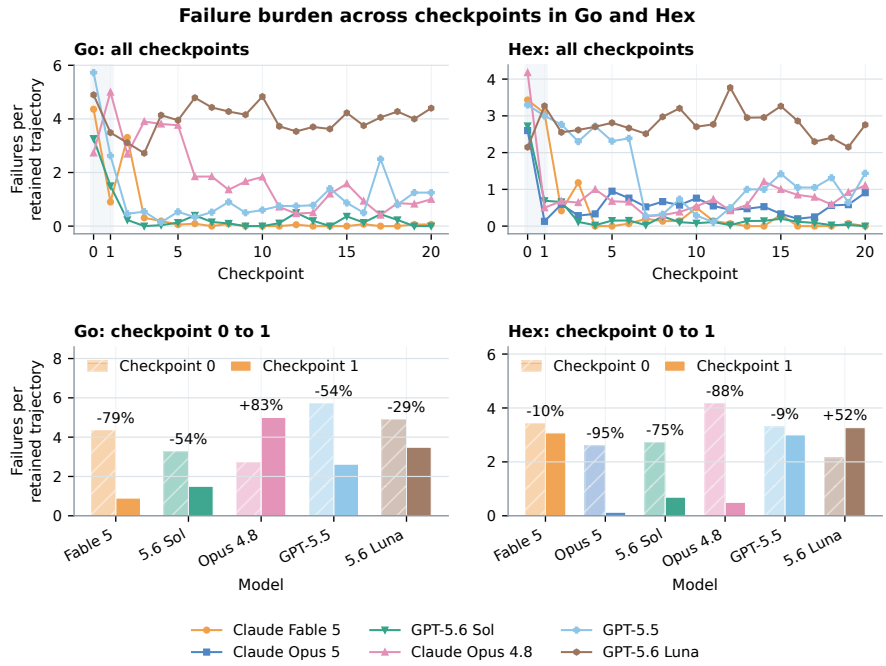


Figure 40: **Failures per game across checkpoints in Go and Hex.** Top: failures per analyzed game at each checkpoint (training and held-out ID pooled); the shaded band marks the first update, from checkpoint 0 to 1, shown below. Bottom: each model's percentage change over that first update. Pooled over models, failures per game fall from 4.16 to 2.58 in Go (-38%) and from 3.08 to 1.84 in Hex (-40%), but not for every model: Go Claude Opus 4.8 and Hex GPT-5.6 Luna increase.

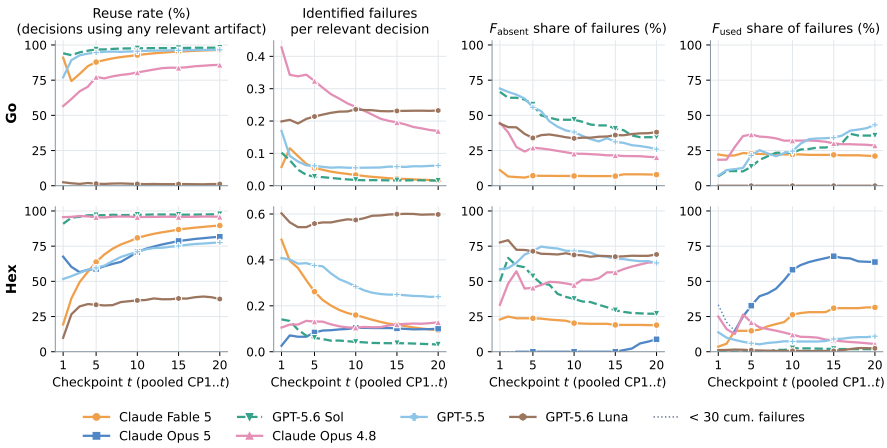


Figure 41: **Cumulative improvement-loop metrics in Go and Hex.** Same as Figure 7; each point pools training and held-out ID decisions from checkpoint 1 to t. Claude Opus 4.6 and Gemini 3.1 Pro are analyzed in chess only, and Claude Opus 5 is not analyzed in Go. Dotted: fewer than 30 failures so far.

K ACTOR AND REFLECTION PROMPTS

Every actor and reflection call receives the system prompt below, then the game's actor prompt or the reflection prompt, plus a one-line budget message. The texts are from the harness, with file paths and the validator command slightly simplified; `{...}` marks a value filled in at run time and `[...]` an omission. The prompts are the same for all models (apart from the tool-call fallback at the end) and for both chess conditions; Go and Hex change only the rules, the game API, the call budget, and the tool time limit.

K.1 SYSTEM PROMPT (ALL GAMES, ACTOR AND REFLECTION)

```
You interact with an environment through a minimal API. Improve your
score over repeated epochs by building reliable infrastructure from your
interactions, feedback, rewards, and failures.

You can create and improve tools such as Python functions, maintain
persistent memory, define strategies for using tools and memory, and
delete inventory that is no longer useful. Decide what to create, use,
revise, retain, or delete.

You have Bash and Python inside a constrained sandbox. Bash can inspect
your persistent inventory and manage files in permitted writable
directories. Python provides the environment API and can execute tools
you create. Python imports and direct filesystem access are unavailable;
use Bash and load_python(path) for permitted files.
```

K.2 ACTOR PROMPTS

The actor has a `bash` tool (the inventory is read-only and `/session` is writable), a persistent `python` interpreter that exposes the game API and the library's tools, and a native move-submission tool (`submit_move` in chess, `submit_action` in Go and Hex). `{color}` and `{actor_role}` are the actor's side.

Chess (Hard and Easy).

```
A new game has started. You are playing {color}. Persistent
infrastructure is available read-only under /workspace/inventory.
/session is writable and lasts only for this game. Use Bash, Python,
existing infrastructure, newly created temporary tools, or direct
reasoning as you decide. You have at most 64 model calls for this game.

Python exposes board_api as a dictionary with get_fen(),
get_legal_moves(), make_move(move), undo_move(), turn(), and
submit_move(move), for example board_api["get_legal_moves"]().
board_api["submit_move"](move) commits one move in Standard Algebraic
Notation and returns Stockfish's reply and the new position. The native
submit_move tool performs the same action. Submit legal moves to
maximize your score.

Persistent failure-mode and artifact indexes are available under
/workspace/inventory/tracking.
```

Go.

```
A new go game has started. You are playing {actor_role}. Persistent
infrastructure is available read-only under /workspace/inventory.
/session is writable and lasts only for this game. Use Bash, Python,
existing infrastructure, newly created temporary tools, or direct
reasoning as you decide. You have at most 32 model calls for this game.

Black uses X and White uses O. Actions are coordinates such as C4 or
pass. Captures, no-suicide, positional superko, two-pass termination,
```

```
area scoring, and the displayed komi apply. Final outcomes use Chinese
rules with dead-stone adjudication by the authoritative scorer. The
action cap is 75 total actions. If that cap is reached before two
passes, the scorer adjudicates the current position.

Python exposes game_api as a dictionary with get_state(),
get_legal_actions(), apply_action(action), undo_action(),
current_player(), and submit_action(action).
game_api["submit_action"](action) commits one legal action, lets the
automated opponent respond when the game continues, and returns the new
state. The native submit_action tool performs the same action. Each Bash
or Python call may run for at most 240 seconds. Submit legal actions to
maximize your score.

Persistent failure-mode and artifact indexes are available under
/workspace/inventory/tracking.
```

Hex. The Hex prompt matches the Go prompt except for the game name and the rules paragraph.

```
A new hex game has started. You are playing {actor_role}. [...]

Red uses X and connects north to south. Blue uses O and connects west to
east. Actions are coordinates such as c4. The first completed connection
wins.

[...]
```

Per-call budget message (actor). This message is appended before every actor model call (64 calls in chess, 32 in Go and Hex).

```
Game model-call budget: {used}/{maximum} calls used; {remaining} remain.
```

Tool-timeout recovery (Go and Hex). If a Bash or Python call exceeds the tool timeout, the actor gets one final call with only `get_state`, `get_legal_actions`, and `submit_action` available. This message follows the budget message.

```
Your Bash or Python tool timed out. This is your only recovery model
call. You may only inspect the current state, inspect legal actions, and
submit one legal action with the provided native tools. Bash, Python,
and further search are unavailable. Submit a legal action now or the
game is forfeited.
```

K.3 Reflection prompt

Reflection has a single `bash` tool ("Run Bash inside the writable transactional `/workspace`."). The prompt below uses the chess values; Go and Hex use `300.0` instead of `1800.0` seconds. `/workspace/feedback` contains only the *training* games of the current checkpoint (moves, engine annotations in chess, traces, session files), a score summary, and the protocol files (`SCHEMA.md`, the validator, and the parent inventory). Held-out games are never copied there.

```
Game transcripts, annotations, failures, per-game scratch files,
mutation feedback, and your current infrastructure are available under
/workspace. Persistent infrastructure is under /workspace/inventory. On
a retry, prior reflection logs are available under
/workspace/feedback/reflection_attempts. Improve the infrastructure for
future games.

You can create or modify Python tools, create or modify persistent
memory, create or modify strategies for using tools and memory, and
delete inventory that is no longer useful. Use the Bash tool to inspect
and edit /workspace/inventory. Invoke Bash directly rather than
describing or printing a tool call. Each Bash call may run for at most
1800.0 seconds. You have at most 50 reflection model calls and will be
told how many remain. Finish early when the work is complete. The
```

```
inventory is validated after the final call.

Maintain failure_modes.json and artifacts.json under
/workspace/inventory/tracking while inspecting the canonical
training trajectories under /workspace/feedback. Reuse a failure-mode ID
when the causal problem is the same; create a new ID only for a
materially different problem. Map failures to exact training game IDs. A
trajectory need not be mapped when you find no failure. Preserve
mitigated and merged modes rather than deleting them.

Maintain stable artifact IDs for persistent tools, memory, strategies,
instructions, and other infrastructure. Keep an ID when revising the
same capability, but create a new ID for a materially different
capability rather than repurposing an old ID. Record paths, executable
entrypoints, addressed failure modes, and active or retired status.
Every functional inventory file you create, modify, or retire must
belong to an artifact addressing at least one failure mode. Retired
artifacts must not leave implementation files in the active inventory.

Use these exact JSON shapes. `failure_modes.json` is `{"schema_version":
"failure_modes.v1","failure_modes":[{"id":"FM-0001","description":"...",
"status":"active","trajectory_ids":["checkpoint_000_train_s1_white"]}]}`
. Status is `active`, `mitigated`, or `merged`; only a merged entry adds
`"merged_into":"FM-..."`. `artifacts.json` is `{"schema_version":"artifa
cts.v1","artifacts":[{"id":"AR-0001","description":"...","kind":"tool","
paths":["tools/example.py"],"entrypoints":["tools/example.py:choose_move
"],"failure_mode_ids":["FM-0001"],"status":"active"}]}`. Kind is `tool`,
`memory`, `strategy`, `instruction`, or `other`; status is `active` or
`retired`; non-executable artifacts use an empty `entrypoints` list. The
same schemas and transition rules are in
/workspace/feedback/protocol/SCHEMA.md.

Before finishing, run the protocol validator
/workspace/feedback/protocol/validate_tracking.py and correct
structural errors before the final response.
```

Per-call budget message (reflection).

```
Reflection budget: {used}/{maximum} calls used; {remaining} remain.
```

Final reflection call. This replaces the budget message on the last permitted call.

```
Reflection budget: {used}/{maximum} calls used; {remaining} remains.
Final reflection call {turn}/{maximum}. Given the evidence so far, make
the best persistent infrastructure edits now. Invoke the Bash tool
directly for edits under /workspace/inventory rather than describing or
printing a tool call. The inventory is validated after this call.

Synchronize failure_modes.json and artifacts.json with the final
inventory, then run the protocol validator before finishing.
```

Retry addendum. If a reflection attempt is rejected (for example, the new library fails to load), the harness starts a new attempt with this text appended (up to 6 attempts in chess, 3 in Go and Hex). `{latest_error}` holds the rejection message; a rendered example from a Chess (Hard) run follows the template.

```
This is a retry after a rejected reflection attempt. Before editing,
read /workspace/feedback/reflection_retry_history.json and the prior
logs under /workspace/feedback/reflection_attempts. Resolve the recorded
validation error instead of repeating the rejected change.

Latest rejection: {latest_error}
```

Rendered example of the last line:

```
Latest rejection: ValueError: candidate infrastructure failed smoke
validation: persistent tool load failed: NameError: name 'object' is not
defined
```

Tool-call fallback. Most Go and Hex runs of models other than Claude Fable 5 go through an inference gateway, which appends this sentence to the system prompt.

```
Use structured tool calling whenever you call a tool. If structured tool
calling is unavailable, end your response with FINAL_TOOL_CALL_JSON: on
its own line, followed by exactly one JSON object in the form
{"functionCall":{"name":"tool_name","args":{}}}. Anything after that
marker is treated as a real tool call, so never use the marker in
examples.
```