RevengeBench: Reverse Engineering Code-Space Policies from Behavioral Experiments*

Babak Rahmani† **Sebastian Dziadzio*** **Joschka Strüber***
Sergio Hernández-Gutiérrez **Matthias Bethge**
Tübingen AI Center

Leaderboard Code Dataset

Abstract

For most of scientific history, researchers studying behavior could only infer hidden mechanisms from outward actions—an inverse problem that becomes more tractable when observation is augmented by targeted intervention. We pose a computational analogue: given only behavioral traces of an agent in a game environment, can a learner reconstruct the underlying decision program as executable code, and how much does this reconstruction improve with the ability to design controlled experiments? We introduce REVENGEBENCH, a benchmark of 75 LLM generated, Elo-calibrated policies across five game environments, drawn from CodeClash tournament trajectories. The learner observes the hidden target policy play against sampled opponents and designs behavioral probes in the form of custom opponent policies that elicit informative behavior. It then submits an executable hypothesis, which is evaluated using continuous action-distance metrics. We further validate that recovered code carries informative signal in downstream player-versus-player tournaments. Across twelve frontier LLMs, recovery quality varies substantially (34–72% of initial distance closed), with reconstructed policies yielding measurable competitive advantage, particularly for weaker models that otherwise struggle to design effective counter-strategies. Our benchmark positions behavioral recovery of programmatic policies as a tractable inverse problem in code-space, opening a path to opponent modeling, policy interpretability, and the broader question of inferring latent mechanisms from observations.

1 Introduction

Scientific discovery often starts from an asymmetry: effects are visible, but their causes are hidden. Researchers trying to understand animal behavior could not look inside the organisms they studied. Instead, they observed actions, designed experiments, and inferred the latent mechanisms—neural, hormonal, ultimately genetic—that produced what they saw. Such inverse problems become more tractable when passive observation can be paired with active intervention: the same observational evidence can support many candidate mechanisms, but targeted interventions help distinguish among them [Hacking, 1983]. We study a computational analogue: given only behavioral traces of an opaque policy in a game environment, can a learner reconstruct a program that faithfully reproduces the target's behavior? How much does reconstruction improve when the learner can actively probe the target through controlled interactions? Unlike most natural settings, where the true mechanism remains inaccessible, our framework holds the target policy fixed and measurable while varying the learner's epistemic access from passive observation to active probing.

*This is a workshop version of the paper. The full paper is available on arXiv at arXiv:2606.26094.

†Equal contribution. Correspondence to rahmani.b91@gmail.com.

Published at the Second Workshop on Agents in the Wild: Safety, Security, and Beyond (AIWILD) at ICML 2026.

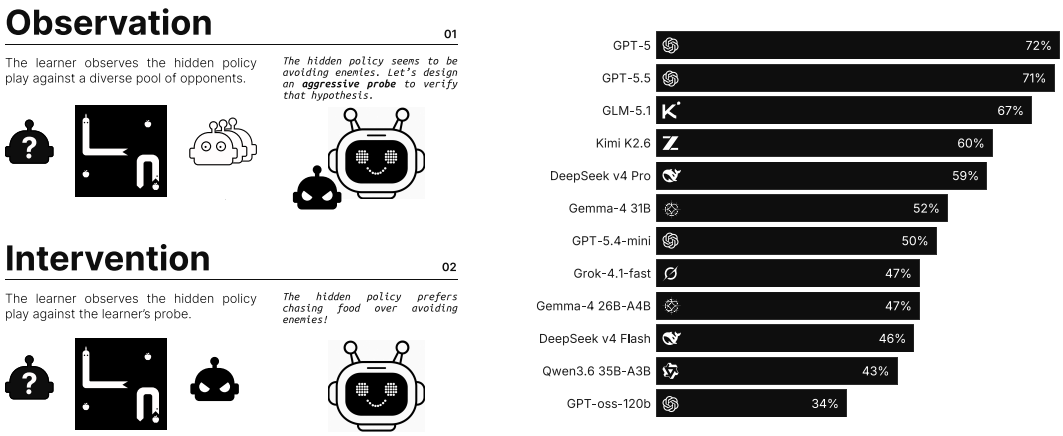


Figure 1: **Benchmark overview. Left:** the learner alternates between passively observing the hidden policy play against sampled opponents and actively probing it with self-authored opponents. **Right:** fraction of initial action distance closed (↑) for twelve frontier LLMs using `mini-SWE-agent`.

Inverse problems in agent modeling often assume one of two access regimes: a fixed corpus of demonstrations to imitate or invert [Abbeel and Ng, 2004, Ng and Russell, 2000, Torabi et al., 2018], or direct input-output examples with a query interface for arbitrary inputs, as in recent benchmarks for programming by example [Geng et al., 2025, Naik et al., 2025, Wei et al., 2025]. Real scientific inference is usually neither. A biologist, psychologist, or physicist cannot place a system in an arbitrary latent state. They can only perturb it through the channels the environment makes available, and observe the behavior those perturbations elicit. REVENGEBENCH models this intermediate epistemic regime. The learner cannot query arbitrary target states directly. Instead, it writes *probe opponents* whose interactions with the target induce informative trajectories. A probe is a hypothesis dressed as an opponent: a policy crafted to drive the target into desired regions of state space. The benchmark thus tests *constrained experimental inference over executable policies*: recovery of programs from behavior, with interventions limited by the dynamics of the environment itself.

Existing accounts of action understanding formalize this problem as recovering a posterior over discrete goals [Baker et al., 2009], a reward function [Abbeel and Ng, 2004, Ng and Russell, 2000], or a parametric policy [Torabi et al., 2018]. Each commits the latent mechanism in advance to a particular form, even though strategically important behavior may arise from routines, heuristics, or exceptions rather than from pure utility maximization. A recent line of work argues that behavior can be effectively modeled as executable code: a representation rich enough to encode procedure, memory, and conditionals, and precise enough to be tested at the level of individual actions [Jha et al., 2026]. It is already established that LLMs can write executable strategies in the forward direction: recent work synthesises counter-policies in code space [Bachrach et al., 2025, Hennes et al., 2026]. We adopt this representation but raise the bar for the learner: rather than weighting an ensemble of candidate programs [Jha et al., 2026], we ask for a single runnable policy, executed on the hidden target's own states across held-out trajectories.

We instantiate this in REVENGEBENCH, a benchmark of 75 executable policies across five game environments from CodeClash [Yang et al., 2025] spanning discrete, grid-based, sequential, multi-unit, and continuous control settings. The policies are written by LLMs for CodeClash competitions. We extract policies from multiple models and pitch them against one another to estimate Elo scores, giving a controlled difficulty axis. In each tournament, the learner observes the target playing against sampled opponents, optionally designs probe opponents, and submits an executable policy.

Figure 2 illustrates the inference process of the agent in a single trajectory. The learner moves from mismatch inspection to targeted probing, then revises hypotheses about food pursuit, safety filtering, tie-breaking, and reachable area. The action-distance curve shows that intermediate edits can temporarily worsen recovery before later evidence improves the hypothesis, exposing the iterative nature of behavioral inference.

We evaluate twelve frontier LLM coding agents on REVENGEBENCH. Our primary score is the fraction of action distance closed relative to an initial random policy. Recovery quality varies

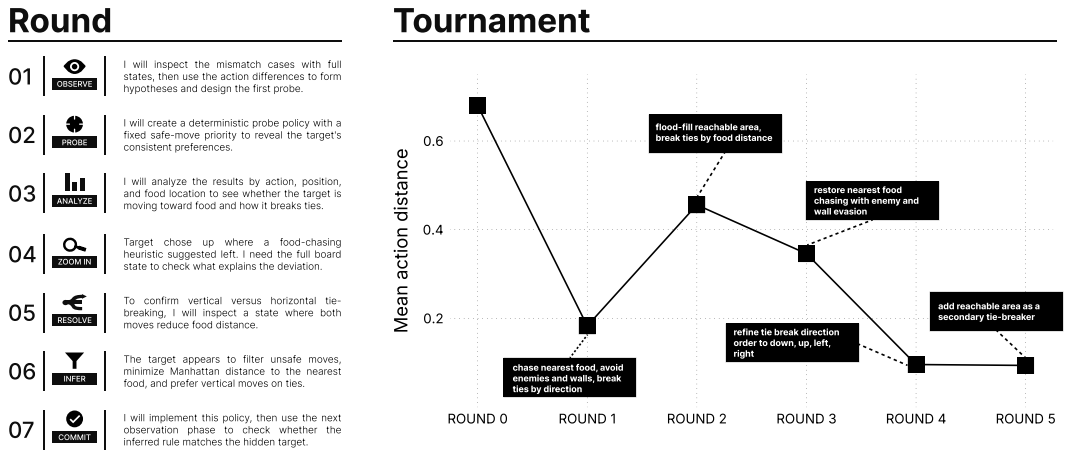


Figure 2: **Iterative reasoning in BattleSnake. Left:** the reasoning trace of GPT-5 within a single round, all quotes verbatim. **Right:** GPT-5.4-mini's per-round action distance and submitted strategy summaries: a good initial hypothesis is followed by a regression and a self-correction by round 5.

substantially: the best model closes 72% of initial behavioral distance, the weakest only 34%. Active probing helps some models and does not hurt others, indicating that intervention yields benefits when the learner is strong enough to use it. As a secondary check, we test whether behavioral fidelity carries strategic value: we give the recovered program to the same LLM and ask it to write a counter-policy against the target. Counter-policies designed from recovered code have measurable competitive advantage, especially for weaker models that otherwise fail to design effective counters, suggesting it extracts structure that models cannot synthesise from scratch.

Our contributions are threefold. (1) We introduce REVENGEBENCH, consisting of 75 hidden target policies and an evaluation pipeline. (2) We formalise a protocol in which learners infer executable policies from behavior, with intervention restricted to designing opponent policies that interact with the target through gameplay. (3) We evaluate twelve frontier LLM coding agents, showing that recovery closes 34–72% of random policy behavioral distance and that recovered code yields counter-policies that are effective against the target. The setting matters in two ways. Practically, as LLM agents are increasingly deployed in multi-agent workflows [OpenClaw, 2026], the ability to anticipate an opaque agent's behavior from observation has direct value. Methodologically, our benchmark opens a path towards understanding when observation is sufficient for LLM agents to recover a hidden mechanism, and where intervention is required.

2 Problem Setting

Our benchmark operationalizes the inverse task introduced in Section 1: a learner observes a hidden policy playing a game and must submit a runnable program that matches its behavior. Because the hidden policies are themselves executable, hypotheses can be scored mechanically against ground truth, a property that behavioral inverse problems generally lack.

Setting. Let $\mathcal{S}$ be a state space, $\mathcal{A}$ an action space, and $P(s_{t+1} \mid s_t, a_t)$ the environment dynamics. A *target* is an unknown, fixed policy $\pi^* : \mathcal{S} \to \mathcal{A}$, implemented as an executable strategy. The learner has no access to π^* or its internal representation. Interaction occurs through an opponent pool $\mathcal{O}$: for any opponent $o \in \mathcal{O}$, rolling out π^* against o yields a trajectory

$$\tau = (s_0, a_0, s_1, a_1, \ldots, s_T), \quad a_t = \pi^*(s_t).$$

The learner observes a collection of such trajectories and may optionally generate additional trajectories through controlled probing interactions.

Environment and arenas. We instantiate the task in five code-based arenas drawn from the CodeClash benchmark [Yang et al., 2025]: BattleSnake [Chung et al., 2020], Halite [Truell and Spector, 2016], Poker [Kumar et al., 2025], RoboCode [Hartness, 2004], and RobotRumble [Outkine and Oxer,

2020].[3] The five arenas span four programming languages (Python, C, Java, JavaScript) and diverse game mechanics: full versus partial observability, simultaneous versus sequential play, discrete versus continuous control, and action spaces ranging from a single discrete move to structured outputs over hundreds of board locations. Together, these arenas cover a wide range of policy complexity: the most elaborate policies reach roughly 500 lines of code and implement heuristics over evolving game state and opponent behavior (see Tables 4 and 7).

Our framework reuses the CodeClash codebase, which exposes a sandboxed Linux environment in a Docker container. The learner agent acts through the `mini-SWE-agent` scaffold [Yang et al., 2024] by issuing bash commands to read and write files, run scripts, and submit probes or candidate policies for scoring. We deliberately use a minimal harness; the trade-off is discussed in Appendix D.2.

Policy pool. We construct a pool of 40 policies for each arena using CodeClash tournament artifacts. We start from about 15,000 policy files across all arenas and filter them in several stages. First, we apply structural filters to eliminate policies that violate basic constraints. Next, we cluster the remaining files to remove near-duplicates and candidates that do not meet a threshold for semantic complexity. Finally, we verify that each remaining policy is runnable and supports self-play in the corresponding arena container. This process yields 40 validated policies per arena.

For each arena, the 15 highest-Elo policies among these 40 are designated as hidden targets (see Table 4). For each target, the other 39 policies form the opponent pool, from which 20 are sampled each tournament round. Restricting targets to the top tier excludes degenerate entries (e.g., always folding pre-flop in Poker). See Appendix B for the full construction pipeline.

Starter policy. Each arena has a simplistic but functional *starter policy* $\hat{\pi}_0$. The learner's hypothesis and probes start from this policy, and our main metric measures improvement relative to it.

Action distance and recovery metric. We score recovery using a per-game action distance $d_g : \mathcal{A}_g \times \mathcal{A}_g \to [0, 1]$. Each distance is symmetric, satisfies $d_g(a, a) = 0$, and equals 1 when either action is missing or invalid. Game-specific distances are aggregated over action components (e.g., Halite cells, RobotRumble units) to handle variable-size action outputs. A graded distance is essential: some policies contain quantitative heuristics (raise sizes, turn rates, per-cell move distributions) for which "right behavior, wrong magnitude" is different from "qualitatively wrong." For example, in Poker, betting 90 chips instead of 100 is much closer to the target behavior than folding, even though both would be incorrect under exact-match accuracy. Table 7 summarizes the arena-specific definitions of d_g, with full definitions in Appendix F. The mean per-step action distance on a trajectory τ is

$$D(\hat{\pi}, \pi^*; \tau) \;=\; \frac{1}{T} \sum_{t=1}^{T} d_g(\hat{\pi}(s_t),\, \pi^*(s_t))\,, \tag{1}$$

evaluated on held-out trajectories generated by π^* against opponents from $\mathcal{O}$. We report two summaries: $D_{\min}$ (↓), the lowest mean action distance achieved across rounds, and the distance reduction (↑) $\Delta = 1 - D_{\min}/D_0$, where D_0 is the mean action distance of the starter policy $\hat{\pi}_0$. Δ normalizes for differences in baseline difficulty across targets and arenas, and is used in Fig. 1.

Passive and active access. In the *passive* regime, the learner observes a fixed collection of trajectories generated by π^* against opponents sampled from $\mathcal{O}$, with fresh simulations added each round. In the *active* regime, the learner writes up to B probe opponents per round to elicit targeted behavior, for example forcing states where the current hypothesis $\hat{\pi}$ and π^* are likely to disagree.

Inference as a closed loop. The benchmark induces a closed-loop inference problem. Each round, the learner receives fresh trajectories, evaluates its current hypothesis against them, and refines its code. In the active regime, it may interleave probe-writing with refinement so that intervention results feed back into the same round. A more detailed description of the loop and the full version of the algorithm can be found in Appendix C.

3 Experimental Setup

We evaluate frontier language models on REVENGEBENCH across all five arenas (BattleSnake, Halite, Poker, RoboCode, and RobotRumble) with 15 targets per game, for a total of 75 targets. Unless

[3] CodeClash also includes CoreWar, which we exclude because its action representation does not admit the action-distance comparison required by our evaluation.

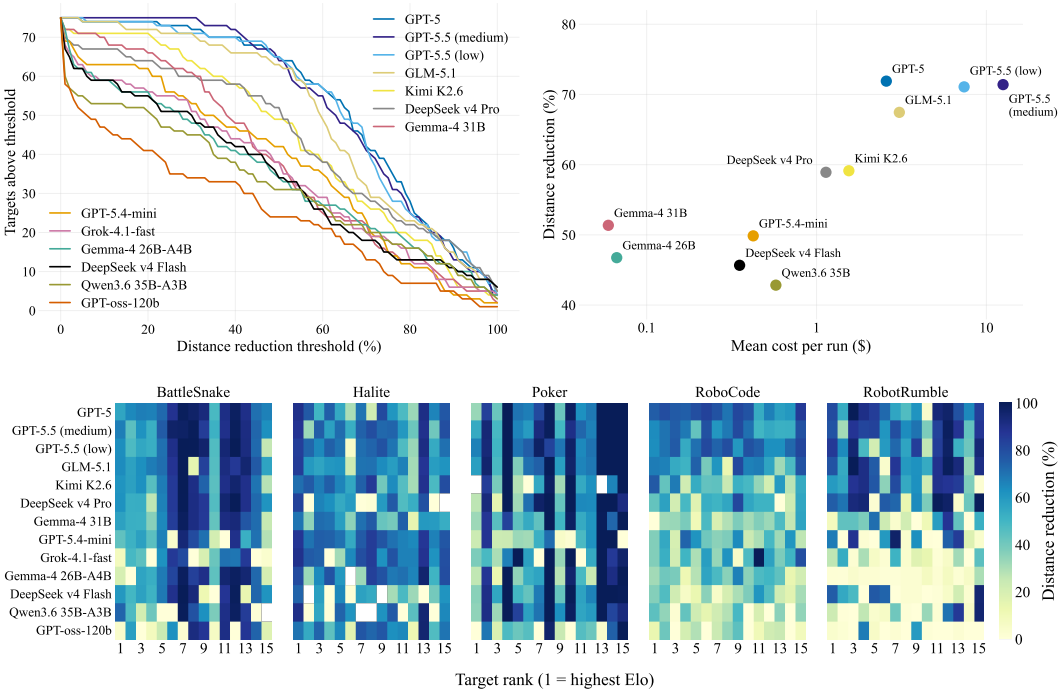


Figure 3: **Strategy recovery performance. Top left:** Cumulative distribution of distance reduction across all 75 targets. **Top right:** Cost–quality trade-off: mean API cost per run vs. distance reduction (%). **Bottom:** Per-model, per-game mean distance reduction (%).

otherwise stated, each run consists of R=5 rounds with 30 edit steps per round. In each round, the learner observes 20 opponent traces and may optionally use up to B=5 probes.

Agent. We use `mini-SWE-agent` [Yang et al., 2024], a minimal scaffold that operates inside a sandboxed Linux environment in a Docker container. It has a single tool, a bash terminal, through which it reads and writes files, runs evaluation scripts, and submits candidate policies. This minimal interface reduces scaffold-specific advantages, ensuring comparisons primarily reflect model capability rather than agent engineering. To compare agent scaffolds, we also evaluate Codex CLI [OpenAI, 2025] with GPT-5.5 [OpenAI, 2026] (high and low reasoning effort) on the same task suite (Appendix D).

Models. We evaluate twelve frontier models as learner agents: GPT-5.5, GPT-5 [Singh et al., 2025], GPT-5.4-mini, GPT-oss-120b, GLM-5.1, Gemma 4 31B, DeepSeek v4 Pro, DeepSeek v4 Flash, Gemma 4 26B-A4B, Kimi K2.6 [Kimi Team et al., 2025], and Qwen3.6 35B-A3B. Each model is run once per target, for a total of 75 runs per model. Gemma 4 31B is additionally run with five seeds per target for the robustness analysis (Section D). For the method-comparison and baseline experiments, we report results on GPT-5, GPT-5.4-mini, Gemma 4 31B, and DeepSeek v4 Pro. Closed-source models are accessed through their first-party APIs, and open-weight models through OpenRouter.

4 Results

4.1 Strategy recovery

Figure 1 shows the distance reduction Δ per model using the `mini-SWE-agent` scaffold. Recovery ranges from 71.9% (GPT-5) to 33.8% (GPT-oss-120b), with the top four models clustering above 68%, a mid-tier (Kimi, DeepSeek v4 Pro, Gemma-4 31B) recovering 52–60%, and the remaining models falling between 42–50%. All models substantially outperform the pool-mean baseline (Appendix E.1), confirming genuine strategy inference.

Recovery is widespread but not uniform (Figure 3). The strongest models fully close the behavioral gap ($\Delta \geq 99\%$) on a subset of targets (up to 7 of 75 for GPT-5.5-low). The cost–quality scatter (top right) reveals three distinct clusters: expensive closed models (GPT-5.5 low/medium, $7–13 per run),

Table 1: **Method comparison.** Mean±std best action distance (↓) across targets per game. *Pool Mean*: expected distance from sampling N=5 random pool strategies with no learning; *Probe*: full active method; *No Probe*: passive observation only; *Desc-Only*: natural-language summary input replacing raw traces [Hennes et al., 2026]; *BPI*: Bayesian best-of-N (N=5 to equalise compute budget) from a single observation round [Jha et al., 2026]. Cells marked "–" indicate invalid code generation by the model. Shaded rows denote the best method per model.

Model	Method	BattleSnake	Halite	Poker	RoboCode	RobotRumble	Mean
Pool Mean (no learning)		0.35±0.11	0.57±0.06	0.24±0.03	0.10±0.02	0.51±0.06	0.35
GPT-5	Probe (Active)	**0.15**±0.11	0.31±0.17	**0.05**±0.05	**0.08**±0.03	0.14±0.11	**0.15**
	No Probe	0.17±0.14	0.37±0.21	0.05±0.05	0.08±0.02	**0.12**±0.08	0.16
	Desc-Only	0.25±0.07	**0.20**±0.08	0.09±0.07	0.16±0.04	0.16±0.11	0.17
	BPI	0.27±0.11	0.32±0.22	0.15±0.14	0.17±0.06	0.22±0.22	0.23
GPT-5.4-mini	Probe (Active)	0.32±0.19	0.21±0.07	**0.08**±0.06	**0.17**±0.05	0.38±0.30	0.23
	No Probe	**0.21**±0.14	0.24±0.11	0.10±0.08	0.17±0.04	0.44±0.31	**0.23**
	Desc-Only	0.42±0.18	**0.20**±0.07	0.10±0.08	0.21±0.04	0.84±0.25	0.35
	BPI	0.47±0.17	0.25±0.16	0.16±0.24	0.26±0.04	**0.25**±0.16	0.28
DeepSeek v4 Pro	Probe (Active)	**0.19**±0.14	0.44±0.24	**0.05**±0.06	**0.16**±0.05	0.19±0.17	**0.20**
	No Probe	0.27±0.21	0.52±0.26	0.12±0.24	0.18±0.05	0.20±0.16	0.26
	Desc-Only	0.28±0.09	**0.20**±0.09	–	0.17±0.03	**0.14**±0.11	0.20
Gemma-4 31B	Probe (Active)	**0.21**±0.15	0.36±0.14	0.06±0.05	**0.19**±0.04	0.33±0.12	0.23
	No Probe	0.24±0.19	0.30±0.22	**0.05**±0.06	0.20±0.04	**0.31**±0.09	**0.22**
	Desc-Only	0.42±0.26	**0.29**±0.27	–	0.19±0.05	0.46±0.28	0.34

a mid-cost tier (GLM-5.1, GPT-5, Kimi, DeepSeek Pro at \$1–3), and a low-cost tier below \$1—where Gemma-4 31B achieves 51% recovery at \$0.06 per run. No meaningful correlation emerges between target playing strength and reverse-engineering difficulty (Spearman ρ=0.10).

Performance varies across arenas (Figure 3, bottom). Poker is the easiest (GPT-5.5-low reaches 82%; most models exceed 60%), while RobotRumble exhibits the widest model spread (5–72%). BattleSnake and RoboCode reward the strongest models, while weaker ones lag considerably. Halite reshuffles the ranking most: mid-tier models such as Gemma-4 26B-A4B (67%) and GPT-5.4-mini (73%) match or exceed GPT-5/5.5 (61–66%). Appendix G.1 provides the per-game breakdown.

4.2 Observation and intervention

Beyond final recovery, we evaluate how agent's observation mode and inference procedure affect performance. We compare the *Active* and *Passive* regimes from Section 2 with two additional baselines: *Description*, which replaces raw traces with natural-language behavioral summaries from an LLM summariser [Hennes et al., 2026], and *Bayesian Program Inference* (BPI), which replaces iterative refinement with best-of-N hypothesis generation followed by Bayesian-style selection [Jha et al., 2026]. All modes except BPI use the same iterative code-refinement loop. Implementation details are in Appendix E.

Active probing yields model-dependent gains. Table 1 shows that raw behavioral traces are the strongest observation channel. Passive refinement is already competitive: probing improves recovery in 16 of 20 model–game pairs (see Figure 9), but the benefit is concentrated in the stronger models. DeepSeek v4 Pro gains the most (mean boost +0.052, with the largest effects in BattleSnake, Halite and Poker), followed by GPT-5 (+0.013, four of five games). GPT-5.4-mini and Gemma 4 31B show near-zero average gains, as positive and negative effects cancel across games. We analyse the mechanisms behind these model-dependent probing gains in Section 5.

Baselines. Raw observations are difficult to compress or replace. Description-only inputs are sometimes competitive, especially on Halite, but generally lose fine-grained state–action evidence. BPI underperforms iterative refinement for GPT-5 and GPT-5.4-mini; DeepSeek v4 Pro and Gemma 4 31B were unable to submit valid code within BPI's single-round editing budget.

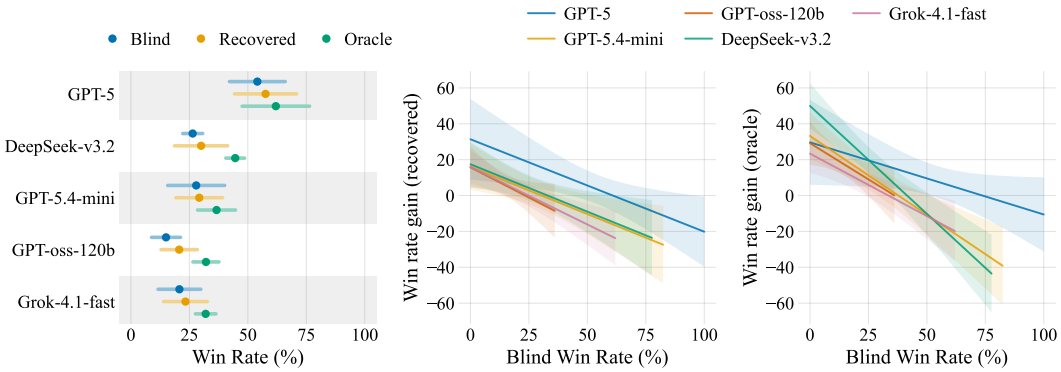


Figure 4: **PvP tournament**: a challenger agent writes a counter-strategy against a fixed target over 5 rounds, under three levels of opponent information (blind, recovered, oracle). Results are averaged over 20 targets (5 per game), 5 rounds, and 3 seeds per model. **Left:** Win rate per challenger model, averaged across games and rounds. Horizontal ticks show the cross-game mean. The ordering oracle > recovered > blind holds for every model is observed. **Middle, Right:** Each point is one (model, game, target) combination. Regression lines with 95% CI bands show that models with lower blind baselines gain more from opponent intelligence, for both oracle (middle) and recovered (right) conditions.

4.3 From Inference to Action

Strategy recovery produces a computational model of the target's decision procedure. But does possessing such a model confer a practical advantage? We test this by asking whether an agent armed with the recovered code can *win more often* against the target than one operating blind.

Setup. We run a player-versus-player (PvP) tournament in which a challenger LLM iteratively writes a counter-strategy against a fixed target over 5 rounds. The challenger sees one of three levels of opponent information: (a) **blind** (game rules only), (b) **recovered** (the code produced by our inverse pipeline), or (c) **oracle** (the target's true source code). Five challenger models of varying strength are evaluated across four games (BATTLESNAKE, HALITE, ROBOCODE, ROBOTRUMBLE) with 5 targets per game (20 targets total), yielding 540 tournament runs and ∼81K simulations (full protocol in Appendix H). We measure each challenger's win rate per round and define the *win rate gain* as the win-rate difference between a given condition (oracle or recovered) and blind ($\mathrm{WR}_{\mathrm{condition}} - \mathrm{WR}_{\mathrm{blind}}$).

Recovered policy carries useful signal. Figure 4 (left) shows that across all models the ordering *blind* < *recovered* < *oracle* holds consistently: access to the recovered strategy lifts win rates above blind play, and the oracle provides a further ceiling. The recovered code is imperfect, yet it carries enough actionable signal to produce measurable competitive advantage.

Weaker models benefit more. The value of opponent intelligence depends on the challenger's baseline ability. Figure 4 (middle, right) shows a significant negative correlation between blind win rate and win rate gain: models that perform poorly without opponent information derive the largest gains from it. Strong models discover competitive strategies through blind play in later rounds, eroding the initial advantage; weaker models never close this gap, so the win rate gain persists or grows (Appendix H.6, H.7).

5 Agent Behavior Analysis

Trajectory analysis. Following Yang et al. [2026], we classify each bash command into one of five action types by keyword matching in priority order: *Probe* (calling `PROBE_SUBMIT`), *Write* (create or edit files, e.g. heredocs, shell redirects, `sed -i`), *Execute* (execute non-probe programs such as `python3`), *Read* (inspect files, list directories, search), or *Other*.

Turn counts vary substantially across models (Figure 5). Three distinct behavioral clusters emerge. *High-throughput models* (DeepSeek v4 Flash, DeepSeek v4 Pro, GLM-5.1, Qwen3.6 35B-A3B) maintain near-constant activity at the 75-target ceiling across all five rounds, dominated by Execute

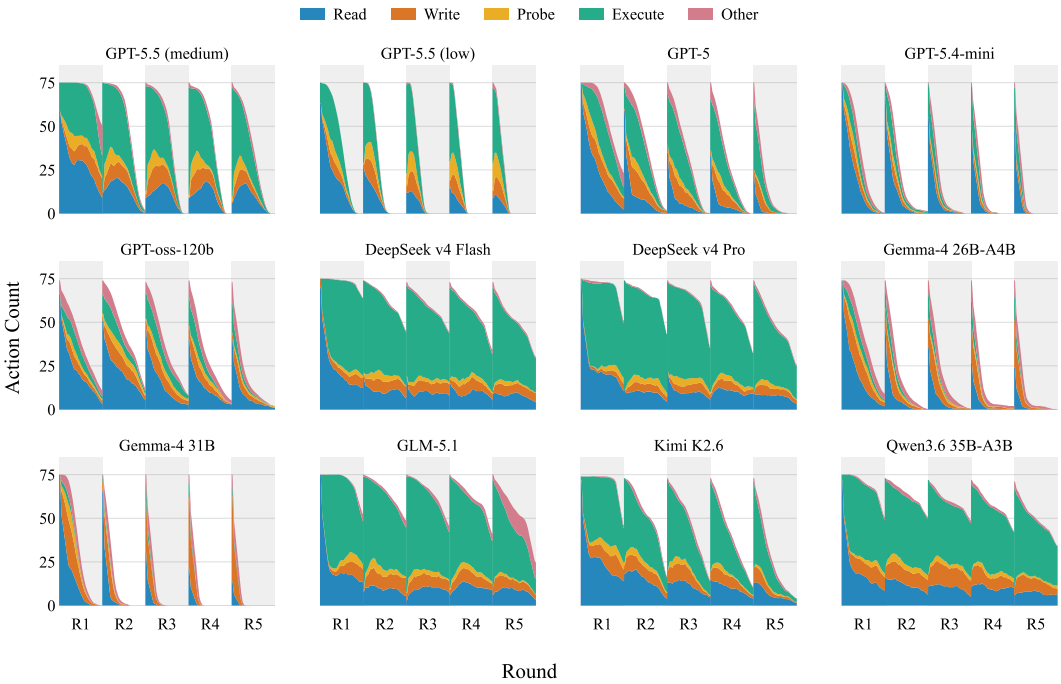


Figure 5: **Step-level action distribution across rounds** for twelve models over all 75 targets. Cumulative height reflects the number of trajectories still active at each step.

actions with Read as the secondary category; these models iterate rapidly through code-run-inspect cycles and routinely exhaust the 30-step budget per round. *Mid-throughput models* (GPT-5.5-medium, Kimi K2.6, GPT-5, GPT-5.5-low, GPT-oss-120b) show more varied action profiles and visible round-over-round decay as targets converge or exhaust: GPT-5.5-medium favours Execute and Read, while GPT-oss-120b inverts this pattern. *Low-throughput models* (Gemma-4 26B-A4B, GPT-5.4-mini, Gemma-4 31B) drop sharply within the first few steps of each round. Gemma-4 31B spends the plurality of turns on Write with very little Execute, suggesting it attempts large edits rather than incremental test-fix loops.

Why does probing help selectively? A probe is an executable experiment the learner must design, implement, and deploy as valid code.

Reliability varies: failure rates range from near-zero (GPT-5.5 variants) to the highest values for Grok-4.1-fast and Gemma-4 31B (Figure 8). *Design* matters more than volume: stronger models craft targeted experiments:

> **GPT-5** (BattleSnake): *a deterministic, safety-aware policy that avoids walls and occupied cells [...] This should lead to longer survival and more diverse states, giving better insight into the target's decision rules*

whereas weaker models default to minimal baselines (e.g. GPT-5.4-mini frequently submits a probe that *"always stays still"* or *"does nothing"* to elicit the target's behavior against a static reference).

Common failure modes. Trajectory inspection reveals three recurring failure patterns that affect all models to varying degrees.

Step-budget exhaustion. High-throughput models frequently hit the 30-step ceiling before submitting (68–74% of rounds for the high-; 38–39% for mid-throughput models). A common trigger is repeated analysis scripts that fail on the same bug:

> **GPT-oss-120b** (BattleSnake, round 2, step 24 of 30): *Identify mismatched entries where the target's chosen move does NOT maximize distance from opponent heads. Print those states to examine possible patterns...* → `SyntaxError` *(same error repeated at steps 26, 31)*

Low-throughput models (GPT-5.4-mini, Gemma-4 31B, Gemma-4 26B) almost never exhaust their budget ($\leq 3\%$ of rounds), as they issue far fewer commands.

Regression from best submission. Editing a submission can make it worse. Every model exhibits this: 16–55% of targets end with a final distance worse than the best achieved in an earlier round. Stronger models regress least, while weaker models regress in up to half their targets.

> **GPT-5** (BattleSnake, round 4): *The latest round shows a significant regression, suggesting my last change (vertical-only greedy and down>up>right>left fallback) diverges from the target's true behavior.*

Invalid submissions. Some models submit code that crashes at evaluation time. Grok-4.1-fast is the most affected, producing at least one evaluation failure in 44% of its runs, spread across all five games. GPT-oss-120b fails in 23% of runs, with errors in all games. The remaining models cluster at 0–8%, with failures concentrated in Halite.

> **GPT-5.4-mini** (Poker, round 5, after `nan` distance): *The latest bot caused an evaluation failure with NaN distance, so I need to inspect the current* `client/player.py` *for invalid expressions or bad return handling.*

6 Related Work and Ablations

Related work is discussed in Appendix A, where we position REVENGEBENCH relative to prior work on behavioral agent modelling, executable policy inference, and active experimental design. Additional ablations on key agent design choices are provided in Appendix D.

7 Limitations

Our setting is a controlled proxy for behavioral inference. The targets are synthetic, fixed game policies rather than adaptive or stochastic agents. They do not adapt across interactions, conceal strategy, or react to probing. Real-world agents may respond to observation or intervention, making reverse engineering substantially harder.

Action distance is only a proxy for policy equivalence. It measures behavioral fidelity on visited states, but rare or adversarially induced states may matter disproportionately, and small action differences can still cause large downstream effects. More fundamentally, behavioral recovery is not always identifiable: distinct programs may induce near-identical behavior under finite interaction budgets. Exact source recovery can therefore be underdetermined, with the recovered policy representing one member of a behaviorally consistent equivalence class rather than a unique solution.

Finally, benchmark reliability is arena-dependent. Our robustness analysis in appendix J shows that BattleSnake and Poker yield relatively stable rankings across runs and evaluators, while Halite, RoboCode, and RobotRumble exhibit substantially higher variance, requiring multi-run evaluation for fine-grained comparison. In addition, probing entangles reasoning, coding, debugging, and experimental design, making failure modes difficult to attribute to any single capability.

8 Conclusion

REVENGEBENCH studies whether LLM coding agents can recover executable policies from behavioral evidence. Across frontier models, recovery is substantial but far from solved. Our results suggest three main conclusions. First, passive behavioral evidence is often surprisingly informative: even without intervention, models recover a substantial fraction of target behavior, indicating that many programmatic strategies leave strong behavioral signatures in trajectory data. Second, active probing helps only when agents can use it effectively. Probing requires identifying uncertainty, designing informative interventions, implementing valid probes, and integrating the resulting evidence into policy revisions. Third, exact recovery is not necessary for downstream utility. Even imperfect reconstructions improve counter-strategy generation, especially for weaker challengers, suggesting that strategically useful opponent models need not faithfully reproduce the target source code. These findings position behavioral recovery of executable policies as a useful testbed for opponent modelling, policy interpretability, and experimental reasoning in LLM agents. Future work should extend this setting to adaptive agents that learn or strategically conceal behavior, richer domains such as web and scientific agents, and inference methods that recover distributions or equivalence classes of plausible policies rather than a single executable hypothesis.

References

P. Abbeel and A. Y. Ng. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the Twenty-First International Conference on Machine Learning*, ICML '04, page 1. Association for Computing Machinery, 2004.

E. Akata, L. Schulz, J. Coda-Forno, S. J. Oh, M. Bethge, and E. Schulz. Playing repeated games with large language models. *Nature Human Behaviour*, 9(7):1380–1390, 2025.

Y. Bachrach, E. Toledo, K. Hambardzumyan, D. Magka, M. Josifoski, M. Jiang, J. Foerster, R. Raileanu, T. Shavrina, N. Cancedda, et al. Combining code generating large language models and self-play to iteratively refine strategies in games. In *Thirty-Fourth International Joint Conference on Artificial Intelligence (IJCAI 2025), Montreal, Canada, August 16-22, 2025*, pages 10999–11003, 2025.

C. L. Baker, R. Saxe, and J. B. Tenenbaum. Action understanding as inverse planning. *Cognition*, 113(3):329–349, 2009.

D. Choudhury, S. Williamson, A. Golinski, N. Miao, F. B. Smith, M. Kirchhof, Y. Zhang, and T. Rainforth. BED-LLM: Intelligent information gathering with LLMs and bayesian experimental design. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=qyylZMLYT8.

J. Chung, A. Luo, X. Raffin, and S. Perry. Battlesnake challenge: A multi-agent reinforcement learning playground with human-in-the-loop. *arXiv preprint arXiv:2007.10504*, 2020.

A. Costarelli, M. Allen, R. Hauksson, G. Sodunke, S. Hariharan, C. Cheng, W. Li, J. M. Clymer, and A. Yadav. GameBench: Evaluating strategic reasoning abilities of LLM agents. In *Language Gamification - NeurIPS 2024 Workshop*, 2024. URL https://openreview.net/forum?id=qrzKE533Jr.

L. Cross, V. Xiang, A. Bhatia, D. Yamins, and N. Haber. Hypothetical minds: Scaffolding theory of mind for multi-agent tasks with large language models. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu, editors, *International Conference on Learning Representations*, volume 2025, pages 6507–6546, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/12f483f624b378f9f3058d8ecd3c7ff5-Paper-Conference.pdf.

J. Duan, R. Zhang, J. Diffenderfer, B. Kailkhura, L. Sun, E. Stengel-Eskin, M. Bansal, T. Chen, and K. Xu. GTBench: Uncovering the strategic reasoning capabilities of LLMs via game-theoretic evaluations. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=ypggxVWIv2.

J. A. Duque, M. Aghajohari, T. Cooijmans, razvan ciuca, T. Zhang, G. Gidel, and A. Courville. Advantage alignment algorithms. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=QFO1asgas2.

J. Foerster, R. Y. Chen, M. Al-Shedivat, S. Whiteson, P. Abbeel, and I. Mordatch. Learning with opponent-learning awareness. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*, AAMAS '18, page 122–130, Richland, SC, 2018. International Foundation for Autonomous Agents and Multiagent Systems.

R. Froger, P. Andrews, M. Bettini, A. Budhiraja, R. S. Cabral, V. Do, E. Garreau, J.-B. Gaya, H. Laurençon, M. Lecanu, et al. ARE: Scaling up agent environments and evaluations. *arXiv preprint arXiv:2509.17158*, 2025.

K. Gandhi, J.-P. Fraenken, T. Gerstenberg, and N. Goodman. Understanding social reasoning in language models with language models. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36, pages 13518–13529. Curran Associates, Inc., 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/2b9efb085d3829a2aadffab63ba206de-Paper-Datasets_and_Benchmarks.pdf.

J. Geng, H. Chen, D. Arumugam, and T. L. Griffiths. Are large language models reliable AI scientists? assessing reverse-engineering of black-box systems. In *LLM for Scientific Discovery: Reasoning, Assistance, and Collaboration*, 2025. URL https://openreview.net/forum?id=z8O4VP2iGI.

I. Hacking. *Representing and intervening: Introductory topics in the philosophy of natural science*. Cambridge University Press, 1983.

K. Hartness. Robocode: using games to teach artificial intelligence. *Journal of Computing Sciences in Colleges*, 19(4):287–291, 2004.

D. Hennes, Z. Li, J. Schultz, and M. Lanctot. Code-space response oracles: Generating interpretable multi-agent policies with large language models. In *Proceedings of the 25th International Conference on Autonomous Agents and Multiagent Systems*, AAMAS '26, page 3525–3527, Richland, SC, 2026. International Foundation for Autonomous Agents and Multiagent Systems. ISBN 9798400723179.

A. Hochlehnert, H. Bhatnagar, V. Udandarao, S. Albanie, A. Prabhu, and M. Bethge. A sober look at progress in language model reasoning: Pitfalls and paths to reproducibility. In *Second Conference on Language Modeling*, 2025. URL https://openreview.net/forum?id=90UrTTxp5O.

K. Jha, A. Y. Huang, E. Ye, N. Jaques, and M. Kleiman-Weiner. Modeling others' minds as code. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=vHXo7xIer6.

Y. Jing, B. Liu, K. Li, Y. Zang, H. Fu, Q. FU, J. Xing, and J. Cheng. Opponent modeling with in-context search. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=bGhsbfyg3b.

A. Khan, T. Willi, N. Kwan, A. Tacchetti, C. Lu, E. Grefenstette, T. Rocktäschel, and J. Foerster. Scaling opponent shaping to high dimensional games. In *Proceedings of the 23rd International Conference on Autonomous Agents and Multiagent Systems*, AAMAS '24, page 1001–1010, Richland, SC, 2024. International Foundation for Autonomous Agents and Multiagent Systems. ISBN 9798400704864.

H. Kim, M. Sclar, T. Zhi-Xuan, L. Ying, S. Levine, Y. Liu, J. B. Tenenbaum, and Y. Choi. Hypothesis-driven theory-of-mind reasoning for large language models. In *Second Conference on Language Modeling*, 2025. URL https://openreview.net/forum?id=yGQqTuSJPK.

Kimi Team, Y. Bai, Y. Bao, Y. Charles, C. Chen, G. Chen, H. Chen, H. Chen, J. Chen, N. Chen, et al. Kimi K2: Open agentic intelligence. *arXiv preprint arXiv:2507.20534*, 2025.

B. Kumar, H. Nguyen, and R. Jin. Husky hold'em bench, 2025. URL https://huskybench.com/. Accessed: 2025-09-08.

H. Li, Y. Chong, S. Stepputtis, J. P. Campbell, D. Hughes, C. Lewis, and K. Sycara. Theory of mind for multi-agent collaboration via large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 180–192, 2023.

X. Liu, D. Wang, Z. Feng, and A. Mehta. Scaling inference-time computation via opponent simulation: Enabling online strategic adaptation in repeated negotiation. In *Workshop on Multi-Agent Learning and Its Opportunities in the Era of Generative AI*, 2026. URL https://openreview.net/forum?id=pBRFXoYfhj.

C. Lu, T. Willi, C. A. S. De Witt, and J. Foerster. Model-free opponent shaping. In *International Conference on Machine Learning*, pages 14398–14411. PMLR, 2022.

A. Naik, Y. Mathur, D. Agrawal, M. Kapadnis, Y. An, C. Marr, C. Rose, D. Mortensen, et al. PBEBench: A multi-step programming by examples reasoning benchmark inspired by historical linguistics. *arXiv preprint arXiv:2505.23126*, 2025.

S. Nashed and S. Zilberstein. A survey of opponent modeling in adversarial domains. *J. Artif. Int. Res.*, 73, May 2022. ISSN 1076-9757. URL https://doi.org/10.1613/jair.1.12889.

A. Y. Ng and S. Russell. Algorithms for inverse reinforcement learning. In P. Langley, editor, *Proceedings of the Seventeenth International Conference on Machine Learning (ICML 2000), Stanford University, Stanford, CA, USA, June 29 - July 2, 2000*, pages 663–670. Morgan Kaufmann, 2000.

OpenAI. Codex CLI, 2025. URL https://github.com/openai/codex.

OpenAI. GPT-5.5 system card. Technical report, OpenAI, 2026. URL https://openai.com/index/gpt-5-5-system-card/. Technical report describing the GPT-5.5 model family.

OpenClaw. OpenClaw. https://github.com/openclaw/openclaw, 2026. Open-source personal AI assistant, version 2026.4.27, accessed 2026-04-30.

A. Outkine and N. Oxer. Robot rumble, 2020. URL https://robotrumble.org/.

G. Piatti, Z. Jin, M. Kleiman-Weiner, B. Schölkopf, M. Sachan, and R. Mihalcea. Cooperate or collapse: Emergence of sustainable cooperation in a society of LLM agents. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=0zWzJj6lO3.

N. Rabinowitz, F. Perbet, F. Song, C. Zhang, S. M. A. Eslami, and M. Botvinick. Machine theory of mind. In J. Dy and A. Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 4218–4227. PMLR, 10–15 Jul 2018.

M. Riemer, Z. Ashktorab, D. Bouneffouf, P. Das, M. Liu, J. D. Weisz, and M. Campbell. Position: Theory of mind benchmarks are broken for large language models. In *Forty-second International Conference on Machine Learning Position Paper Track*, 2025. URL https://openreview.net/forum?id=BCP8UU2BcU.

B. Romera-Paredes, M. Barekatain, A. Novikov, M. Balog, M. P. Kumar, E. Dupont, F. J. Ruiz, J. S. Ellenberg, P. Wang, O. Fawzi, et al. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.

P. Shojaee, K. Meidani, S. Gupta, A. B. Farimani, and C. K. Reddy. LLM-SR: Scientific equation discovery via programming with large language models. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=m2nmp8P5in.

A. Singh, A. Fry, A. Perelman, A. Tart, A. Ganesh, A. El-Kishky, A. McLaughlin, A. Low, A. Ostrow, A. Ananthram, et al. OpenAI GPT-5 system card. *arXiv preprint arXiv:2601.03267*, 2025.

H. Tang, D. Key, and K. Ellis. WorldCoder, a model-based LLM agent: Building world models by writing code and interacting with the environment. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems*, volume 37, pages 70148–70212. Curran Associates, Inc., 2024. doi: 10.52202/079017-2243. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/820c61a0cd419163ccbd2c33b268816e-Paper-Conference.pdf.

F. Torabi, G. Warnell, and P. Stone. Behavioral cloning from observation. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, IJCAI'18, page 4950–4957. AAAI Press, 2018. ISBN 9780999241127.

M. Truell and B. Spector. Halite: Two sigma's first artificial intelligence programming challenge, 2016. URL https://github.com/HaliteChallenge/Halite.

S. I. Wang. Measuring all the noises of LLM evals. *arXiv preprint arXiv:2512.21326*, 2026.

A. Wei, T. Suresh, J. Cao, N. Kannan, Y. Wu, K. Yan, T. S. F. X. Teixeira, K. Wang, and A. Aiken. CodeARC: Benchmarking reasoning capabilities of LLM agents for inductive program synthesis. In *Second Conference on Language Modeling*, 2025. URL https://openreview.net/forum?id=Q5pVZCrrKr.

Y. Yamada, R. T. Lange, C. Lu, S. Hu, C. Lu, J. Foerster, J. Clune, and D. Ha. The AI scientist-v2: Workshop-level automated scientific discovery via agentic tree search. *arXiv preprint arXiv:2504.08066*, 2025.

J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. R. Narasimhan, and O. Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://arxiv.org/abs/2405.15793.

J. Yang, K. Lieret, J. Yang, C. E. Jimenez, O. Press, L. Schmidt, and D. Yang. CodeClash: Benchmarking goal-oriented software engineering. *arXiv preprint arXiv:2511.00839*, 2025.

J. Yang, K. Lieret, J. Ma, P. Thakkar, D. Pedchenko, S. Sootla, E. McMilin, P. Yin, R. Hou, G. Synnaeve, et al. ProgramBench: Can language models rebuild programs from scratch? *arXiv preprint arXiv:2605.03546*, 2026.

C. Yin, T. Wu, Y. Shu, A. Gu, Y. H. Wang, J. Shao, xun jiang, and P. Li. Investigating advanced reasoning of large language models via black-box interaction. In *First Workshop on Foundations of Reasoning in Language Models*, 2025. URL https://openreview.net/forum?id=h8q694mExC.

X. Yu, W. Zhang, and Z. Lu. LLM-based explicit models of opponents for multi-agent games. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 892–911, 2025.

Y. Zhang, J. Lu, and N. Jaitly. Probing the multi-turn planning capabilities of llms via 20 question games. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1495–1516, 2024.

Z. Zhang, C. Jin, M. Y. Jia, S. Zhang, and T. Shu. AutoToM: Scaling model-based mental inference via automated agent modeling. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2026. URL https://openreview.net/forum?id=oeZZusZheP.

Acknowledgments and Disclosure of Funding

BR, SD, JS, SH, and MB acknowledge support by the Tübingen AI Center. BR acknowledges funding from the European Union's Horizon Europe research and innovation program under the Marie Skłodowska-Curie grant agreement No. 101151549. SD, JS, and SH thank the International Max Planck Research School for Intelligent Systems (IMPRS-IS) for support. JS and SH acknowledge funding by the Federal Ministry of Research, Technology and Space (BMFTR), FKZ: 16IS24085B. MB acknowledges Coefficient Giving funded by the Good Ventures Foundation. MB acknowledges funding by the Federal Ministry of Research, Technology and Space (BMFTR), FKZ: 16IS24079A. MB is a member of the Machine Learning Cluster of Excellence, funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC number 2064/1 – Project number 390727645.

Broader impact

The techniques that yield interpretable opponent models for coordination and scientific-inference studies could also be used to reverse-engineer proprietary decision systems, or, if extended beyond programmatic targets, to infer hidden processes from human behavior. We limit the benchmark to synthetic game policies and release artifacts under terms that discourage adversarial deployment; broader applications would require additional safeguards.

Appendix Outline

A. Extended Related Work
B. Strategy Pool Construction
C. Inference Strategy Loop: Full Pseudocode
D. Ablations
E. Baselines
F. Per-Game Action Distances
G. Extended Results
H. From Inference to Action
I. Context History Management
J. Noise-decomposition Methodology
K. Cost Analysis

A Extended Related Work

We group prior work into three themes most relevant to REVENGEBENCH: reasoning about other agents from behavior, representing policies as executable code, and active experimentation for hidden-system inference.

Theory of mind, opponent modelling, and strategic-agent evaluation. A large body of work studies whether models can infer the latent states, goals, or policies of other agents from observed behavior. Early machine theory-of-mind (ToM) models framed this as inferring an agent's goals and future actions from trajectories [Rabinowitz et al., 2018]. More recent LLM work has mostly evaluated ToM through static or semi-static text settings, including causal-template vignettes [Gandhi et al., 2023], cooperative games with explicit belief states [Li et al., 2023], and inference-time methods that combine LLM hypothesis generation with Bayesian or model-based reasoning over latent mental states [Kim et al., 2025, Zhang et al., 2026]. Related work also explores structured scaffolds for theory-of-mind reasoning in multi-agent settings [Cross et al., 2025]. A common critique is that many such benchmarks test whether models can answer theory-of-mind questions, rather than whether inferred beliefs improve decisions in interaction [Riemer et al., 2025].

Opponent modelling studies the related problem of inferring an opponent's policy from observed actions to improve decision-making [Nashed and Zilberstein, 2022]. Classical approaches include

opponent-aware learning and shaping in strategic games [Duque et al., 2025, Foerster et al., 2018, Khan et al., 2024, Lu et al., 2022], while more recent methods use in-context search over possible opponent policies [Jing et al., 2024] or explicit LLM-based opponent models in social and negotiation settings [Liu et al., 2026, Yu et al., 2025]. In parallel, strategic-agent benchmarks evaluate how well LLMs reason in repeated, game-theoretic, or multi-agent interactions [Akata et al., 2025, Costarelli et al., 2024, Duan et al., 2024, Piatti et al., 2024], with recent infrastructure enabling larger-scale agent evaluations [Froger et al., 2025].

REVENGEBENCH differs from these lines of work in two ways. First, the inferred model is not latent or textual but an executable policy, making it directly testable and inspectable. Second, we evaluate whether the recovered model transfers to action: Section 4.3 measures whether code inferred from behavior improves downstream counter-strategy generation.

Code-space policies and programmatic models. A growing line of work represents policies and latent mechanisms directly as executable code. In code-based games, LLMs can generate and iteratively refine policies through self-play [Bachrach et al., 2025] or response-oracle methods [Hennes et al., 2026]. These approaches study the forward direction: synthesizing policies that perform well in competitive environments.

Program-synthesis work applies the same representation more broadly. WorldCoder builds environment models by writing executable code and interacting with an environment [Tang et al., 2024]. LLM-SR frames scientific equation discovery as program synthesis [Shojaee et al., 2025]. Most closely related, Jha et al. [2026] model other agents' minds as code by generating candidate programs and scoring them against observed behavior.

REVENGEBENCH adopts the same premise that code is a useful representation for latent decision procedures, but studies the inverse problem under a different inference regime. Rather than selecting among independently generated program hypotheses, the learner iteratively edits a single executable policy and evaluates it against behavioral mismatches on held-out states. This makes recovery a closed-loop process of hypothesis revision rather than passive candidate selection.

Active experimentation and scientific discovery. REVENGEBENCH is also related to LLM-driven scientific discovery, where models propose hypotheses, run experiments, and use evaluators to search for hidden mechanisms or solutions. Examples include program-search systems for mathematical discovery [Romera-Paredes et al., 2024] and end-to-end agentic pipelines for automated scientific discovery [Yamada et al., 2025]. Active experimentation benchmarks more directly test whether LLMs can identify hidden systems by choosing informative interventions or queries [Geng et al., 2025, Wei et al., 2025, Yin et al., 2025]. Related work applies Bayesian experimental design to LLM-based question selection [Choudhury et al., 2026] or uses multi-turn deduction games to study information gathering [Zhang et al., 2024].

These works motivate our active-probing protocol: interventions can reveal distinctions that passive observations may miss. The key difference is the access regime. In many prior settings, the learner can query arbitrary inputs or directly choose experiments. In REVENGEBENCH, interventions are constrained by the environment. The learner cannot place the target in arbitrary states, but must instead write probe opponents whose interactions induce informative trajectories. This makes the setting closer to experimental design in natural systems, where interventions are mediated by the dynamics of the environment itself.

B Strategy Pool Construction

Our inverse strategy benchmark is built from a large corpus of strategies collected from public CodeClash tournament artifacts Yang et al. [2025]. The goal of the construction pipeline is to obtain, for each game, a pool of executable and behaviorally diverse strategies that span a range of coding styles, structural complexity, and competitive strength. This appendix summarizes the data source, filtering pipeline, and pool selection procedure.

Table 2: Number of strategies retained at each stage of the strategy construction pipeline. Structural filtering removes invalid submissions, deduplication retains one strategy per similarity cluster, stratified selection builds the candidate pool, and Elo calibration defines the final target set.

Game	Language	Raw corpus	Structural filtering	Deduplication + selection	Elo calibrated targets
BattleSnake	Python	4,372	438	40	15
Halite	C	1,743	622	40	15
Poker	Python	3,064	477	40	15
RoboCode	Java	1,676	522	40	15
RobotRumble	JavaScript	3,603	746	40	15
Total	–	14,458	2,805	200	75

B.1 Source Corpus

We crawl strategy code from tournament artifacts hosted on the CodeClash website.[4] We extract the final submitted strategy files for each participating model and game. The resulting corpus contains 14,891 raw strategy files across 1,901 tournament artifacts, covering six games and eight source models. Table 2 summarizes the corpus by game.

The six games span multiple programming languages and game mechanics: BattleSnake and Poker in Python, CoreWar in Redcode, Halite in C, RoboCode in Java, and RobotRumble in JavaScript.

B.2 Filtering and Selection Pipeline

We construct the final strategy pools using a three stage pipeline: structural filtering, deduplication and stratified selection, and Elo calibration. The pipeline is game agnostic at the top level, with game specific validators and feature extractors used where needed.

Stage 1: Structural filtering. We first remove strategies that are clearly unusable without running full game simulations. A strategy must satisfy basic structural constraints such as the presence of the expected submission file, minimum code length, valid syntax, required handlers, and a valid runtime entrypoint. Additional game specific checks are applied when necessary. This stage reduces the raw corpus from 14,891 files to 3,314 structurally valid strategies.

Stage 2: Deduplication and candidate selection. Many CodeClash submissions are near duplicates, for example minor edits of the same underlying strategy or repeated generations with small formatting changes. To avoid over representing such families, we cluster strategies using cosine similarity over language appropriate feature representations. For Python games we use AST based feature vectors that capture node distributions, call patterns, control flow, and selected algorithmic motifs. For the other languages we use a combination of keyword indicators and hashed line shingles to represent structural similarity.

Within each game, we greedily construct a set of representatives by sorting strategies by a complexity score and keeping a new representative only when its similarity to all existing representatives is below a game specific threshold. The threshold is calibrated automatically from the empirical similarity distribution for that game. This yields 1,242 representative clusters across the six games.

Each representative is assigned a composite complexity score intended to capture structural richness rather than empirical win rate. The score combines factors such as code length, number of functions, nesting depth, branching structure, use of algorithmic patterns, and data structure diversity, with game specific feature extraction where needed. We then bucket strategies into low, medium, and high complexity tiers by percentile within each game.

To form the candidate pool for each game, we perform deterministic stratified sampling across source model and complexity tier. This encourages diversity both in where the strategy came from and in how sophisticated its code structure appears. For each game, we select 40 candidates. We exclude CoreWar from the final benchmark because its action representation does not admit the action-distance comparison our evaluation requires.

[4] https://codeclash.ai/

Validation after selection. After candidate selection, we apply a second validation gate to the populated pool to make sure the strategies run in the environment. We do this via self play inside the game Docker environment. If a selected strategy is found to be broken, it is replaced by the next available spare from the same deduplicated candidate set. This post selection validation is important because some failures, such as missing runtime assumptions or compilation issues, are not always detected by initial structural checks alone.

Stage 3: Elo calibration and target selection. The 40 selected candidates for each arena are rated using a Swiss-system Elo tournament. In each round, strategies are paired against opponents with similar current ratings, and Elo is updated from head-to-head outcomes. We use 15 rounds with 10 simulations per match.

After rating, the 15 highest-Elo strategies per arena are designated as hidden targets. We restrict targets to the strongest policies because lower-rated candidates frequently exhibit failure modes that make them uninteresting as inverse-learning targets: their decision rule may be trivial (a single conditional or a constant action) or pathologically degenerate (for example, always folding pre-flop in poker, which produces deterministic play that loses predictably without exposing strategic structure). Restricting to the top tier avoids both pitfalls and provides headroom as frontier models continue to improve.

The remaining 39 strategies in each arena serve as the opponent pool. For each target, 20 opponents are sampled per tournament round to generate the behavioral traces the learner observes.

Table 3: Summary of Elo calibration tournaments for the 40 selected strategies in each game, ordered by Elo spread (highest to lowest).

Game	Matches	Elo range
Poker	300	1285–1642
RobotRumble	300	1365–1692
Halite	300	1334–1652
BattleSnake	300	1298–1574
RoboCode	300	1412–1584
Total	1,500	

Target statistics. Table 4 summarizes the 75 hidden target policies selected after Elo calibration (15 per arena), including code length in non-empty lines of code and Elo ranges.

Table 4: **Target policy statistics.** 75 CodeClash targets (top-15 Elo per game): code length in non-empty LoC and Elo ranges.

Game	Language	#LoC			Elo Range
		Min	Max	Med.	
BattleSnake	Python	139	443	234	1509–1599
Halite	C	49	195	86	1505–1652
Poker	Python	85	359	227	1519–1642
RoboCode	Java	49	442	125	1505–1584
RobotRumble	JS	29	245	74	1515–1691

C Inference Strategy Loop: Full Pseudocode

Algorithm 1 gives the full inverse-strategy procedure. It makes explicit three implementation details that are otherwise hidden inside the refinement loop.

Mismatch set $\mathcal{M}_r$. At the start of each round, we evaluate the previous hypothesis $\hat{\pi}_{r-1}$ against the current-round trajectories $\mathcal{T}_r$ and collect the (state, predicted action, target action) tuples on which the two disagree. The agent receives $\mathcal{M}_r$ alongside the raw traces as log files that can be read, giving it a

Algorithm 1 Inverse Strategy

Require: Black-box target π^*, opponent pool $\mathcal{O}$, rounds R, probe budget B
Ensure: Executable hypothesis $\hat{\pi}$

Initialize $\hat{\pi}_0$
for $r = 1, \ldots, R$ **do**
  Observe: Sample opponents $O_r \subset \mathcal{O}$; collect traces $\mathcal{T}_r \leftarrow \textsc{Interact}(\pi^*, O_r)$
  Evaluate: $\mathcal{M}_r \leftarrow \{(s_t, \hat{\pi}_{r-1}(s_t), \pi^*(s_t)) : s_t \in \mathcal{T}_r,\ \hat{\pi}_{r-1}(s_t) \neq \pi^*(s_t)\}$
  Refine: $\hat{\pi}_r \leftarrow \hat{\pi}_{r-1}$; $b_r \leftarrow 0$
  repeat ▷ agent editing steps
    $\hat{\pi}_r \leftarrow \textsc{Edit}(\hat{\pi}_r,\ \mathcal{T}_{1:r},\ \mathcal{M}_r)$
    if agent requests probe **and** $b_r < B$ **then** ▷ active probing
      Write probe o; $\tau \leftarrow \textsc{Interact}(\pi^*, o)$; $b_r \leftarrow b_r + 1$
      $\mathcal{T}_r \leftarrow \mathcal{T}_r \cup \{\tau\}$; recompute $\mathcal{M}_r$
    end if
  until step budget exhausted
end for
return $\hat{\pi}_R$

focused signal of where its code is currently wrong without forcing it to re-derive the disagreements from logs.

Step budget. Refinement within a round runs until a per-round step budget is exhausted. A step is any tool call: editing a file, running an evaluation script, reading a log, or submitting a probe. This bounds the compute the agent can spend per round and forces it to allocate effort between exploration and refinement.

Probe-edit interleaving. The probe budget B is separate from the step budget. Each probe costs one unit of B, triggers a new INTERACT call against π^*, appends the resulting trace to $\mathcal{T}_r$, and recomputes $\mathcal{M}_r$. Crucially, probes are issued from inside the same refinement loop as edits: the agent can write a probe, observe the target's response, and incorporate that response into its next edit within the same round.

D Ablations

D.1 Full conversation history outperforms per-round reset

Our agent carries its full conversation history across rounds, with older observations compacted via rule-based summaries (Appendix I).

Clearing the history each round consistently degrades recovery (Table 5). Both conditions retain a second, agent-driven channel of round-to-round memory [Yang et al., 2025], so the gap reflects information that persistent context carries beyond the agent's own summaries. Compaction reduces API cost by 11–22% without measurable performance loss (Appendix I.3).

Table 5: **Persistent vs. reset**: fraction of initial action distance closed (%).

Model	Pers.	Reset
GPT-5	71.9	67.2
GPT-5.4-mini	49.9	46.6
Grok-4.1-fast	47.2	46.1
GPT-oss-120b	33.8	30.8

D.2 Scaffold and reasoning effort: which matters more?

One concern when evaluating LLMs as agents is the influence of the scaffold and reasoning effort. We address this with an ablation that fixes the model to GPT-5.5 and varies the scaffold (`mini-SWE-agent` vs. Codex CLI) and the reasoning effort (low vs. high), evaluating on the same 15-target pool. Table 6 summarises mean best action distances per game.

Both factors matter comparably. Switching from `mini-SWE-agent` to Codex at low effort lowers the cross-game mean from 0.15 to 0.12. Raising effort from low to high within Codex lowers it

Table 6: **Codex vs. mini-SWE-agent.** Mean±std best action distance (↓) across 15 pool targets per game. *Low* / *high* refer to the model's reasoning effort.

Method	BattleSnake	Halite	Poker	RoboCode	RobotRumble	Mean
GPT-5.5 low (`mini-SWE-agent`)	0.17±0.15	0.29±0.17	0.03±0.03	0.09±0.02	0.17±0.12	0.15
GPT-5.5 low (Codex)	0.14±0.13	0.26±0.13	0.03±0.04	0.11±0.03	0.09±0.09	0.12
GPT-5.5 high (Codex)	**0.13**±0.12	**0.14**±0.13	**0.02**±0.03	**0.07**±0.03	**0.05**±0.07	**0.08**

further to 0.08. The two effects are similar and stack additively for a total relative reduction of 47% from mini-SWE-low to Codex-high.

mini-SWE-agent supports cross-model comparison. Because the Codex CLI is restricted to GPT models, we use `mini-SWE-agent` as the common scaffold and report the Codex result as a per-model upper-bound point for GPT-5.5.

Higher effort helps everywhere. At fixed harness, high effort improves every game (Halite halves: $0.26 \rightarrow 0.14$; three games reach ≤ 0.07).

E Baselines

We include three baselines to isolate the contributions of different components of the inverse-strategy pipeline: (1) a pool-mean baseline that requires no learning, (2) a description baseline that replaces raw traces with natural-language summaries [Hennes et al., 2026], and (3) a Bayesian Program Inference (BPI) baseline that replaces iterative refinement with best-of-N hypothesis selection [Jha et al., 2026].

E.1 Pool-mean baseline

The pool-mean baseline estimates the expected recovery achievable by randomly selecting an existing strategy from the pool without any learning. For each target, we sample 5 strategies uniformly at random from the 40-strategy pool (excluding the target) and evaluate each against the target by computing the action distance. The best (lowest) distance among the 5 samples is recorded. We then compute the normalised recovery score for model evaluation. The reported pool-mean score is the average over all targets. This baseline quantifies how much of the behavioral gap can be closed by chance similarity between pool strategies and the target, without any inference or code synthesis.

E.2 Description-only baseline

Description tests whether raw behavioral traces can be replaced by a compact natural-language observation channel, inspired by description-input code generation [Hennes et al., 2026]. After each round, a GPT-5.4-mini summariser converts sampled target behavior into a structured description of apparent decision rules. The learner receives only these summaries and refines its hypothesis code over R=5 rounds. It has no access to raw trace files and no probing budget. This compares raw-trace refinement against a passive summarise-then-refine pipeline.

Summariser. For each round, a GPT-5.4-mini summariser receives a sampled set of mismatch examples, i.e. state–action cases where the learner's action differs from the target's action. It produces a concise structured report of the target's apparent decision rules, including state-dependent behaviors, opponent-specific responses, threshold-like patterns, and consistent action preferences. The report is written as a text file in the learner's observation directory and remains available in later rounds; the underlying behavioral traces are not exposed.

Learner configuration. The learner uses the same initial code template, system prompt, editing interface, and round structure as the main method, but active probing is disabled. It runs for R=5 rounds with 20 passive opponents per round, matching the main evaluation protocol.

E.3 Bayesian Program Inference baseline

Bayesian Program Inference (BPI) tests whether iterative refinement can be replaced by passive best-of-N program inference, inspired by program-hypothesis scoring methods [Jha et al., 2026]. BPI uses raw behavioral traces, generates several independent executable hypotheses, and selects the candidate that best explains the observed target behavior under a distance-based posterior score.

Procedure. BPI runs one passive simulation round between the target and the fixed set of 20 opponents. It then spawns N=5 independent LLM editing sessions, each initialized with the same code template and given the game specification, policy API, and a bootstrap-resampled subset of the observed behavioral traces. Each session produces one candidate policy, with no shared memory or intermediate outputs across sessions. Candidates are validated using the same sandbox and API checks as the main learner. The highest-scoring candidate is selected as the final recovered policy.

Hypothesis diversity. To reduce collapse to near-duplicate samples, candidate generation uses two diversity mechanisms. First, each candidate receives a different bootstrap-resampled subset of the behavioral traces, so different opponent interactions can drive different hypotheses. Second, each candidate receives an approach-conditioning hint that encourages a distinct explanatory lens, such as minimal if–then rules, spatial or positional patterns, state-dependent mode switching, opponent-adaptive behavior, or resource/value optimisation. These mechanisms affect only generation; all candidates are scored on the same full observed trace set.

Distance-based posterior score. For candidate i, let $D_i = |\mathcal{O}|^{-1} \sum_{o \in \mathcal{O}} d(i, o)$ be its mean action distance to the target across the observed opponent set $\mathcal{O}$. We score candidates as

$$s_i = -\beta D_i - \alpha c_i,$$

where c_i is the number of non-trivial lines in the candidate code divided by 100. We use β=10 and α=0.01. This score corresponds to a distance-based likelihood with a weak complexity prior favouring shorter programs among candidates with similar behavioral fit. The submitted policy is the MAP candidate, i.e. $\arg\max_i s_i$.

F Per-Game Action Distances

This appendix gives the full definition of each per-game action distance $d_g : \mathcal{A}_g \times \mathcal{A}_g \to [0, 1]$ summarized in Table 7. All five distances satisfy the conventions established in Sec. 2: $d_g(a, a) = 0$, d_g is symmetric, d_g equals 1 when either argument is missing or invalid, and aggregation over action components is normalized so that scores remain comparable across states of differing dimensionality.

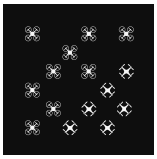
The recurring design choice across arenas is whether a mismatch is binary or graded. Where the action space is genuinely discrete and unordered (e.g., a single direction in BattleSnake), a binary distance is the only honest summary: "almost right" has no defensible metric meaning. Where the action carries quantitative content (raise sizes in poker, turn rates in RoboCode, per-cell move counts in Halite), a continuous distance is essential. The strategies we study encode behaviors in those quantities, and "right behavior, wrong magnitude" is a qualitatively different failure mode from "completely wrong." Collapsing the two onto a binary scale would flatten partial recovery into a single bit and make the metric blind to most of the recovery signal we care about.

Importantly, we always use one policy (the target) to roll out the actions against an opponent. For each state, we compare the action predicted by the agent's policy and compute the distance between the two. This ensures that we always compare actions that are directly comparable because they were taken in the exact same situation. Rolling out games using the target policy also guarantees that the learned policy is functionally similar to the target if it shows a small average distance, because it prevents cheating by only visiting states where the correct action is easy to infer. Conceptually, this is similar to the differences between forward and backward Kullback-Leibler Divergence.

F.1 BattleSnake [Chung et al., 2020]

In BattleSnake, the action space is $\mathcal{A}_{\text{BS}} = \{\text{UP}, \text{DOWN}, \text{LEFT}, \text{RIGHT}\}$: at each turn the snake selects exactly one of four cardinal directions. There is no natural ordering on this set: opposite directions

Table 7: **Action structure.** *Action* is what $\pi^*(s_t)$ returns; d_g is the per-state action distance in Eq. 1.

	BATTLESNAKE	HALITE	POKER	ROBOCODE	ROBOTRUMBLE
					
Action	1 of 4 directions	per-cell move list	fold, check, call, raise(r)	$\mathbb{R}^5$ (vel., turns, fire)	per-unit (type, dir.)
d_g	$\mathbf{1}[a_1 \neq a_2]$	fraction mismatched cells	log-normalized raise; fold binary	mean normalized component difference	0/0.5/1 per unit; averaged

are typically lethal (the snake collides with itself), but adjacent directions can have radically different outcomes depending on board state. We therefore use a binary distance,

$$d_{\text{BS}}(a_1, a_2) \;=\; \mathbf{1}[a_1 \neq a_2]. \tag{2}$$

A continuous relaxation (e.g., 0.5 for adjacent directions, 1 for opposite) was considered and rejected: the same direction can be optimal or fatal depending on the local configuration, so any geometric weighting would impose structure that the game does not have.

F.2 Halite [Truell and Spector, 2016]

In Halite, an action $a \in \mathcal{A}_{\text{Hal}}$ is a partial assignment from owned cells to moves: each ship or factory under the player's control receives one of a small set of commands (e.g., STAY, $\{$N, S, E, W$\}$, SPAWN, CONSTRUCT). Because the number of owned cells varies across game states, we cannot simply compare actions component-wise. Instead, we score each action over the union of cells assigned by either policy.

Let $\mathcal{C}(a) \subseteq \mathcal{C}_t$ denote the set of cells to which a assigns a command in state s_t, and let $\mathcal{C}_\cup = \mathcal{C}(a_1) \cup \mathcal{C}(a_2)$. For any cell $c \notin \mathcal{C}(a)$ we set $a[c] = \varnothing$ (no command), treated as a distinct value from any explicit command. The distance is the fraction of cells in the union on which the two policies disagree:

$$d_{\text{Hal}}(a_1, a_2) \;=\; \frac{1}{|\mathcal{C}_\cup|} \sum_{c \in \mathcal{C}_\cup} \mathbf{1}[a_1[c] \neq a_2[c]], \tag{3}$$

with $d_{\text{Hal}} = 0$ when both policies issue empty commands ($\mathcal{C}_\cup = \emptyset$). Averaging over $|\mathcal{C}_\cup|$ (rather than over $|\mathcal{C}_t|$) penalizes disagreements on commanded cells without rewarding either policy for declining to issue commands. Cells commanded by only one policy are scored as full mismatches, on the same principle: an omission in one policy is a substantive deviation from the other, not a free pass.

F.3 Poker (HuskyBench) [Kumar et al., 2025]

In Poker, an action is one of $\{$FOLD, CHECK, CALL, RAISE:$r\}$ with $r \in (0, 1]$ encoding the raise size as a fraction of the actor's chip stack at the decision point. The metric design separates two effects: the categorical decision to fold versus continue, and (for non-folds) the continuous magnitude of commitment.

Folds are handled categorically. We set $d_{\text{HB}}(\text{FOLD}, \text{FOLD}) = 0$ and $d_{\text{HB}}(\text{FOLD}, a) = 1$ for any non-fold a. This reflects the qualitative gap between folding (conceding the hand) and any continuation: folds and non-folds have no reasonable interpolation. We also identify CHECK and CALL for distance purposes, treating both as a zero-commitment RAISE:0. The two are mutually exclusive in any given state (the legal action depends on whether the actor faces a bet), so distinguishing them in the metric would only add noise.

For two non-fold actions, we compute a stack-aware log-commitment distance. Let S be the actor's chip stack at the decision point and let $\rho(a) \in [0, 1]$ be the raise ratio (0 for CHECK/CALL, r for

RAISE:r). Define the commitment function

$$c(\rho; S) \;=\; \frac{\log(1+\rho S)}{\log(1+S)}.$$

The distance between non-fold actions is then

$$d_{\text{HB}}(a_1, a_2) \;=\; \big|c(\rho(a_1); S) - c(\rho(a_2); S)\big|. \tag{4}$$

The log scale reflects two empirical facts about poker. First, the strategic distance between a 1%-of-stack raise and a 5%-of-stack raise is larger than between a 50%-of-stack raise and a 55%-of-stack raise, even though the absolute differences are equal. Second, the smallest legal raise should be meaningfully separated from a passive CHECK/CALL rather than collapsing to nearly-zero commitment. Normalizing by $\log(1+S)$ rather than a constant guarantees $d_{\text{HB}} \in [0, 1]$ in every state: an all-in is at distance 1 from a CHECK regardless of stack depth.

F.4 RoboCode [Hartness, 2004]

In RoboCode, an action is a five-dimensional continuous control vector $a \in \mathbb{R}^5$ with components (velocity, turn_body, turn_gun, turn_radar, fire_power). The components are physically heterogeneous: a difference of 1 in fire_power is qualitatively different from a difference of 1 in turn_radar. We therefore normalize each component by its valid range $R^{(i)}$ before averaging:

$$d_{\text{RC}}(a_1, a_2) \;=\; \frac{1}{5}\sum_{i=1}^{5} \min\left(\frac{|a_1^{(i)} - a_2^{(i)}|}{R^{(i)}},\; 1\right), \tag{5}$$

with component ranges taken from the RoboCode specification:

$$R^{\text{velocity}} = 16 \;\; ([-8, 8]), \qquad R^{\text{turn_body}} = 20 \;\; ([-10, 10]), \quad R^{\text{turn_gun}} = 40 \;\; ([-20, 20]),$$
$$R^{\text{turn_radar}} = 90 \;\; ([-45, 45]), \quad R^{\text{fire_power}} = 3 \;\; ([0, 3]).$$

The inner min clips per-component differences at 1, so that an out-of-range output (which the game engine would otherwise clip silently) cannot push the overall distance above 1. Equal weighting across components reflects that we have no principled basis for declaring one channel more strategically important than another; targets in our pool exercise all five.

F.5 RobotRumble [Outkine and Oxer, 2020]

In RobotRumble, a player controls multiple units simultaneously, and an action is a list of (unit_id, (TYPE, DIR)) pairs, where TYPE $\in$ {MOVE, ATTACK} and DIR is one of the eight unit-step directions. Disagreement on TYPE is a categorically larger error than disagreement on DIR alone: moving and attacking change different properties of the world, while a wrong direction within the same action type is a localized tactical mistake. We encode this asymmetry in a three-level per-unit distance.

Let $U = \mathcal{U}(a_1) \cup \mathcal{U}(a_2)$ be the set of unit identifiers commanded by either action. For a unit $u \in U$, define

$$d_u(a_1, a_2) \;=\; \begin{cases} 0 & \text{if } a_1[u] = a_2[u], \\ 0.5 & \text{if } \textsc{type}(a_1[u]) = \textsc{type}(a_2[u]) \text{ and both actions are present,} \\ 1 & \text{otherwise (different type, or one action missing for } u), \end{cases}$$

The full distance averages over the set U of units commanded in the state:

$$d_{\text{RR}}(a_1, a_2) \;=\; \frac{1}{|U|}\sum_{u \in U} d_u(a_1, a_2). \tag{6}$$

where $a[u]$ is taken to be missing if $u \notin \mathcal{U}(a)$. The action distance is the average over U:

$$d_{\text{RR}}(a_1, a_2) \;=\; \frac{1}{|U|}\sum_{u \in U} d_u(a_1, a_2), \tag{7}$$

with $d_{\text{RR}} = 0$ when both actions are empty ($U = \emptyset$). Failing to issue a command for a unit that the other policy commands is treated as a maximally different choice: in RobotRumble, idleness is itself a strategic decision and is rarely correct.

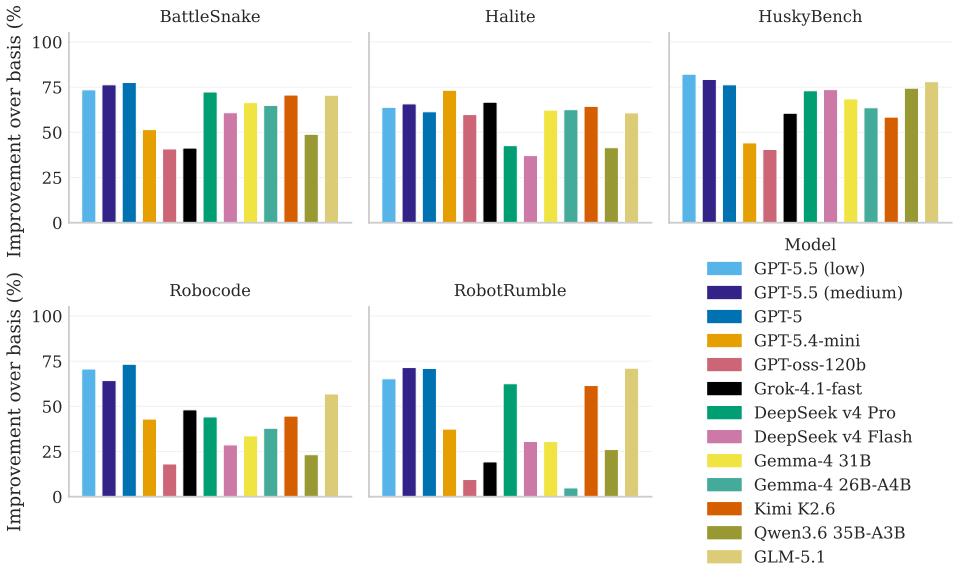


Figure 6: Per-game normalised recovery score for all models. Performance rankings vary across environments.

G Extended Results

G.1 Per-game strategy recovery

Figure 6 breaks down the aggregate recovery scores from Section 4.1 by game environment. Model rankings are not uniform across arenas, revealing environment-specific strengths. GPT-5.5-Codex-high achieves the highest overall recovery (83%) and leads or co-leads in every arena.

BattleSnake. GPT-5.5-Codex-high leads (81%), followed by GPT-5 and GPT-5.5-Codex-low (78%), GPT-5.5-medium (76%), and GPT-5.5-low (74%). Kimi K2.6 (71%) and DeepSeek v4 Pro (72%) form a strong second tier.

Halite. GPT-5.5-Codex-high dominates (82%), followed by GPT-5.4-mini (73%), GPT-5.5-Codex-low (68%), Gemma-4 26B-A4B and Grok-4.1-fast (67%). GPT-5 (61%) and DeepSeek v4 Flash (40%) trail.

Poker. This is the easiest arena overall. GPT-5.5-Codex-high leads (87%), followed by GPT-5.5-Codex-low (84%), GPT-5.5-low (82%), and GPT-5.5-medium (79%). DeepSeek v4 Flash and Qwen3.6 35B reach 74%.

RoboCode. GPT-5.5-Codex-high leads (74%), followed by GPT-5 (73%) and GPT-5.5-low (71%). GPT-5.5-medium and GPT-5.5-Codex-low reach 62–64%. GPT-oss-120b recovers only 18%.

RobotRumble. This arena shows the widest model spread. GPT-5.5-Codex-high dominates (89%), followed by GPT-5.5-Codex-low (82%). GPT-5, GPT-5.5-medium, and GLM-5.1 cluster around 71%. Gemma-4 26B-A4B trails at 5%.

G.2 Probing statistics

Figures 7 and 8 characterise how models use their probing budget across games.

Probe usage. Figure 7 shows the mean number of probes executed per tournament, broken down by game and model. Each model has a budget of 5 probes per round. Probe usage is highest in early rounds and decreases in later rounds, as models converge on a hypothesis and shift effort toward code

refinement rather than further experimentation. Some games (BattleSnake, Halite) tend to elicit more probing.

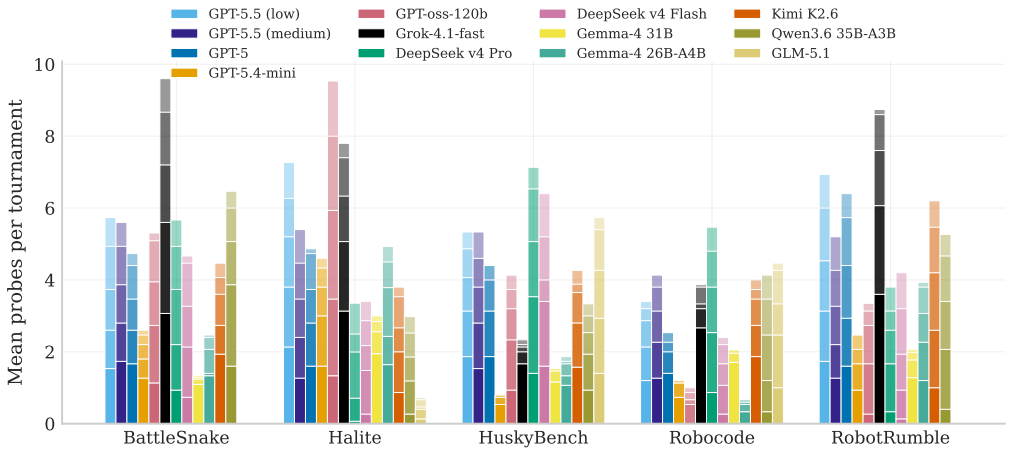


Figure 7: Mean number of probes executed per tournament, grouped by game. Each bar is segmented by round (darker = earlier rounds), showing the per-round breakdown within each model×game combination.

Probe failures. Figure 8 shows the mean number of inline probe failures (invalid submissions, sandbox errors, or timeouts) per tournament. Weaker models exhibit higher failure rates. This confirms that probing is a tool-use capability: models must not only decide *what* to probe but also produce valid, executable probe policies. Failed probes waste budget without yielding information, explaining why some models perform worse with probing enabled than without (Section 4.2).

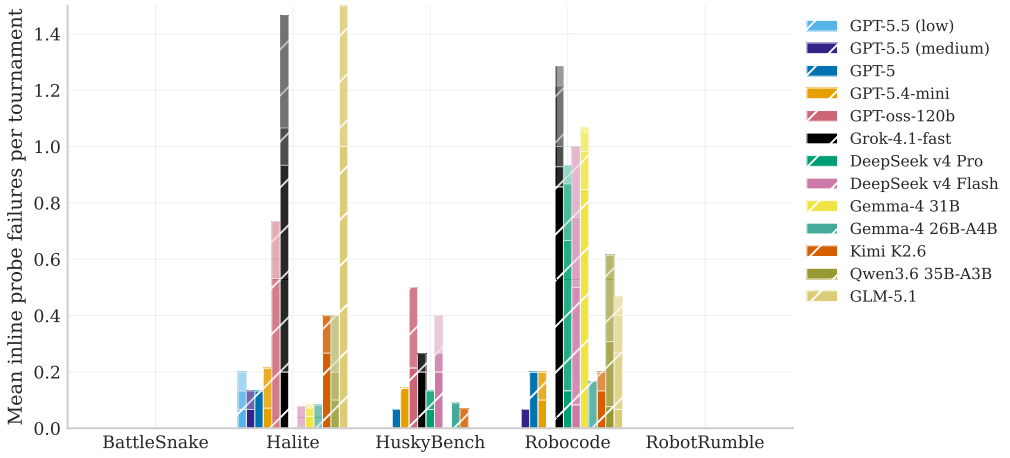


Figure 8: Mean inline probe failures per tournament. Each bar is segmented by round (darker = earlier rounds), showing when failures occur during refinement. Hatched bars indicate failed probe attempts (invalid code, sandbox errors, or timeouts). Failures concentrate in Halite, RoboCode, and RobotRumble; Grok-4.1-fast and GLM-5.1 show the highest rates in Halite, while DeepSeek v4 Pro and Grok-4.1-fast dominate RoboCode failures. GPT-5.5 variants (including Codex) achieve near-zero failure rates across all arenas.

G.3 Probe effectiveness

To quantify whether active probing improves strategy recovery, we compare best action distance D_{min} between Active (with probes) and Passive (no probes) conditions across the same four models. We define the *probe boost* as the reduction in distance afforded by probing: $\Delta = D_{min,\text{ passive}} - D_{min,\text{ active}}$, where positive values indicate that probing helped.

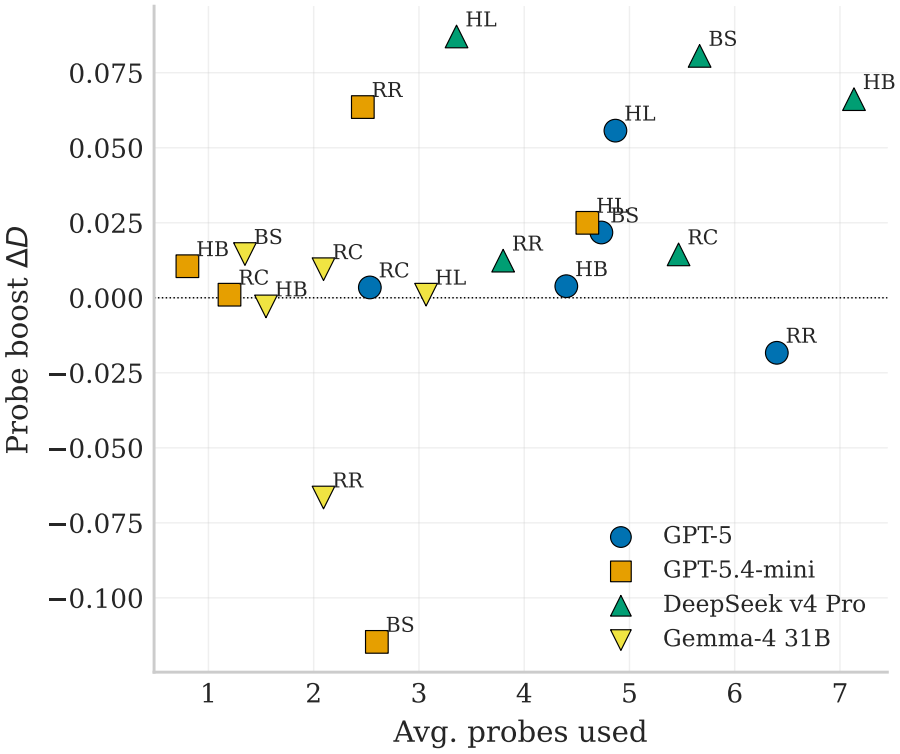


Figure 9: Probe boost vs. average probes used per tournament. Each point is one (model × game) pair. Points above the dashed line indicate probing reduced action distance (helped recovery); 16 of 20 points are positive. Marker shape encodes model; game abbreviations are annotated (BS=BattleSnake, HL=Halite, HB=Poker, RC=Robocode, RR=RobotRumble).

Figure 9 plots probe boost against the average number of probes consumed per tournament for each (model × game) pair. Across all 20 combinations, 16 out of 20 (80%) show a positive boost, with an overall mean improvement of $+0.013$ (σ=0.046). The effect is most pronounced for DeepSeek v4 Pro, which benefits from probing in all five games (mean boost $+0.052$), with the largest gains in Halite ($+0.087$) and Poker ($+0.066$). GPT-5 benefits in four of five games (mean $+0.013$), particularly in Halite ($+0.056$). GPT-5.4-mini and Gemma-4 31B show near-zero mean benefit, with individual games exhibiting both positive and negative effects. This indicates that for weaker models, probing can occasionally hurt when it consumes budget without yielding actionable information.

H From Inference to Action

The inverse-strategy pipeline recovers a target's decision procedure from behavioral traces and interaction with the target. A natural question is whether knowing the target's strategy is useful: if an agent has access to a model of its opponent's logic, can it leverage that model to win more often? This connects our pipeline to the broader question of theory of mind in competitive settings: whether understanding how an opponent makes decisions translates into a concrete strategic advantage.

We test this with a *PvP* (player-versus-player) experiment. A *challenger* agent receives the game rules and must write a counter-strategy over multiple rounds of iterative editing, receiving game outcomes after each round. The key manipulation is what the challenger knows about its opponent: nothing (blind), the code recovered by our inverse pipeline (recovered), or the opponent's actual source code (oracle). If opponent intelligence helps, the recovered and oracle conditions should outperform blind; if the recovery is faithful, the recovered condition should approach the oracle.

H.1 Protocol

Each experiment instance pits two players against each other over R=5 rounds of S=30 simulations each.

Player 1 (target). Plays its original strategy code, held fixed throughout the tournament. The target is not an LLM agent; it simply executes the source code that was the subject of the inverse-strategy evaluation.

Player 2 (challenger). An LLM-powered coding agent (Section 3) that iteratively edits a counter-strategy. After each round, the challenger observes the game results and may revise its code. The challenger sees the standard game description prompt for the relevant game; the only variable across conditions is the presence and source of opponent intelligence.

We compare three levels of opponent information:

(a) **Blind.** The challenger receives only the game rules. No opponent code is provided. This is the control baseline: any wins come from the challenger's intrinsic coding ability and its iterative adaptation to game outcomes.
(b) **Recovered.** The challenger receives the code recovered by our inverse-strategy pipeline (the best-round submission from the main inverse strategy evaluation). The code is injected into the game prompt as an "opponent intelligence" section.
(c) **Oracle.** The challenger receives the target's actual source code. This serves as a ceiling: if even perfect knowledge does not help, opponent modelling is uninformative for the game in question.

In the recovered and oracle conditions, the opponent code is appended to the game description with the following prompt, deliberately worded to discourage brittle hard-coded exploits that would succeed against the exact target code but fail against minor perturbations:

Opponent Intelligence Prompt

You have obtained your opponent's strategy code: `[code]`. Use this intelligence to inform your strategy. A well-designed bot that accounts for the opponent's tendencies will outperform a fragile exploit. Focus on writing robust, well-tested code first, then incorporate any insights from the opponent's logic as refinements.

H.2 Games, Targets, and Models

We evaluate on four games from our benchmark: BATTLESNAKE, HALITE, ROBOCODE (Java oracle / Python recovered), and ROBOTRUMBLE. For each game, we select the top-5 targets by behavioral distance from the main inverse strategy (Section 4), i.e., the five targets whose strategies were most faithfully recovered. Selecting well-recovered targets ensures that the recovered condition contains a meaningful approximation of the target's logic; poorly recovered targets would make the comparison uninformative.

Five challenger LLMs span a range of coding ability: GPT-5, GPT-5-mini, GPT-oss-120b, DeepSeek-v3.2, and Grok-4.1-fast. Each model is used as the challenger in all three conditions against all 20 targets (5 per game $\times$ 4 games). All models are evaluated with three independent tournament seeds. This yields $4 \times 3 \times 60 = 720$ tournament runs and approximately 81,000 simulations; all reported win rates are averaged across seeds. Using models of varying strength allows us to test whether the value of opponent intelligence depends on the challenger's baseline capability.

H.3 Metric

We measure the challenger's *win rate* per round:

$$\mathrm{WR}_r = \frac{s_{\text{challenger},r}}{s_{\text{challenger},r} + s_{\text{target},r}} \times 100\%, \tag{8}$$

where $s_{\cdot,r}$ is the cumulative simulation score in round r. The primary summary statistic is the mean win rate over rounds 1–5. We define the *win rate gain* as $\mathrm{WR}_{\text{condition}} - \mathrm{WR}_{\text{blind}}$, which isolates the contribution of opponent information from the challenger's baseline performance. Specifically, *oracle lift* $= \mathrm{WR}_{\text{oracle}} - \mathrm{WR}_{\text{blind}}$ and *recovered lift* $= \mathrm{WR}_{\text{recovered}} - \mathrm{WR}_{\text{blind}}$.

H.4 Hypotheses

We test three hypotheses:

H1: Opponent information > Blind. Knowing the opponent's strategy, whether the exact source code (oracle) or the code recovered by our inverse pipeline (recovered), helps the challenger win.

H2: Advantage decays over rounds. The initial boost from opponent knowledge shrinks as the blind challenger iteratively improves from game feedback alone. A strong decay would suggest that opponent intelligence provides a warm-start advantage rather than a persistent structural edge.

H3: Weaker models benefit more. Models with lower blind baselines derive larger advantages from opponent information. If the blind baseline is already high, there is less room for improvement; if it is low, opponent intelligence fills the gap.

H.5 H1: Opponent Information Improves Win Rate

Figure 4 (left) in the main paper shows that across all five models, oracle outperforms recovered, which in turn outperforms blind. The gap between recovered and oracle reveals that imperfect recovery captures only a fraction of the full intelligence value; nevertheless, the recovered condition consistently outperforms blind, establishing that the inverse-strategy pipeline produces actionable intelligence. The survival-CDF view (Figure 10) reinforces this: for weaker models, oracle and recovered curves shift visibly right of blind, meaning these challengers beat more targets, and beat them more decisively, when armed with opponent information.

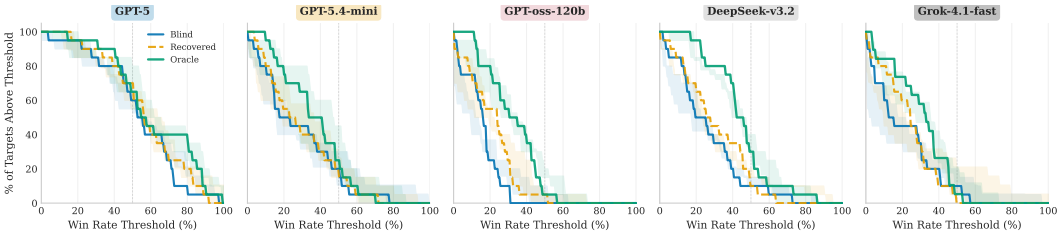


Figure 10: Fraction of targets where the challenger achieves at least a given win rate, by condition. For GPT-5, the three curves nearly overlap, as blind performance is already strong. For weaker models, oracle and recovered curves shift visibly right of blind.

H.6 H2: The Intel Advantage Decays for Strong Models but Persists for Weak Ones

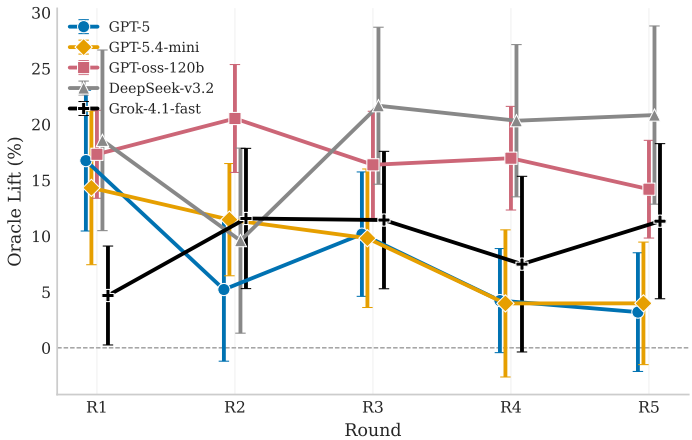


Figure 11: Oracle lift (oracle minus blind win rate) across rounds, with ± 1 standard error bars. GPT-5 and GPT-5-mini start with large lifts that decay toward zero as their blind baselines catch up through iterative play. Weaker models show persistent lift because their blind performance barely improves.

Figure 11 traces the oracle lift from Round 1 through Round 5. Two regimes emerge. GPT-5 and GPT-5.4-mini start with large lifts that shrink as their blind baselines catch up through iterative play. The picture reverses for weaker models: DeepSeek-v3.2, and Grok-4.1-fast show persistent oracle lift because their blind performance barely improves across rounds.

H.7 H3: Weaker Models Benefit More from Intel

Figure 4 (middle, right) from the main paper makes the relationship between baseline strength and win rate gain benefit explicit. The negative slopes are significant ($p < 0.05$) for four of five models on both oracle lift and recovered lift (Table 8), confirming that the lower the blind baseline, the larger the lift from opponent intelligence.

Table 8: Pearson correlation between blind win rate and win rate gain per model. Negative r confirms that weaker blind baselines benefit more from opponent intelligence.

	Oracle lift		Recovered lift	
Model	r	p	r	p
GPT-5	-0.41	0.069	-0.53	0.017
GPT-5.4-mini	-0.74	<0.001	-0.55	0.012
GPT-oss-120b	-0.48	0.033	-0.43	0.059
DeepSeek-v3.2	-0.81	<0.001	-0.53	0.016
Grok-4.1-fast	-0.66	0.002	-0.66	0.002

Crucially, the pattern holds for recovered lift (Table 8 right panel) as well as oracle lift, confirming that the recovered code, not just the ground-truth source, carries enough actionable signal to benefit weaker challengers.

H.8 Per-Model Win-Rate Trajectories

Figure 12 shows per-game challenger win-rate trajectories for each of the five models across all four games and three intel conditions. Each row corresponds to one challenger model, ordered from strongest (GPT-5) to weakest (Grok-4.1-fast); each column within a row is one game.

I Context History Management

The inverse-strategy agent operates over multi-round interaction, where each round appends environment observations (evaluation mismatches, probe traces), file reads, and model responses to the conversation history. Without management, context grows linearly in the number of rounds, risking context-window overflow and inflating inference cost.

We address this with two complementary mechanisms that control context size while preserving the information the agent needs at each editing step: *history compaction*, which truncates and summarises older observations, and a *per-summary cap* that hard-limits any remaining long summaries.

I.1 History Compaction

History compaction is the primary context-reduction mechanism. It operates at two levels: an observation-level hard cap applied at *write* time, and a summary-based compaction applied at *query* time on a read-only copy of the conversation.

Observation truncation. Before an observation enters the history, each tool output is capped at L_{out} characters. Outputs exceeding the limit retain the first and last $L_{\text{out}}/2$ characters; the middle is replaced with an omission marker. After truncation every observation satisfies:

$$|o_i| \;\leq\; L_{\text{out}}. \tag{9}$$

Summary-based compaction. Let k denote the number of recent observations preserved verbatim. The history is partitioned into a *recent* window (the last k user observations) and a *past* set (all earlier

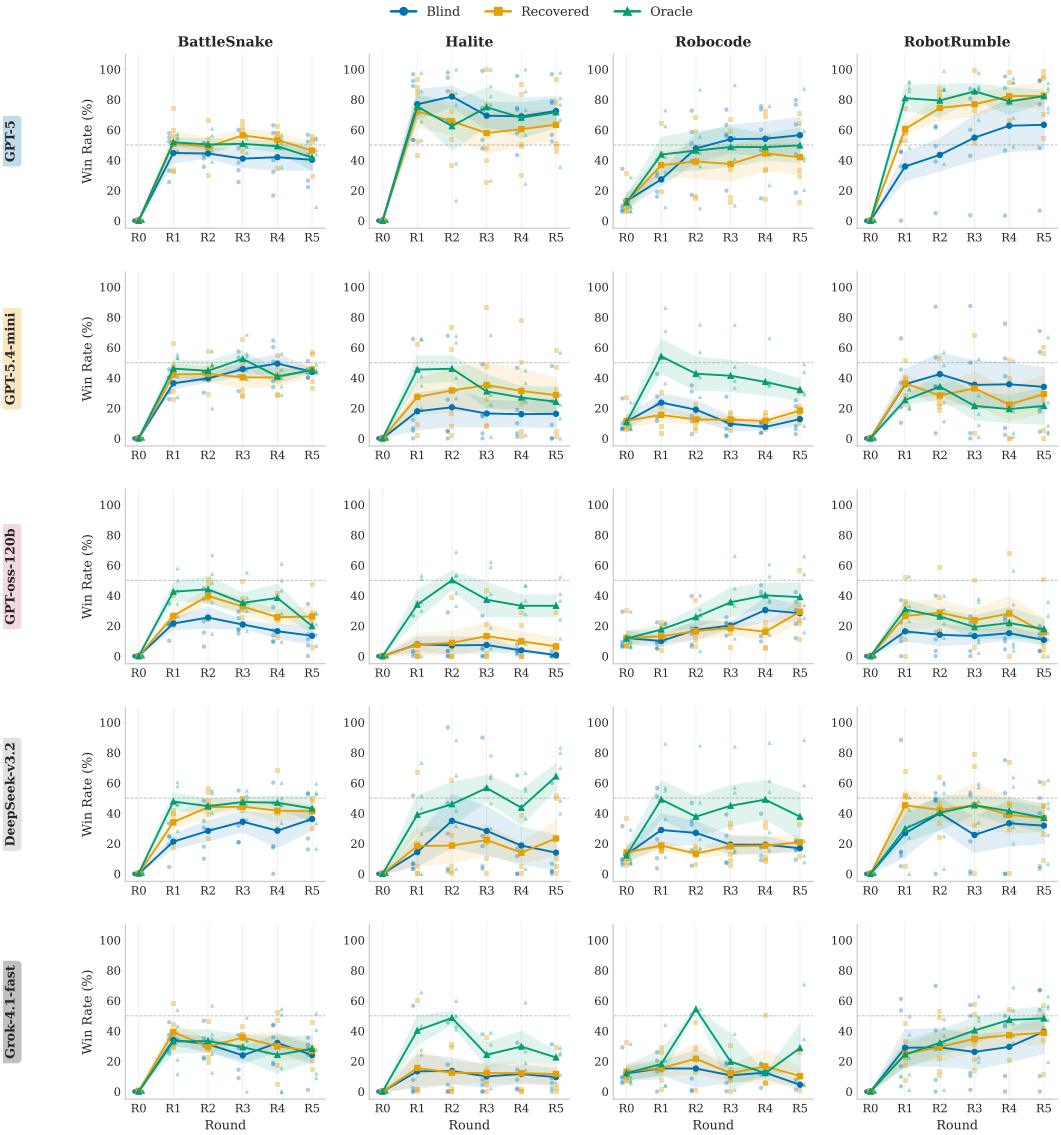


Figure 12: Per-model PvP win-rate trajectories across games and intel conditions (blind, recovered, oracle), ordered from strongest (top) to weakest (bottom). Each row is one challenger model; each column is one game. Strong models show overlapping conditions or decaying oracle lift; weaker models exhibit persistent or growing oracle lift.

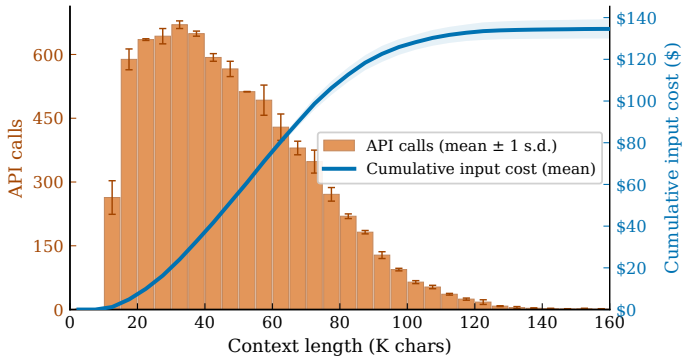


Figure 13: Context-length distribution and cumulative input cost across two independent GPT-5 runs over all targets without compaction (7 712 and 8 052 API calls; mean total input cost \$135). Bars show the mean call count per 5 K-char bin (error bars: ± 1 s.d. across runs); the blue curve shows cumulative input cost. Calls above 60 K chars account for roughly half the input bill, motivating history compaction.

observations). Each past observation o_i is replaced by a structured summary:

$$o_i \longrightarrow \tilde{o}_i = \textsc{Summarize}(o_i). \tag{10}$$

The summarizer is rule-based and type-aware: it recognises probe feedback, game traces, source-code reads, and truncated outputs, producing a one-line pointer for each (e.g., `[Compacted probe 3/5: 42 pairs, 2 sims]` which points to compaction of 42 state-action pairs in 2 simulations).

Because the most recent k observations carry the information the agent is actively working with—the latest mismatch analysis, the last code edit, the most recent probe result—older observations are largely redundant. Retaining short summaries rather than discarding them entirely prevents the agent from re-exploring strategies it has already tried.

I.2 Per-Summary Cap

After compaction, each summarized past observation is further hard-truncated to L_{past} characters:

$$\tilde{o}_i \longrightarrow \hat{o}_i = \textsc{Truncate}(\tilde{o}_i,\ L_{\text{past}}), \qquad |\hat{o}_i| \le L_{\text{past}}. \tag{11}$$

This serves as a safety net for cases where the summarizer's output is still large—for example, when a single observation contained interleaved outputs or the original text was so long that even the summary preview is substantial. Alternatively, it can be used on its own as a lightweight compaction strategy when full summarization is not needed.

I.3 History Compaction Ablation

To isolate the effect of history compaction, we run a controlled ablation on BATTLESNAKE and HALITE with GPT-5, comparing three conditions: no compaction ($k{=}\varnothing$), aggressive compaction ($k{=}1$), and moderate compaction ($k{=}3$). Each condition uses 3 targets $\times$ 2 seeds $=$ 6 runs over 5 rounds.

Figure 13 shows why compaction matters: without it, context lengths reach 150 K chars with a heavy right tail, and calls above 60 K chars account for roughly half the input cost. Figure 15 shows the effect of compaction on the context-length distribution: both $k{=}3$ and $k{=}1$ shift the mass leftward, cutting median context from 55 K to 30 K and 27 K chars respectively.

Ablation cost. Figure 14 shows the per-step cost trajectory (top row) and cumulative cost (bottom row) for both games. Without compaction, per-step cost rises steadily as the context grows; both compaction settings keep it flat. On BATTLESNAKE, $k{=}3$ reduces total cost by 14% (\$2.87 vs \$3.35) and $k{=}1$ by 11% (\$2.99); on HALITE, savings reach 20–22% (\$2.89 and \$2.83 vs \$3.61). Note that $k{=}1$ is *more* expensive than $k{=}3$ on BATTLESNAKE despite a lower per-step cost: with less context retained, the agent takes more steps per round to reach the same editing decisions, and the

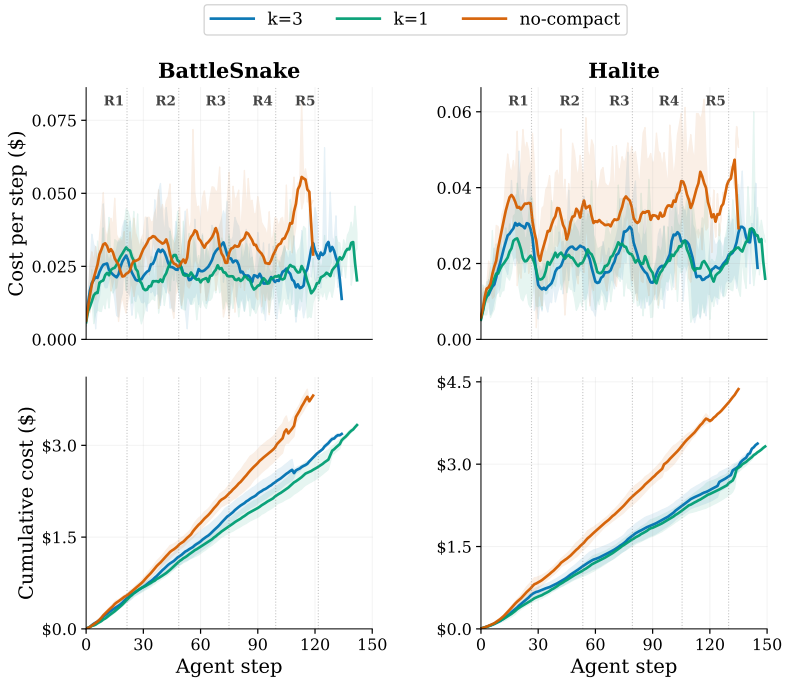


Figure 14: History compaction ablation (GPT-5, 3 targets × 2 seeds per condition). **Top:** per-step API cost; without compaction (vermillion), cost per call rises as context grows, while both k=1 and k=3 keep it flat. **Bottom:** cumulative cost over agent steps.

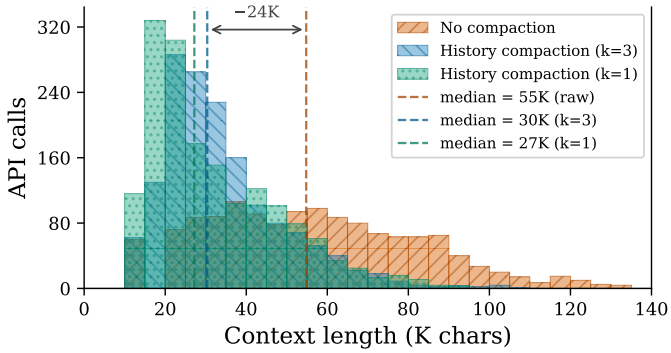


Figure 15: Effect of history compaction on context length (same ablation as Figure 14). Compaction shifts the distribution leftward: median context drops from 55 K chars (no compaction) to 30 K (k=3) and 27 K (k=1).

additional steps offset the per-call savings. This effect is more pronounced on BATTLESNAKE, where the agent's editing loop is more sensitive to the amount of recent context available.

Strategy recovery. Table 9 summarises cost and behavioral distance across conditions. Both compaction settings reduce total cost by 11–22% while producing comparable or better final-round distances. The Δ columns show the change relative to the no-compact baseline: negative Δ_{cost} indicates savings; negative Δ_d indicates equal or improved strategy recovery. All distance differences fall well within one standard deviation of the baseline, confirming that summarizing old observations does not degrade performance.

Table 9: History compaction ablation summary (GPT-5, 3 targets × 2 seeds per condition). Δ columns show change relative to no-compact. For cost, ↓ is better; for distance, ↓ is better.

		Total cost ($)		R5 distance		Best distance	
Game	Condition	mean±std	Δ%	mean±std	Δ	mean±std	Δ
BSNAKE	k=3	2.87±0.38	−14%	.371±.319	+.051	.272±.237	−.008
	k=1	2.99±0.37	−11%	.456±.258	+.136	.275±.239	−.005
	no-compact	3.35±0.48	—	.320±.250	—	.280±.235	—
HALITE	k=3	2.89±0.39	−20%	.549±.152	−.041	.384±.181	−.086
	k=1	2.83±0.39	−22%	.421±.263	−.169	.380±.204	−.090
	no-compact	3.61±0.63	—	.590±.252	—	.470±.180	—

J Noise-decomposition Methodology

J.1 Setup

We run all 15 targets per arena across the five REVENGEBENCH arenas through Gemma 4 31B with five distinct values of the `tournament.seed` config field (42, 100, 200, 300, 400). The target's behavior is fixed across runs (`fixed_target: true`), and the 20 opponents the target plays in each round k are picked once by `random.Random(strategy_pool.seed + target_idx)` with `strategy_pool.seed` held at 42 across the five runs, so **the opponent set per (target, round) is identical across all five runs**. The differences between runs come from (a) game-engine internal RNG (Halite map generation, BattleSnake food spawns), (b) Python's global `random.shuffle` in the arena code, and (c) LLM API non-determinism (no `seed` is passed to OpenRouter). Each (target, run) tuple produces six rounds (round 0 = basis evaluation of the starter code; rounds 1–5 = the agent's edits). Within each round the learner's code is evaluated against the 20 opponent-game traces to produce an action distance per game (`per_simulation`) and a round-level mean (`distance_history[r]`). 75 tournaments × 6 rounds × 20 simulations = 9,000 sim-level evaluations per arena cell.

J.2 Variance components

Let d_{tsri} be the action distance of target t's code at round r against opponent i in run s. We report standard deviations σ (`ddof` $= 1$) rather than variances so units stay in the metric's native scale (action distance $\in [0, 1]$).

- **Within-run (σ_{sim}).** $\sigma_{\text{sim}}^{(t,s,r)} = \text{sd}_i\, d_{tsri}$, then averaged over (t, s) at the requested round. Holds the learner's code, the target's behavior, and the run fixed; only the opponent identity and per-sim RNG vary. Captures the irreducible spread of one tournament round across its opponent set.

- **Across-run (σ_{seed}).** $\sigma_{\text{seed}}^{(t,r)} = \text{sd}_s\, \bar{d}_{tsr}$, averaged over targets. Holds the target and the opponent set fixed; varies only the run. Captures the residual process-level non-determinism after averaging over the 20 opponents.

- **Across-target (σ_{target}).** $\sigma_{\text{target}}^{(r)} = \text{sd}_t\, \bar{d}_{t\cdot r}$ where the inner mean is over runs. The signal: how much targets actually differ in difficulty.

We report all three at round 5 (the final round) in Table 10; Figure 16 shows them as grouped bars per round, not stacked, since standard deviations do not add. This split parallels the across-task / within-task decomposition recently formalised for one-shot QA evaluation [Wang, 2026].

Two arenas (BattleSnake, Poker) have σ_{target} larger than both noise sources and ICC > 0.8; their headline metric is signal-dominated. Three arenas (Halite, RoboCode, RobotRumble) have σ_{seed} comparable to or larger than σ_{target} and ICC ≤ 0.41; scores there move substantially run-to-run, so single-run point estimates should be replaced with run-aggregated means or paired-test framings when comparing models.

Table 10: Within-Gemma noise decomposition at round 5 across the five arenas (15 targets, 5 runs each). σ_{sim}, σ_{seed}, and σ_{target} are standard deviations of the round-5 mean action distance with the inner aggregation specified in §J.2. ICC and $\bar{\rho}$ are computed on the (target $\times$ run) matrix of normalised recovery score (§J.3). Bootstrap CI uses 2,000 hierarchical resamples.

Arena	σ_{sim}	σ_{seed}	σ_{target}	ICC	$\bar{\rho}$	recovery (95% CI)
BattleSnake	0.096	0.098	0.150	0.81	0.88	66.4% [54.7, 78.0]
Halite	0.115	0.206	0.104	0.07	0.14	63.1% [55.6, 69.3]
Poker	0.089	0.028	0.059	0.85	0.84	68.5% [55.5, 80.4]
RoboCode	0.067	0.038	0.030	0.41	0.32	33.7% [27.9, 39.8]
RobotRumble	0.116	0.179	0.118	0.19	0.23	30.5% [22.3, 38.6]

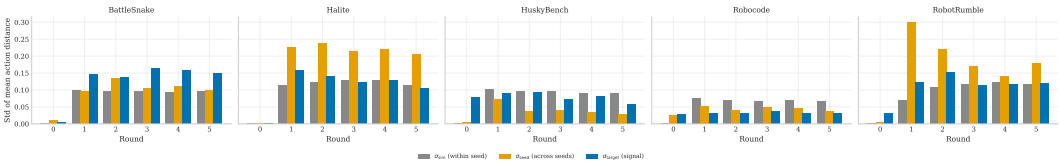


Figure 16: Within-Gemma variance components at each round, grouped side-by-side (not stacked, since standard deviations do not add). σ_{sim} is the within-run, across-opponent spread; σ_{seed} is the across-run process-level noise at fixed target and opponent set; σ_{target} is the across-target signal. Round 0 has near-zero σ_{seed} and σ_{target} because the starter code is identical across runs and basis difficulty is uniform within an arena.

J.3 Headline statistics

We compute consistency ICC (Shrout–Fleiss two-way random-effects, single-rater) on the (target $\times$ run) matrix of bri_{ts} values, where

$$\text{bri}_{ts} = 100 \cdot \frac{d_{ts0} - \min_{r \geq 1} d_{tsr}}{d_{ts0}}$$

is the normalised recovery score over the basis distance. Closed form:

$$\text{ICC} = \frac{\text{MS}_{BT} - \text{MS}_E}{\text{MS}_{BT} + (k-1)\,\text{MS}_E},$$

where MS_{BT} is the between-target mean square and MS_E is the residual mean square (run main effect excluded; this is the consistency form rather than absolute agreement). ICC near 1 means signal dominates noise; ICC near 0 means the opposite. We additionally report the mean pairwise Spearman ρ between per-target rankings across runs (full 5×5 matrix per arena in Figure 17). The 95% confidence interval on the mean recovery score is from a hierarchical bootstrap (2,000 resamples: targets with replacement, then runs within each chosen target).

J.4 Cross-model validity

The within-Gemma analysis above describes how reliably one model is scored. To test whether the benchmark itself assigns consistent difficulties to its targets across evaluators, we apply the same Spearman ρ and ICC machinery across our 12-model lineup at one run per model. Gemma 4 31B contributes its 5-run mean as a stabilised reference; the other 11 evaluators (DeepSeek v4 Pro / Flash, GLM-5.1, GPT-5.5, GPT-5, GPT-5.4-mini, GPT-oss-120b, Gemma 4 26B-A4B, Grok-4.1-fast, Kimi K2.6, Qwen3.6 35B-A3B) contribute their `seed=42` run.

The ordering tracks the within-Gemma reliability picture. Poker is the only arena with strong cross-evaluator agreement (ICC 0.58, $\bar{\rho} = 0.61$), and BattleSnake is moderate (0.32, 0.51); the original within-Gemma claim that BattleSnake rankings are run-stable does not extend cleanly to evaluator-stability. Halite, RoboCode, and RobotRumble disagree across evaluators on per-target difficulty (ICC ≤ 0.20), tracking the same high run-to-run noise visible in the within-Gemma sweep rather than reflecting evaluator-specific behavior.

Table 11: Cross-model agreement on per-target normalised recovery score at round 5. $N_{\text{models}} = 12$ (Gemma 4 31B at its 5-run mean; the other 11 at their single seed=42 run). $\text{ICC}_{\text{cross}}$ is consistency ICC on the (target $\times$ model) matrix; $\bar{\rho}_{\text{cross}}$ is the mean pairwise Spearman ρ between models on per-target rankings.

Arena	N_{models}	N_{targets}	$\text{ICC}_{\text{cross}}$	$\bar{\rho}_{\text{cross}}$
BattleSnake	12	15	0.32	0.51
Halite	12	15	0.20	0.33
Poker	12	15	0.58	0.61
RoboCode	12	15	0.07	0.09
RobotRumble	12	15	0.11	0.14

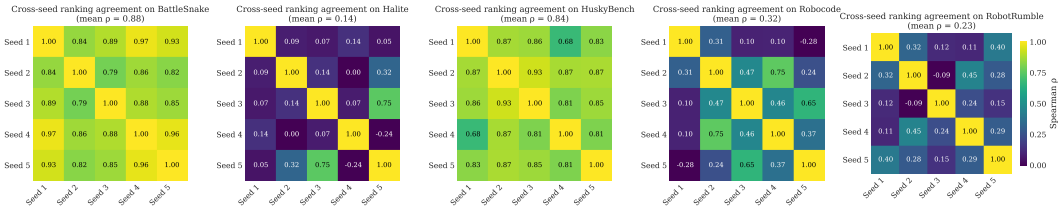


Figure 17: Within-Gemma cross-run Spearman ρ between per-target normalised-recovery rankings, one panel per arena. Mean ρ in each panel title. BattleSnake and Poker show uniformly high agreement; Halite, RoboCode, and RobotRumble show large run-to-run rank flips.

σ_{sim} is an arena-level property. Figure 18 shows the σ_{sim} distribution at round 5 per (model, arena). Within each arena the 12 per-model boxplots overlap heavily, confirming that the irreducible opponent-draw noise depends on the arena's mechanics, not on the evaluator. The within-Gemma σ_{sim} values reported above therefore generalise to the other 11 evaluators.

J.5 Limitations

Five runs is a small sample. Recent reproducibility studies of small reasoning benchmarks argue for $K \approx 30$ runs as a stabilisation point for per-question pass-rate estimates [Hochlehnert et al., 2025]; our 5-run budget is well below that, so we treat per-cell tests as descriptive and lean on the aggregate bootstrap and ICC for the headline reliability claims.

Only one model has multi-run coverage. The 11 additional evaluators contribute one run each, so the within-arena σ_{seed} reported here is Gemma-specific. The σ_{sim} consistency check (Figure 18) is the closest evidence that the rest of the noise-structure characterisation generalises.

textbfOutlier sensitivity. Some runs fail catastrophically: the agent's context window overruns, its code crashes the engine, or its tournament hits the step / cost budget mid-way. We keep these in the analysis because a model that destroys its own context or submits broken code is exhibiting real (negative) capability. On the Gemma 4 31B sweep, RoboCode has 14 / 75 runs ($\approx 19\%$) truncated, Halite has 5 runs with round-5 distance ≥ 0.85, Poker has 4 truncated runs, and BattleSnake and RobotRumble have none. The headline numbers therefore mix two distinct sources: noise on successful runs, and a binary "this run failed" signal. Table 10 should be read as upper bounds on the noise share that survives outlier removal, particularly for RoboCode (truncations) and Halite (high-distance failures).

K Cost Analysis

All models are evaluated on the same pool of 75 target strategies (15 per game $\times$ 5 games). Figures 20 and 21 visualise the cost and token-usage breakdowns.

Inference cost. Figure 20 breaks down the total API cost per model into prompt (input), reasoning, and answer (output) token charges. GPT-5.5 (medium) is the most expensive configuration at \$940 for the full pool, driven by its high reasoning-effort setting and per-token pricing. GPT-5.5 (low)

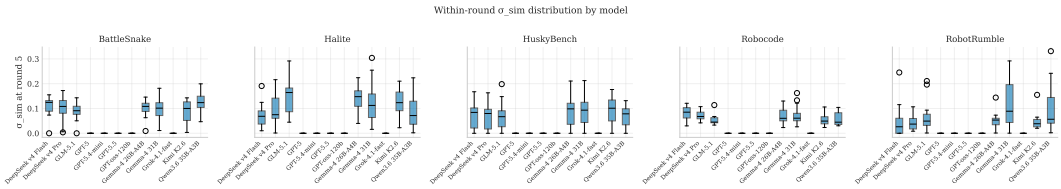


Figure 18: Distribution of σ_{sim} at round 5, one boxplot per evaluator, faceted by arena. The within-arena overlap across the 12 evaluators supports treating σ_{sim} as an arena-level property rather than a model-specific quantity.

costs \$554, while the Codex variants (low and high) cost only \$5 and \$0 respectively due to Codex's bundled pricing. GPT-5.4-mini remains highly cost-efficient at \$43. DeepSeek v4 Pro and Kimi K2.6 occupy a middle ground at \$116 and \$193 respectively.

Token usage. Figure 21 shows raw token consumption summed over all 75 runs. DeepSeek v4 Pro consumes the most tokens overall ($\sim$293 M), reflecting both long prompts from its multi-turn context window and substantial reasoning overhead. GPT-5.5 (Codex high) and GPT-5.5 (Codex low) use the fewest tokens ($\sim$38–72 M), benefiting from Codex's more efficient scaffold. GPT-5.5 (medium) uses $\sim$194 M tokens despite being the most expensive, indicating that cost is driven by per-token pricing rather than volume alone. GPT-5.4-mini remains token-frugal at $\sim$72 M.

Figure 19: Per-target difficulty agreement: Gemma 4 31B's 5-run mean recovery score (x-axis) versus each other evaluator's single-run recovery (y-axis), one row per other model and one column per arena. Dashed line is the 1:1 reference; Spearman ρ is in each panel title. On Poker most $\rho \geq 0.6$; on RoboCode and RobotRumble many ρ are below 0.3 with sign flips, consistent with the low cross-model ICC in Table 11.

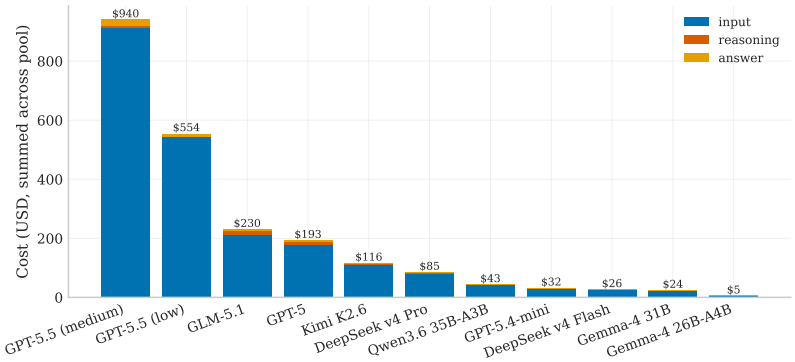


Figure 20: Total API cost breakdown per model across all 75 runs. Stacked segments show prompt, reasoning, and answer token charges.

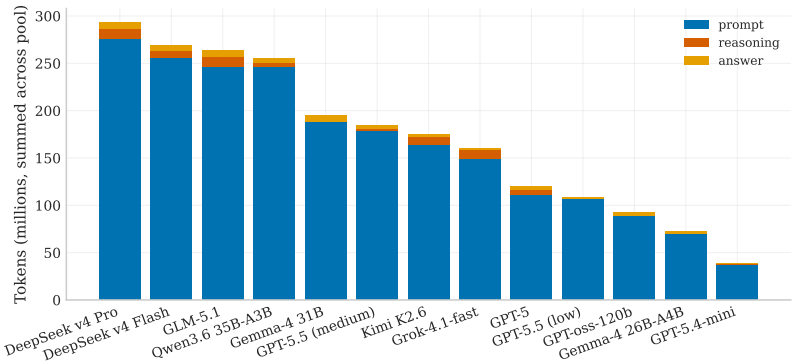


Figure 21: Total token usage per model (millions). Stacked segments show prompt, reasoning, and answer tokens. DeepSeek v4 Pro is the most token-intensive model; GPT-5.4-mini the most frugal.