OPINE-WORLD: PROGRAMMATIC WORLD MODELING WITH ONTOLOGY-ERROR-PRIORITIZED INTERACTIVE EXPLORATION FOR ARC-AGI-3

Anonymous authors
Paper under double-blind review

ABSTRACT

Learning a useful world model from minimal interaction is central to building agents that adapt to unfamiliar tasks. Programmatic world modeling approaches such as WorldCoder quickly learn transition models when given pre-supplied symbolic representations; however, such approaches are insufficient when the symbolic representation and goal are unknown and continually changing, and are intractable under large action spaces due to rigid planning heuristics. Extending the programmatic world modeling approach, OPINE-World additionally maintains an abstracted representation in which provisional objects, incomplete causal explanations, and unresolved interactions can exist in a Bayesian, exploration-centric hypothesis space. This enables learning an unknown, dynamically evolving state, transition, and goal. Exploration and program revision share this evolving abstraction, allowing new evidence to revise both the hypothesized mechanics (transition and reward behaviors) and their implementation. ARC-AGI-3 presents this open-world problem as a general-intelligence learning benchmark. Through our OPINE-World model learning paradigm, we improve the Relative Human Action Efficiency score of Claude Opus 4.8 high from 1.5% to 78.40% on the ARC-AGI-3 benchmark.

1 INTRODUCTION

World model learning is essential for active reasoning and decision-making, but latent approaches rely on more training data than is feasible online (Sutton, 1991; Ha & Schmidhuber, 2018; Hafner et al., 2025). Programmatic models such as WorldCoder learn executable transition and reward models over a pre-supplied symbolic representation (Tang et al., 2024; Piriyakulkij et al., 2025) and reuse dynamics efficiently (Diuk et al., 2008; Battaglia et al., 2018; Kipf et al., 2020), but rely on a rare assumption of accurate scene decomposition. Such approaches fail when the symbolic state and goal are simultaneously unknown and continually changing. Extending the world modeling approach of WorldCoder, our idea is to maintain two levels of representation: a qualitative, object-oriented natural language (NL) proto-representation of the world, and an executable world model in Python. In programmatic form, internal simulation and reward-directed planning to goal search become possible, whereas the NL proto-representation allows qualitative abstractions, enabling the management of hypothesis spaces and heuristics that require further exploration before conversion to programmatic form. Exploration and program revision share this evolving abstraction, allowing new evidence to revise both the hypothesized mechanics and their implementation in tandem at test time. The shared object abstraction additionally supports lifted Bayesian tracking of epistemic uncertainty over object–action effects. These beliefs identify sparsely tested or correlative effects, allowing for targeted exploration to learn causal relationships.

Using Claude Opus 4.8 high, OPINE-World clears 20 of 25 ARC-AGI-3 games, improving the Relative Human Action Efficiency (RHAE) score of the base model from 1.5% to 78.40% (Section 5.1)[1].

[1]As of July 25, 2026, the only system that performed better on the Community leaderboard was NVIDIA's NOOA (Furgale et al., 2026), on GPT-5.6-sol xhigh, reported 85.1% versus our 78.40%. There exists a high RHAE disparity of 4.6x between Opus 4.8 high and GPT 5.6 Sol xhigh base models, motivating our exclusion of significantly stronger base models from the comparison in Table 1.

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

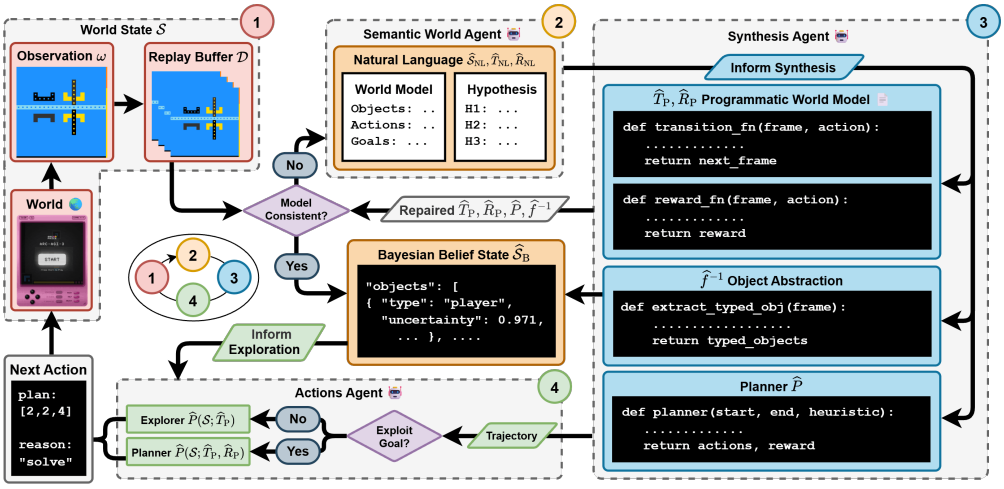


Figure 1: **OPINE-World.** The world produces an observation ω, stored in replay buffer $\mathcal{D}$ and checked against the programmatic model. Model inconsistency repairs $(\widehat{T}_{\text{P}}, \widehat{R}_{\text{P}}, \widehat{P}, \widehat{f}^{-1})$. A consistent model routes a trajectory to the action agent, which uses the planner $\widehat{P}(\mathcal{S}; \widehat{T}_{\text{P}}, \widehat{R}_{\text{P}})$ to exploit, or the explorer $\widehat{P}(\mathcal{S}; \widehat{T}_{\text{P}})$ to resolve uncertainty with the NL and Bayesian belief state $\widehat{\mathcal{S}}_{\text{B}}$.

Removal of the Bayesian epistemic layer $\widehat{\mathcal{S}}_{\text{B}}$ for exploration reduces RHAE by 20.24 points. We show that retaining our object-oriented proto-representation significantly reduces actions, synthesis rounds, and tokens during early online learning compared to a purely programmatic approach (Section 5.4), and enforces program structure that increases generalization and robustness (Section 5.3).

Contributions.

- To avoid inefficient, low-diversity exploration patterns, we introduce an object-oriented Bayesian belief state $\widehat{\mathcal{S}}_{\text{B}}$ that converts uncertainty over type–action effects into targeted exploration priorities and proposes missing state distinctions.
- To prevent early programmatic commitment and costly world model repairs, we show under ablations that $\widehat{\mathcal{S}}_{\text{B}}$'s type-centric abstraction encourages more generalizable, maintainable, and repairable program patterns than monolithic, object-oriented, and naive methods.
- To preserve non-executable hypotheses, we introduce a hybrid NL, Bayesian, and programmatic model state and show that the NL proto-representation improves initial online learning over a purely programmatic model.

By extending WorldCoder with qualitative and executable world models that improve through uncertainty-aware exploration, the OPINE-World learning loop increases world modeling capability far beyond the base model's standalone performance and solves 80% of public ARC-AGI-3 domains.

2 Problem Setting: Open-world interaction

Consider an agent placed inside an unfamiliar environment. At step t, the environment has a structured state $s_t \in \mathcal{S}$, but only exposes a rendered abstraction ω_t. The agent selects an action a_t from a finite interface $\mathcal{A}$, then receives a reward and the next observation ω_{t+1}:

$$\omega_t = g(s_t), \qquad s_{t+1} = T(s_t, a_t), \qquad r_t = R(s_t, a_t, s_{t+1}). \tag{1}$$

The structured state $\mathcal{S}$, action semantics, transition function T, reward function R, and goal are all withheld. In ARC-AGI-3, the agent can learn only from pixels, returned rewards, and its own interactions. Its evidence after t interactions is $\mathcal{D}_t = \{(\omega_\ell, a_\ell, r_\ell, \omega_{\ell+1})\}_{\ell<t}$. A later level may introduce a new object, relation, action, or reward condition; thereby, the objective is not to merely fit $\mathcal{D}_t$, but to efficiently acquire a model that continuously predicts and explores well enough to

reach an unknown goal. This setting is *open world* from the learner's perspective. This framing tells us two sources of useful interaction. First, a reward-directed plan under optimism drives the agent to a state where its current model claims a useful consequence will occur. If the plan succeeds, the agent solves the task; if it fails, reality provides a goal-relevant counterexample. We call this *goal-directed falsification*. Second, an explicit epistemic mechanism can explore an unresolved mechanic independent of reward paths. This is important for mechanics that can't appear in a reward-directed plan because the current model lacks the ontology needed to represent them. We call this *ontology-error-prioritized interactive exploration*.

3 METHODOLOGY

3.1 MODEL REPRESENTATION

$$\widehat{\mathcal{W}}_t = \big((\widehat{\mathcal{S}}_{\mathrm{NL},t}, \widehat{T}_{\mathrm{NL},t}, \widehat{R}_{\mathrm{NL},t}), \widehat{\mathcal{S}}_{\mathrm{B},t}, (\widehat{T}_{\mathrm{P},t}, \widehat{R}_{\mathrm{P},t}, \widehat{P}_t, \widehat{f}_t^{-1})\big). \tag{2}$$

OPINE-World represents the unknown state, transition, and reward through a unified NL, Bayesian, and executable Python system. At a high level, the representation has five parts:

1. **Known action interface $\mathcal{A}$.** The agent receives the action space (but not their effects) (Section 2).
2. **Unknown state $\mathcal{S}$.** $\widehat{f}^{-1}$ extracts object instances, inferred types, properties, and relations from ω; $\widehat{\mathcal{S}}_{\mathrm{NL}}$ describes this ontology and $\widehat{\mathcal{S}}_{\mathrm{B}}$ quantifies object uncertainty (Sections 3.1.1 and 3.1.2).
3. **Unknown transition T.** $\widehat{T}_{\mathrm{NL}}$ encodes qualitative and partially specified mechanics, while a parallel representation $\widehat{T}_{\mathrm{P}}$ encodes a programmatic transition model (Section 3.1.3).
4. **Unknown reward R.** $\widehat{R}_{\mathrm{NL}}$ encodes candidate goals and unresolved hypotheses, while $\widehat{R}_{\mathrm{P}}$ implements a programmatic reward model (Section 3.1.4).
5. **Planning $\widehat{P}$.** The planner rolls $(\widehat{T}_{\mathrm{P}}, \widehat{R}_{\mathrm{P}})$ forward to pursue reward, or uses $\widehat{T}_{\mathrm{P}}$ with $\widehat{\mathcal{S}}_{\mathrm{B}}$ and the NL model for exploration (Section 3.2.2).

3.1.1 QUALITATIVE MODEL $\widehat{\mathcal{S}}_{\mathrm{NL}}, \widehat{T}_{\mathrm{NL}}, \widehat{R}_{\mathrm{NL}}$ AND QUANTITATIVE $\widehat{\mathcal{S}}_{\mathrm{B}}$

The NL model $\widehat{\mathcal{S}}_{\mathrm{NL}}, \widehat{T}_{\mathrm{NL}}, \widehat{R}_{\mathrm{NL}}$ acts as a proto-representation of our MDP. It can represent partial claims such as *"colour x may be an important confounding variable for object y"*. These uncertainties are quantitatively expressed in a Bayesian belief state $\widehat{\mathcal{S}}_{\mathrm{B}}$, which records empirical effect distributions for object type, action, and observable context from a deterministic, replay-based extractor. A mixed distribution of such observations in a deterministic environment indicates that the current model fails to distinguish a confounding variable. Appendix A explains the `ft09` game used in the examples.

Running example: State in $\widehat{\mathcal{S}}_{\mathrm{NL}}$ and $\widehat{\mathcal{S}}_{\mathrm{B}}$ form

$\widehat{\mathcal{S}}_{\mathrm{NL}}$: qualitative

```
TYPE
c1 is an answer_cell.
PROPERTY
color(c1) is usually yellow, but may be blue rarely?
RELATION
c2 is a pictured, occupied  neighbor of c1.
DERIVED STATE
clicking c1 usually changes one other answer cell.
```

$\widehat{\mathcal{S}}_{\mathrm{B}}$: quantitative

```
{
  "object": {"id": "c1", "type": {"answer_cell": 1}},
  "property": {"color(c1)": {"y": 0.98, "b": 0.02}},
  "relation": { "brush_neighbor(c1,c2)": 0.82,
     "no_neighbour": 0.18 },
  "derived": {
    "cell_brush_arity(c1)": 0.75}
}
```

The probabilities schematically illustrate the current belief given $\mathcal{D}$; Appendix B.4 explains the lifted attribution to effects and contexts.

3.1.2 OBSERVATION-TO-OBJECT ABSTRACTION $\widehat{f}^{-1}$

At each step, $\widehat{f}_t^{-1}$ turns the rendered observation ω_t into the object-oriented state:

$$\widehat{s}_t = \widehat{f}_t^{-1}(\widehat{s}_{t-1}, \omega_t). \tag{3}$$

The resulting object records are the common observable state read by $\widehat{T}_{\mathrm{P}}$, $\widehat{R}_{\mathrm{P}}$, the planner, the verifier, and the Bayesian belief state $\widehat{\mathcal{S}}_{\mathrm{B}}$. This state reconstructs the objects, their inferred types and properties, and relations. When a current record does not determine a recoverable hidden phase, a generated engine may augment it with observed history (Appendix B.1).

3.1.3 PROGRAMMATIC TRANSITION MODEL

The transition program translates the qualitative mechanics $\widehat{T}_{\mathrm{NL}}$ into an executable successor:

$$\widehat{T}_{\mathrm{P},t} : \widehat{\mathcal{S}}_t \times \mathcal{A} \to \widehat{\mathcal{S}}_t, \quad \widehat{s}^{+} = \widehat{T}_{\mathrm{P},t}(\widehat{s}, a), \quad \widehat{s}^{+}[X(o)] = F_{X,t}(o; \widehat{s}, a) \quad \forall X(o) \in \widehat{\mathcal{X}}_t. \tag{4}$$

Each $F_{X,t}$ is the learned update for one fluent mutable state and may read object types, current attributes, grounded action arguments, and derived `@fluent` relations – these fluents are synthesized and mechanically attached to objects to determine the relevant context for Bayesian epistemic uncertainty tracking. This interface therefore contains action grounding, the conditions under which an effect applies, and the resulting changes.

$\widehat{T}_{\mathrm{NL}}$ may retain several plausible or partially specified mechanics while $\widehat{T}_{\mathrm{P},t}$ is deterministic and executable. Repeated application of $\widehat{T}_{\mathrm{P},t}$ supports planning.

Running example: Transitions

$\widehat{T}_{\mathrm{NL}}$**: qualitative**

```
ACTION6(x,y) is a click.

Acts only when (x,y) hits an answer cell c.

The marks on c denote neighboring answer cells.

An object's arity is the number of affected neighbors.

Advance c and those neighbors one palette step. Keep
the rest.

Hypothesis: The palette wraps around when exhausted.
```

$\widehat{T}_{\mathrm{P}}$**: quantitative**

```
@fluent
def cell_brush_arity(c, state):
    return len(affected_neighbors(c, state))
def apply_click(state, action):
    nxt = deepcopy(state)
    if action["action_id"] != 6: return nxt
    hit = hit_answer_cell(nxt, action)
    if hit is None: return nxt
    nbrs = affected_neighbors(hit, nxt)
    for c in [hit, *nbrs]:
        advance_palette(c, nxt)
    return nxt
```

3.1.4 PROGRAMMATIC REWARD MODEL

The reward program assigns a prediction to an extracted transition:

$$\widehat{R}_{\mathrm{P},t} : \widehat{\mathcal{S}}_t \times \mathcal{A} \times \widehat{\mathcal{S}}_t \to \mathbb{R}, \qquad \widehat{r}_k = \widehat{R}_{\mathrm{P},t}(\widehat{s}_k, a_k, \widehat{s}_{k+1}). \tag{5}$$

$(\widehat{s}_k, a_k, \widehat{s}_{k+1})$ allow the reward to depend on a precondition, the action that triggers it, and the resulting state. For sparse completion rewards, the program grounds the current goal hypothesis as a boolean predicate $\widehat{G}_t$ (Section 3.2.4) and uses:

$$\widehat{R}_{\mathrm{P},t}(\widehat{s}, a, \widehat{s}') = r_t^0 + \big(r_t^+ - r_t^0\big)\widehat{G}_t(\widehat{s}, a, \widehat{s}'), \qquad \widehat{G}_t \in \{0, 1\}. \tag{6}$$

Appendix A.1 illustrates the corresponding NL and Python representations for `ft09`.

3.2 MODEL LEARNING LOOP

1. **Observe.** Receive (ω_{t+1}, r_t) after executing a_t (Section 3.2.2).
2. **Abstract and store.** Append the transition to replay buffer $\mathcal{D}_{t+1}$, apply $\widehat{f}_t^{-1}$, associate objects across ω_t and ω_{t+1}, and update $\widehat{\mathcal{S}}_{\mathrm{B},t+1}$ (Sections 3.1.2 and 3.2.3).
3. **Counterexample-guided inductive synthesis (CEGIS).** Compare the prediction with the observed truth. Treat each disagreement as a counterexample for the next candidate program and synthesize a revised $(\widehat{T}_{\mathrm{P}}, \widehat{R}_{\mathrm{P}}, \widehat{P}, \widehat{f}^{-1})$ under sufficient batches (Section 3.2.1) (Solar-Lezama et al., 2006; Jha et al., 2010; Alur et al., 2013).
4. **Choose exploitation or exploration.** For a consistent model, use $\widehat{P}(\mathcal{S}; \widehat{T}_{\mathrm{P}}, \widehat{R}_{\mathrm{P}})$ to pursue a goal or $\widehat{P}(\mathcal{S}; \widehat{T}_{\mathrm{P}})$, informed by $(\widehat{\mathcal{S}}_{\mathrm{NL}}, \widehat{T}_{\mathrm{NL}}, \widehat{R}_{\mathrm{NL}})$ and $\widehat{\mathcal{S}}_{\mathrm{B}}$, to explore (Sections 3.2.2 and 3.2.3).
5. **Execute the next actions**, producing a new observation ω (Section 3.2.2).

See Appendix B.8 for in-depth pseudo-code of the world model learning loop.

3.2.1 PREDICTIVE VERIFICATION Φ_1

Let V select the successor fields compared by the verifier and let $\widehat{s}_\ell = \widehat{f}_t^{-1}(\omega_\ell)$. The current executable hypothesis is replay-consistent when

$$\Phi_1(\widehat{f}_t^{-1}, \widehat{T}_{\mathrm{P},t}, \widehat{R}_{\mathrm{P},t}; \mathcal{D}_t) \iff \bigwedge_{\ell<t} \Big[V\Big(\widehat{T}_{\mathrm{P},t}(\widehat{s}_\ell, a_\ell)\Big) = V(\widehat{s}_{\ell+1}) \ \wedge \ \widehat{R}_{\mathrm{P},t}(\widehat{s}_\ell, a_\ell, \widehat{s}_{\ell+1}) = r_\ell \Big]. \quad (7)$$

The comparator V compares the predicted object-record projection with the extracted successor. Passing Φ_1 means that the selected fields and rewards are consistent with observed transitions.

When does each representation update? $\mathcal{D}$ and $\widehat{\mathcal{S}}_{\mathrm{B}}$ update on every interaction. The action agent revises $(\widehat{\mathcal{S}}_{\mathrm{NL}}, \widehat{T}_{\mathrm{NL}}, \widehat{R}_{\mathrm{NL}})$ when ω supports, rejects, or distinguishes a hypothesis. Synthesis also reconciles these components with repaired code. A prediction mismatch stops the current plan and opens a short collection window for related counterexamples. (Appendix C).

3.2.2 SOURCES OF INFORMATIVE INTERACTION

The planner $\widehat{P}_t$ searches the programmatic model for a sequence $\pi = (a_0, \ldots, a_{L-1})$. Starting from $s^0 = \widehat{f}_t^{-1}(\omega_t)$, define

$$\Phi_2\Big(\pi; s^0, \widehat{T}_{\mathrm{P},t}, \widehat{R}_{\mathrm{P},t}\Big) \iff \max_{0\le k<L} \widehat{R}_{\mathrm{P},t}(s^k, a_k, s^{k+1}) > 0. \quad (8)$$

Satisfying Φ_2 gives a *reward-directed plan*. There are two informative outcomes. 1) The predictions hold and the plan reaches reward. 2) A prediction fails and Φ_1 returns a task-relevant counterexample for model repair. Thus, this is an active model-testing mechanism.

An *epistemic experiment* is an additional source of experience. Posterior uncertainty in $\widehat{\mathcal{S}}_{\mathrm{B}}$ identifies under-sampled or conflicting object–action effects, and the semantic world agent uses $(\widehat{\mathcal{S}}_{\mathrm{NL}}, \widehat{T}_{\mathrm{NL}}, \widehat{R}_{\mathrm{NL}})$ to propose actions that aim to find the confounder. This experiment is useful when no Φ_2-satisfying plan exists.

3.2.3 BAYESIAN EFFECT UNCERTAINTY AND STATE REFINEMENT

$\widehat{\mathcal{S}}_{\mathrm{B}}$ groups object transitions into their observed effect under unique value combinations of τ, a, c:

$$j = (\tau, a, c), \quad (9)$$

where τ is an inferred object type, a is the action, and c summarizes relevant context. For effect e, let $C_t(j, e)$ be its count, $n_j = \sum_e C_t(j, e)$, and m the effect alphabet size. With a symmetric Dirichlet prior α_0,

$$\widehat{q}_j(e) = \frac{\alpha_0 + C_t(j, e)}{m\alpha_0 + n_j}, \qquad \eta_j = \frac{\mathrm{H}(\widehat{q}_j)}{\log m}. \quad (10)$$

With $\alpha_0 = 1$, this gives Laplace smoothing. We use the posterior predictive entropy η_j as a diagnostic of sparse evidence or unresolved effect distinctions under the deterministic modeling assumption. To target the latter, the mixed groups with the largest $n_j\eta_j$, the group's total unexplained entropy, are passed to synthesis where a η_j minimizing split is kept if discovered. Each mixed group is then divided by the values of its resulting contexts, and each subgroup is scored as $1 - \frac{\text{count of its most common outcome}}{1+\text{number of observations}}$ and provided as an actor-facing exploration priority (Appendix B.4).

3.2.4 GOAL GROUNDING AND CHECKED PLANNING

After a reward observation or a revised qualitative goal, a goal-grounding agent converts $\widehat{R}_{\mathrm{NL}}$ into a bounded set of Boolean requirements over extracted objects given a new level into $\widehat{R}_{\mathrm{P}}$. The planner rolls candidate actions through $\widehat{T}_{\mathrm{P}}$ and searches for $\Phi_2 = 1$. Before and during use, predicted successors are checked by Φ_1 and subjected to CEGIS.

For our factored-dynamics interpretation refer to Appendix B.2.

4 EXPERIMENTAL SETUP

RQ1: Does our Bayesian epistemic guidance $\widehat{\mathcal{S}}_{\mathrm{B}}$ improve exploration? Alongside a comparison against other systems, we compare the full system against an $\widehat{\mathcal{S}}_{\mathrm{B}}$ ablation, and an ϵ-greedy baseline (Appendix F.2) on the 25 public ARC-AGI-3 games with a 2000-action budget per game, or on a stall of more than 500 actions per level. We measure the per-game difference in RHAE (Appendix G) and inspect the action reasoning traces to identify the failure sources for all 25 games (Appendix F.4).

RQ2: How does the program structure used to encode transition semantics affect generalizability, adaptability, and maintainability (code metrics) of the programmatic world model? Under a fixed action trajectory for all runs, we both prompt-restrict and code-structure-restrict the programmatic world model outputted by the synthesizer agent in various ways:

1. **Monolithic Code Structure.** Classic MDP framing, where the whole world's transitions are described in a single monolithic transition function.
2. **Type-Oriented Code Structure.** An object-oriented framing where transition functions for an object type are defined within each object class, or a type-relational class.
3. $\widehat{\mathcal{S}}_{\mathrm{B}}$ **Code Structure (Unablated).** A hybrid framing encouraged naturally by the Bayesian Belief State where transition functions for an object type are encouraged to be defined within each object type's class only where easily definable through the Bayesian Belief State.
4. **Unrestricted (No $\widehat{\mathcal{S}}_{\mathrm{B}}$).** A completely unconstrained code synthesis for control purposes.

After confirming that the interventions produce different program structures (Appendix C.3), we measure generalizability, adaptability, and maintainability. Adaptability is assessed through locality and synthesis cost of code revisions from a modified environment, measured by changed source lines, engine rewrite percentage, functions changed, synthesis rounds, and output tokens. Generalizability is measured as the prediction accuracy on unseen state transitions across these modified environments. To facilitate these measurements, we change an existing object mechanic (M1), introduce new objects and relations (M2), and change a global rule (M3) (Appendix F.7).

RQ3: Will ablating the NL world model $(\widehat{\mathcal{S}}_{\mathrm{NL}}, \widehat{T}_{\mathrm{NL}}, \widehat{R}_{\mathrm{NL}})$ and using a pure code representation perform as well as our hybrid NL and programmatic world model approach? We compare against an NL ablation that prevents the generation of an explicit NL world model as files. Because the synthesis agent's context is reset at each stage, we can consider this as removing the persistent NL representation of the world. We hypothesize that NL can better represent hypotheses and unresolved correlative observations, which should increase the RHAE and decrease the model-revision cost.

We use **Claude Opus 4.8 high 1m** under Docker isolation from game source and network access. RQ2 and RQ3 studies use low effort and `g50t` as a demanding case study combining history-dependent dynamics, interacting object types, and progressively introduced mechanics (Appendix C.1). The supplementary material contains the full implementation of OPINE-World (Appendix C), and ablation details (Appendix C.1).

5 RESULTS

5.1 OVERALL BENCHMARK PERFORMANCE

OPINE-World achieves the highest RHAE among comparators and completes the most games under matched game versions and scoring (Table 1; per-game results in Appendix F.1). On games completed by both OPINE-World and the strongest comparator, the main advantage is level completion rather than efficiency, suggesting that broader discovery of mechanics contributes more to the aggregate gain than shorter solutions to familiar tasks (RQ1).

We compare against non-ensemble, structured approaches to reasoning that allow a scientific comparison, analysis of design choices, and their contribution to LLM performance. Therefore, we omit the comparison of methods that were built on the structure of the OPINE-World, pure black-box approaches, and methods that require a foundation model with a significant difference in base capability, as these contributions are beyond our ability to explain.

Table 1: **Aggregate RHAE and coverage.** OPINE-World, the $\widehat{\mathcal{S}}_{\mathrm{B}}$ ablation, and other systems. Standard errors are computed across the 25 games. Per-game results appear in Appendix F.1.

	OPINE-World		Other systems			
Metric	Full	No $\widehat{\mathcal{S}}_{\mathrm{B}}$	baseline1	DreamTeam	Continual	OpenClaw
Mean ± SE	**78.4 ± 6.3**	58.2 ± 8.3	63.7 ± 7.4	38.1 ± 7.6	20.5 ± 5.7	5.2 ± 1.0
Levels completed	**160/183**	126/183	146/183	98/183	64/183	28/183
Games completed	**20/25**	12/25	14/25	6/25	3/25	0/25

5.2 RQ1: Effect of Bayesian guidance on online discovery

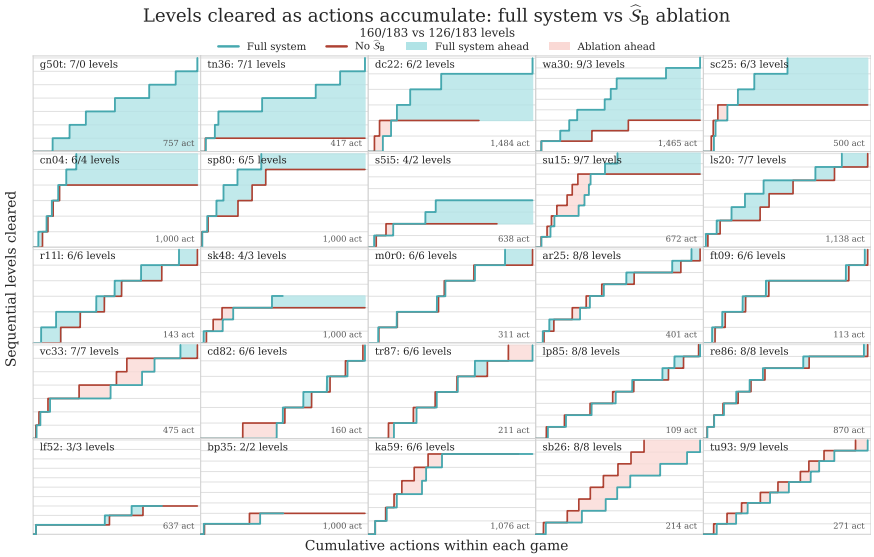


Figure 2: **Online progression with and without Bayesian guidance.** Each step marks a completed level. Blue or red shading marks intervals where the denoted system leads. Ordered by blue area.

The largest gains over the $\widehat{\mathcal{S}}_{\mathrm{B}}$ ablation occur when the agent fails to discover a mechanic needed for progress (Figure 2). In `g50t` and `dc22`, the ablated agent repeats similar sequences or configurations while unresolved interactions remain untested. In `cn04`, it fails to sufficiently model a button, and in `sc25`, it misses an earlier projectile mechanic. Missing an interaction required by the solution prevents level completion, even if the agent has learned the other mechanics correctly, indicating that such exploratory oversights can be catastrophic. These failures suggest that Bayesian guidance helps the agent move beyond familiar actions when its current model fails to explain how to make progress.

The effect can extend beyond the level in which a mechanic is first encountered. In `ls20`, the ablated agent fails to transfer a tested object-class property; in `sp80`, insufficient early probing leaves an incorrect mechanic in the model used for later planning. In some games, additional exploration is not useful, especially when exploration targets uncertainties that are entirely irrelevant to the goal; however, this often cannot be known in advance.

We additionally measure against a naive exploration method: an ϵ-greedy exploration baseline (Figure 6). We see that random object exploration is insufficient to provide valuable evidence, and often is detrimental to goal planning if sufficient knowledge has been accrued. This motivates the case for a more principled exploration method such as $\widehat{\mathcal{S}}_{\mathrm{B}}$. Detailed case-by-case review and trace excerpts are provided in Appendices F.4 and F.5.

5.3 RQ2: Effect of program structure on prediction and revision

With the same evidence supplied to synthesis, $\widehat{\mathcal{S}}_{\mathrm{B}}$ Code Structure (Unablated) has the highest accuracy on unseen transitions across the RQ2 conditions (Figure 3). The Type-Oriented variant performs worst

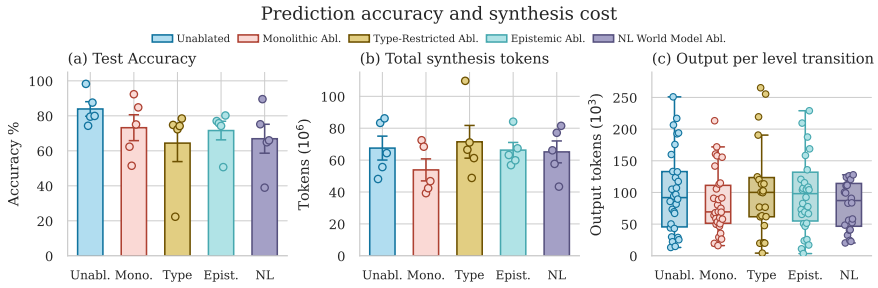


Figure 3: **Prediction and synthesis cost under same evidence.** Error bars show standard errors.

despite using more tokens on average. Upon inspection, we found that successful models encoded object transitions in a shared monolithic structure because heavily coupled effects are difficult to assign to individual objects. As a result, programs that were learned with $\widehat{\mathcal{S}}_{\mathrm{B}}$ Code Structure retained object-indexed state while allowing interactions to be resolved jointly, suggesting that the organization of transition rules should remain flexible even when the state is best represented as object-centric.

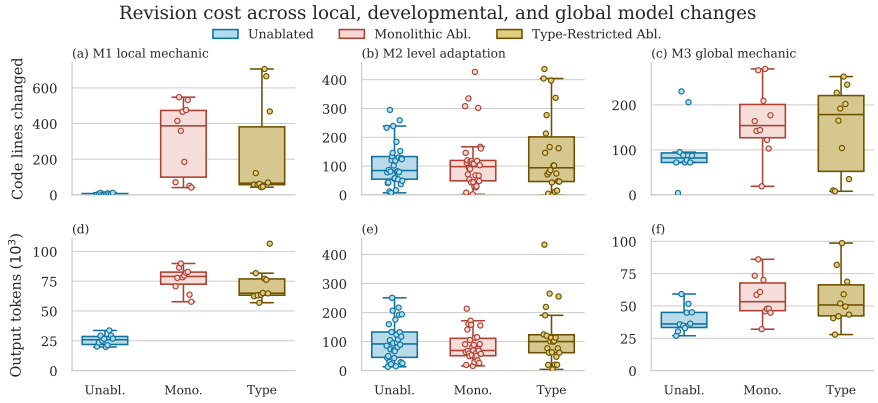


Figure 4: **Revision costs for local changes, new level mechanics, and global changes.**

The clearest revision advantage for $\widehat{\mathcal{S}}_{\mathrm{B}}$ Code Structure occurs when a local mechanic changes (M1): it edits fewer definitions and requires less rewriting, fewer synthesis rounds, and fewer output tokens than Monolithic Code Structure and Type-Oriented Code Structure (Figure 4). Interestingly we see that the type-oriented structure has a large variance. Upon inspection, these costly outliers were due to overcommitment to structure, resulting in wrongly abstracted typings during periods of uncertainty. These are very expensive to repair and were avoided with the type-uncertainty information from $\widehat{\mathcal{S}}_{\mathrm{B}}$.

However, when later levels introduce new mechanics (M2), we see that a new level usually requires significant changes even across multiple types, reducing the advantage of type abstraction. Interestingly, we found that the monolithic system has less structure, giving it an innate advantage for organizational cost that was counteracted by the code's difficulty to maintain. Although costs are similar across methods, the superior generalization of the unablated system shows that such adaptation is more robustly incorporated. Additionally, we see in M3 that although $\widehat{\mathcal{S}}_{\mathrm{B}}$ Code Structure uses fewer output tokens, synthesis round count is comparatively close. Program organization can therefore reduce the tokens required to repair the model each synthesis round. See Appendix F.7 for details.

5.4 RQ3: Effect of persistent NL on model learning

Compared to NL ablation, the complete system requires fewer actions, synthesis rounds, and tokens during early learning when object behaviors are still poorly understood (Figure 5). In the fixed-evidence RQ2 comparison, $\widehat{\mathcal{S}}_{\mathrm{B}}$ Code Structure also has higher mean held-out accuracy than No

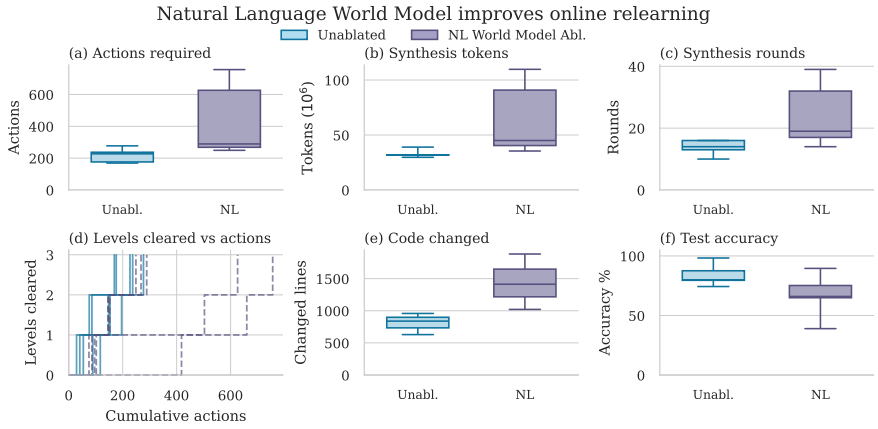


Figure 5: **Model learning with and without persistent NL.** For graphs a–c and e, lower is better. For graphs d and f, higher is better. Detailed run values and tests are provided in Appendix F.8.

NL. This suggests that persistent language helps synthesis interpret observations in addition to any improvement in action selection.

Both conditions encode latent histories and executable mechanics in Python. The full system's artifacts additionally retain the observations supporting a hypothesis, superseded explanations, and unresolved questions. These records preserve hypotheses when the evidence is insufficient. In the NL ablation traces, failures often concern sparsely tested behaviors with several plausible explanations. Retaining these qualitative explanations helps later synthesis revise an uncertain model without needing to reconstruct why its rules were introduced.

6 Related Work

There is a vast amount of work on world models. We defer a broad summary of related areas to Appendix D. Successful recent systems on ARC-AGI-3 combine frontier models with graph-based exploration, executable world models, multi-agent workspaces, and continual agent programmatic harnesses (Rudakov et al., 2025; Rodionov, 2026; Sarafian et al., 2026; Lehmann et al., 2026; Skoutnev et al., 2026; Karten et al., 2026; Furgale et al., 2026), but lack formal approaches to efficient, information-informed exploration. OPINE-World uses a hybrid NL, Bayesian, and programmatic world model to guide exploration and revision, allowing a comparatively weak base model to greatly improve its capability in causal reasoning and world model building.

7 Limitations

We limit our evaluation to deterministic discrete ARC-AGI-3 environments, extension to stochastic or continuous domains would require significant extensions. Replay consistency also provides no guarantees for entirely unvisited objects. Appendix E discusses these limitations in detail.

8 Conclusion

We introduced OPINE-World, which combines Bayesian epistemic guidance, NL world models, and programmatic world models for online world-model learning. On ARC-AGI-3, we show that Bayesian guidance reduces catastrophic discovery failures, $\widehat{\mathcal{S}}_{\mathrm{B}}$ Code Structure supports efficient repair, and an NL proto-representation reduces learning costs. These results support preserving a hybrid NL-Programmatic world model and using $\widehat{\mathcal{S}}_{\mathrm{B}}$for open-world problem solving.

Reproducibility Statement

We provide source code for OPINE-World as supplementary material. Its README documents installation, how to fetch the public ARC-AGI-3 games, the Docker sandbox, and a launch script that runs games with the configuration used for our main results and ablations in RQ1 to RQ3. The models, reasoning effort, action budgets, and the constants of the learning loop and of $\widehat{\mathcal{S}}_{\mathrm{B}}$ are listed in Appendix C, the verifier and the admission checks are specified in Appendix B, and scoring follows Appendix G. Because the agents are language models acting in a live environment, repeated runs vary in their action counts; the statistical tests in Appendix C account for this variation.

References

Zergham Ahmed, Joshua B. Tenenbaum, Christopher J. Bates, and Samuel J. Gershman. Synthesizing world models for bilevel planning. *Transactions on Machine Learning Research*, 2025. URL https://openreview.net/forum?id=m9V4JHLJrD.

Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *FMCAD*, pp. 1–8, 2013.

ARC Prize Foundation. ARC-AGI-3: A new challenge for frontier agentic intelligence. *arXiv preprint arXiv:2603.24621*, 2026a. URL https://arxiv.org/abs/2603.24621.

ARC Prize Foundation. ARC-AGI toolkit changelog. Documentation, 2026b. URL https://docs.arcprize.org/changelog. Accessed 21 September 2026.

ARC Prize Foundation. ARC-AGI-3 scorecards. Documentation, 2026c. URL https://docs.arcprize.org/scorecards. Accessed 21 September 2026.

Peter W. Battaglia, Jessica B. Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

G. Ann Campbell. Cognitive complexity: An overview and evaluation. In *TechDebt*, pp. 57–58, 2018. doi: 10.1145/3194164.3194186.

Carlos Diuk, Andre Cohen, and Michael L. Littman. An object-oriented representation for efficient reinforcement learning. In *ICML*, pp. 240–247, 2008.

Paul Furgale, Severin Klingler, James Nolan, Matt Staats, Gaia Di Lorenzo, Elisa Martinez Abad, Christian Schüller, Razvan Dinu, Alessio Devoto, Pascal Berard, Gal Kaplun, Elad Sarafian, Riccardo Roveri, Leon Derczynski, and Ricardo Silveira Cabral. NVIDIA-labs OO Agents: Native Python object-oriented agents. *arXiv preprint arXiv:2607.20709*, 2026. URL https://arxiv.org/abs/2607.20709.

Andrew Gelman, John B. Carlin, Hal S. Stern, David B. Dunson, Aki Vehtari, and Donald B. Rubin. *Bayesian Data Analysis*. Chapman and Hall/CRC, third edition, 2013.

Carlos Guestrin, Daphne Koller, Ronald Parr, and Shobha Venkataraman. Efficient solution algorithms for factored MDPs. *Journal of Artificial Intelligence Research*, 19:399–468, 2003.

David Ha and Jürgen Schmidhuber. Recurrent world models facilitate policy evolution. In *NeurIPS*, pp. 2455–2467, 2018.

Danijar Hafner, Jurgis Pasukonis, Jimmy Ba, and Timothy Lillicrap. Mastering diverse control tasks through world models. *Nature*, 640(8059):647–653, 2025. doi: 10.1038/s41586-025-08744-2.

Albert O. Hirschman. The paternity of an index. *American Economic Review*, 54(5):761, 1964.

Rein Houthooft, Xi Chen, Yan Duan, John Schulman, Filip De Turck, and Pieter Abbeel. VIME: Variational information maximizing exploration. In *NeurIPS*, 2016.

Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *ICSE*, pp. 215–224, 2010.

Seth Karten, Joel Zhang, Tersoo Upaa, Jr., Ruirong Feng, Wenzhe Li, Chengshuai Shi, Chi Jin, and Kiran Vodrahalli. Continual harness: Online adaptation for self-improving foundation agents. *arXiv preprint arXiv:2605.09998*, 2026. URL https://arxiv.org/abs/2605.09998.

Thomas Kipf, Elise van der Pol, and Max Welling. Contrastive learning of structured world models. In *ICLR*, 2020. URL https://openreview.net/forum?id=H1gax6VtDB.

Michele Lacchia. Radon 6.0.1 documentation. Software documentation, 2023. URL https://radon.readthedocs.io/en/stable/. Accessed 21 September 2026.

Jens Lehmann, Andrei Aioanei, and Sahar Vahdati. Tycho: Active abstraction with programmatic world models for ARC-AGI-3. *arXiv preprint arXiv:2607.28287*, 2026. URL https://arxiv.org/abs/2607.28287.

Guillaume Levy, Cédric Colas, Pierre-Yves Oudeyer, Thomas Carta, and Clément Romac. WorldLLM: Improving LLMs' world modeling using curiosity-driven theory-making. *arXiv preprint arXiv:2506.06725*, 2025. URL https://arxiv.org/abs/2506.06725.

Dennis V. Lindley. On a measure of the information provided by an experiment. *Annals of Mathematical Statistics*, 27(4):986–1005, 1956.

Xiaotian Liu, Armin Toroghi, Jiazhou Liang, David Courtis, Ruiwen Li, Ali Pesaranghader, Jaehong Kim, Tanmana Sadhu, Hyejeong Jeon, and Scott Sanner. Natural language PDDL (NL-PDDL) for open-world goal-oriented commonsense regression planning in embodied AI. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=kWCNhRdcDI.

Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4): 308–320, 1976. doi: 10.1109/TSE.1976.233837.

Andrew K. McCallum. *Reinforcement learning with selective perception and hidden state*. PhD thesis, University of Rochester, 1996.

Deepak Pathak, Pulkit Agrawal, Alexei A. Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. In *ICML*, pp. 2778–2787, 2017.

Wasu Top Piriyakulkij, Yichao Liang, Hao Tang, Adrian Weller, Marta Kryven, and Kevin Ellis. PoE-World: Compositional world modeling with products of programmatic experts. In *NeurIPS*, 2025. URL https://arxiv.org/abs/2505.10819.

Sergey Rodionov. Executable world models for ARC-AGI-3 in the era of coding agents. In *AGI*, volume 16855 of *Lecture Notes in Computer Science*, pp. 198–210. Springer, 2026. doi: 10.1007/978-3-032-33195-3_15. URL https://arxiv.org/abs/2605.05138.

Evgenii Rudakov, Jonathan Shock, and Benjamin Ultan Cowley. Graph-based exploration for ARC-AGI-3 interactive reasoning tasks. *arXiv preprint arXiv:2512.24156*, 2025. URL https://arxiv.org/abs/2512.24156. AAAI 2026 Workshop on AI for Scientific Research.

Scott Sanner. Relational dynamic influence diagram language (RDDL): Language description. Unpublished manuscript, Australian National University, 2010. URL http://users.cecs.anu.edu.au/~ssanner/IPPC_2011/RDDL.pdf.

Elad Sarafian, Gal Kaplun, Ron Banner, Daniel Soudry, and Boris Ginsburg. Workspace optimization: How to train your agent. *arXiv preprint arXiv:2605.09650*, 2026. URL https://arxiv.org/abs/2605.09650.

Ramanan Sekar, Oleh Rybkin, Kostas Daniilidis, Pieter Abbeel, Danijar Hafner, and Deepak Pathak. Planning to explore via self-supervised world models. In *ICML*, pp. 8583–8592, 2020.

Claude E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27: 379–423, 623–656, 1948.

Alexy Skoutnev, Kirill Acharya, Gaston Longhitano, Madeleine Udell, Kevin Ellis, and Iddo Drori. Twin: Playing an unknown game with a test-time digital twin. *arXiv preprint arXiv:2608.14490*, 2026. URL https://arxiv.org/abs/2608.14490.

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, pp. 404–415, 2006.

Alexander L. Strehl, Carlos Diuk, and Michael L. Littman. Efficient structure learning in factored-state MDPs. In *AAAI*, pp. 645–650, 2007.

Richard S. Sutton. Dyna, an integrated architecture for learning, planning, and reacting. *ACM SIGART Bulletin*, 2(4):160–163, 1991.

Hao Tang, Darren Key, and Kevin Ellis. WorldCoder, a model-based LLM agent: Building world models by writing code and interacting with the environment. In *NeurIPS*, 2024.

A Worked Example: ft09

Running example: ft09 Example Domain

Consider this elementary ARC-AGI-3 problem. A set of answer squares neighbors goal squares. Clicking an answer square advances its color through the ordered legend; in later levels, the click may also change nearby squares. Each directional position on a neighboring goal square specifies the target color of the answer square in that relative position. The goal is to make every answer square match its directional goal square colour.
Let $\mathcal{C}$ be the answer squares, $\mathcal{G}$ the goal squares, $\mathcal{P}$ the ordered legend, $\gamma(c)$ the target decoded from $\mathcal{G}$, and $B(c,d)$ indicate that clicking c also changes d. Writing $\texttt{CLICK}(c)$ for a coordinate click that hits c:

$$\textbf{State:}\quad \mathcal{S}=\{s:\mathcal{C}\to\mathcal{P}\},\qquad s(c)=\texttt{color}(c),$$

$$\textbf{Action:}\quad \mathcal{A}=\{\texttt{CLICK}(x,y):x,y\in\{0,\dots,63\}\},\qquad |\mathcal{A}|=64^2=4096,$$

$$\textbf{Transition:}\quad T(s,\texttt{CLICK}(c))(d)=\begin{cases}\texttt{next}_{\mathcal{P}}(s(d)), & d=c\ \vee\ B(c,d),\\ s(d), & \text{otherwise},\end{cases}$$

$$\textbf{Reward:}\quad R(s,a,s')=\mathbf{1}\left[\forall c\in\mathcal{C},\ s'(c)=\gamma(c)\right].$$

Factored dynamics. In this example, color is a state fluent, the grounded click is an action fluent, and the palette, key pixels, and cell geometry are fixed within a level. The transition can be written as a deterministic CPF (Appendix B.2):

$$\texttt{advance}(c;s,a)\equiv\texttt{click}(c)\vee\exists d\,[\texttt{click}(d)\wedge\texttt{brushes}(d,c;s)], \tag{11}$$

$$\texttt{color}'(c)=\begin{cases}\texttt{next}_{\texttt{palette}}(\texttt{color}(c)), & \texttt{advance}(c;s,a),\\ \texttt{color}(c), & \text{otherwise}.\end{cases} \tag{12}$$

Its parents include the current color, ordered palette, click target, and brush relation. The derived fluent $\texttt{cell_brush_arity}(d)=|\{c:\texttt{brushes}(d,c;s)\}|$ counts how many other cells change; the binary relation identifies which cells change. The same neighborhood geometry determines $\texttt{required_color}(c)$, used by the reward model. These dependencies localize prediction errors while allowing Python to compute coupled effects jointly.

A.1 Qualitative and programmatic reward representations

Running example: Reward

$\widehat{R}_{\mathrm{NL}}$: qualitative

```
Each key assigns a required color to an answer
cell.

A cell is satisfied when its color equals its
required color and there is no submit action.

Reward is 1 exactly when every answer cell
is satisfied at once.

Hypothesis: On three-color levels, gray may
be a neutral target.

Heuristic: Try high arity keys first.
```

$\widehat{R}_{\mathrm{P}}$: quantitative

```
@fluent
def required_color(c, state):
    return unique_value(key_demands(c, state))

@fluent
def satisfied(c, state):
    return color(c, state) == required_color(c, state)

def reward_function(state, action, _):
    after_click = apply_click(state, action)
    solved = all(satisfied(c, after_click)
                 for c in answer_cells(after_click))
    return float(solved)
```

In `ft09`, $r_t^0 = 0$, $r_t^+ = 1$, and the inferred completion predicate is

$$\widehat{G}_t(\widehat{s}') = \mathbf{1}\Big[\forall c \in \widehat{\mathcal{C}}(\widehat{s}'),\ \texttt{color}(c;\widehat{s}') = \texttt{required_color}(c;\widehat{s}')\Big]. \tag{13}$$

Notice that NL and Python contain different abstraction levels. Keeping both preserves compact, nuance-rich explanations without weakening executability.

B RDDL Formalization and Diagnostic Details

For compactness, this appendix writes $\widehat{T}$, $\widehat{R}$, and $\widehat{P}$ for the programmatic components $\widehat{T}_{\mathrm{P}}$, $\widehat{R}_{\mathrm{P}}$, and $\widehat{P}_t$ introduced in the main text. We use the vocabulary of RDDL and factored MDPs to describe their object-based state and transition structure (Sanner, 2010; Guestrin et al., 2003).

B.1 Observation abstraction and grounded fluent states

The environment emits an observation ω_t. Let f be the rendering map and let f^{-1} denote the learned or supplied abstraction:

$$\omega_t = f(s_t), \qquad \widehat{s}_t = f^{-1}(\widehat{s}_{t-1}, \omega_t). \tag{14}$$

Equation 14 uses f^{-1} to denote a parser. The extracted state is amodal, subject to the LLM priors, and can infer occluded objects. That is, a generated engine may maintain history $\widehat{s}_{t-1}$ which includes the extracted record and the retained history. Replay requires the current values of $\widehat{s}_t$ to contain the information needed to predict the successor. This construction covers hidden state that can be recovered deterministically from the observed trajectory. A model of irreducible ambiguity, such as random effects, would require a distribution over possible states, which is out of the scope of this system.

Throughout, $\widehat{s}_t$ includes any relevant history reconstructed from replay, so the Markov transition $\widehat{T}(\widehat{s}_t, a_t)$ conditions on this augmented state.

Each extracted object record contains an identity, inferred type, and fields such as position, pixels, color, rotation, and visibility. Matching records across observations distinguishes changes to an existing object from appearances and disappearances.

Table 2: **RDDL notation.** Symbols used in the method.

Symbol	Meaning
$\mathcal{T}, \mathcal{O}$	object types and their concrete instances
$\mathcal{X}$	state properties written with object variables
$\mathcal{A}$	action indicators and the action a_t executed at step t
$\mathcal{N}$	level information that actions don't alter, such as walls
$\mathcal{I}_t$	values produced at step t by derived functions of the current state and action
$X(\bar{o})$	schema $X(\bar{z})$ grounded with a legal object tuple $\bar{o}$
Val_X	allowed values of state property X
Pa_X	selects the inputs used by the update rule for X
$u_t^X(\bar{o})$	values of those selected inputs for objects $\bar{o}$ at step t
$\tau(o)$	assigns object o to a type whose members share an update-rule template
$\mathcal{D}_t$	observed transition and reward history

A lifted schema $X(\bar{z})$ has k typed object arguments $\bar{z} = (z_1, \ldots, z_k)$. Grounding it with compatible objects $\bar{o} = (o_1, \ldots, o_k)$ produces $X(\bar{o})$. The state assigns one value to each legal grounding:

$$s_t = \{X_t(\bar{o}) = v_t^{X,\bar{o}} \mid X \in \mathcal{X},\ \bar{o} \in \mathrm{Ground}(X),\ v_t^{X,\bar{o}} \in \mathsf{Val}_X\}. \tag{15}$$

In Equation 15, $\mathrm{Ground}(X)$ is the set of legal object tuples for schema X, Val_X is its value domain, and $v_t^{X,\bar{o}}$ is the value assigned to grounding $X(\bar{o})$ at step t. For example, $x(p_1)$, $\texttt{shape}(k_1)$, and $\texttt{locked}(d_1)$ are grounded fluents. The implementation stores fluents with a shared object argument in the same object record. This record is a storage representation of the grounded assignment.

Running example: Grounded states

Suppose the player is at cell $(2, 3)$ and action 4 means move right. An open destination gives

$$x_t(p_1) = 2,\ y_t(p_1) = 3 \quad \longrightarrow \quad x_{t+1}(p_1) = 3,\ y_{t+1}(p_1) = 3.$$

If a wall occupies the destination, the same action leaves $x_{t+1}(p_1) = 2$. Both transitions assign values to the schemas $x(p)$ and $y(p)$.

Table 3: **Fluent vocabulary for the appendix micro-example.** The examples give simple names to roles inferred in an ARC-AGI-3 game environment named `ls20`.

Category	Meaning in the world-model setting
$\mathcal{T}$	object types or shared templates, such as player, key, rotator, door, and wall
$\mathcal{O}$	objects in the current level, such as player p_1, key k_1, rotator r_1, and door d_1
$\mathcal{N}$	level information that actions do not alter, such as wall cells, fixed object locations, and a door's target key shape
$\mathcal{X}$	state properties written with object variables, such as $x(p)$, $y(p)$, $\texttt{shape}(k)$, and $\texttt{locked}(d)$
$\mathcal{A}$	action indicators such as $\texttt{move_right}(p)$
$\mathcal{I}$	derived functions such as $\texttt{blocked_right}(p)$ and $\texttt{at_rotator}(p, r)$
CPF	update rule for one next-state fluent, such as the rule for $x'(p)$

B.2 Factored dynamics and conditional probability functions

A conditional probability function (CPF) specifies each next-state fluent from a parent set. Non-fluents describe level information that an action does not alter. In the running example, this includes the wall map and the locations of the fixed rotator and door. A CPF can read this information when predicting what changes.

RDDL intermediate fluents are implemented here as derived functions of the current state and action. Their values are recalculated before the model predicts the next state rather than stored across steps. For example, $\texttt{blocked_right}_t(p)$ returns whether a wall occupies the cell to the player's right. The position rule uses this result to decide whether movement succeeds. Model-declared `@fluent`

functions expose derived object properties and relations to the Bayesian diagnostics, subject to the admission checks in Appendix B.5.

For a next-state fluent $X'(\bar{z})$ and a grounding $\bar{o}$, the CPF has the form

$$P(X_{t+1}(\bar{o}) \mid \mathrm{Pa}_X(\bar{o}; s_t, a_t, \mathcal{N}, \mathcal{I}_t)) . \tag{16}$$

Equation 16 uses Pa_X for the parent or context set of $X'(\bar{o})$. It selects the subset of current state fluents, action fluents, non-fluents, and intermediate fluents relevant to predicting this next fluent. The experiments are deterministic, which reduces this conditional distribution to a point mass:

$$P(X_{t+1}(\bar{o}) = v \mid s_t, a_t) = \mathbf{1}\{v = f_X(\bar{o}; u_t^X(\bar{o}))\} . \tag{17}$$

In Equation 17, v is a candidate next value and $\mathbf{1}\{\cdot\}$ equals one when its condition is true and zero otherwise. The equation assigns all probability to the value returned by f_X.

B.2.1 A TINY LIFTED MODEL

The following lifted CPFs serve as an example domain.

```
blocked_right(p) :=
  wall_at(x(p)+1, y(p))

x'(p) =
  if move_right(p) and not blocked_right(p)
  then x(p)+1
  else x(p)

y'(p) =
  y(p)

at_rotator(p,r) :=
  x(p)=rotator_x(r) and y(p)=rotator_y(r)

shape'(k) =
  if exists p,r: at_rotator(p,r)
  then rotate(shape(k))
  else shape(k)

key_fits(k,d) :=
  shape(k)=target_shape(d)

at_door(p,d) :=
  x(p)=door_x(d) and y(p)=door_y(d)

locked'(d) =
  if exists k: key_fits(k,d)
  then false
  else locked(d)
```

First, the CPFs are *local*. The update for $x'(p)$ does not need the complete frame. It needs the current position, the relevant action, and a local obstacle condition summarized by `blocked_right`(p). Second, the CPFs have *semantic bodies*: the learner must infer which fluent an input changes, by how much, and under which conditions. Third, the CPFs are *lifted*: the rule for $x'(p)$ is written once over a player variable p, then grounded for p_1. The rules for `shape`$'(k)$ and `locked`$'(d)$ are likewise written over key and door variables. A learned template can therefore be reused for compatible objects in later levels.

B.2.2 LOCALITY AND CPF PARENT ERROR

Technically, a CPF may condition on the complete state:

$$X_{t+1}(\bar{o}) = f_X(\bar{o}; s_t, a_t). \tag{18}$$

Equation 18 is formally valid, but it does not identify which parts of the state support reuse or diagnosis. A useful model identifies a compact parent assignment for each next fluent:

$$u_t^X(\bar{o}) \subseteq s_t \cup a_t \cup \mathcal{N} \cup \mathcal{I}_t. \tag{19}$$

Equation 19 restricts this assignment to values available from the current state, action, non-fluents, and derived functions. For the player's horizontal coordinate, a sufficient context contains the current

position, the action, and the binary wall test in the grounded transition example. A mixed empirical row can therefore indicate that the proposed context is too coarse. If this test is omitted, players governed by the same lifted template can show both movement and no-change effects. Adding the test splits those effects into predictable contexts.

This is the RDDL interpretation of a missing-parent error. The type and action may be correct, while the context supplied to the CPF omits a fluent or derived relation that changes the outcome. The observable ledger and declared-fluent registry test this explanation by measuring whether a candidate parent makes the resulting rows more predictable.

Running example: Mixed rows mean missing CPF parents

Suppose the row only indexes

$$j = (\texttt{player}, \texttt{right}, u_0)$$

and records

$$C(j, \texttt{x_change}) = 6, \qquad C(j, \texttt{no_change}) = 4.$$

This need not mean that movement is stochastic (and we assume no stochasticity). The context u_0 may be missing $\texttt{blocked_right}(p)$. Adding that parent gives

$$\begin{aligned} u = \texttt{open_right}: &\quad C(\texttt{x_change}) = 6, &&\text{move succeeds,} \\ u = \texttt{wall_right}: &\quad C(\texttt{no_change}) = 4, &&\text{move is blocked.} \end{aligned}$$

The two contexts now predict separate outcomes, as expected from a deterministic movement CPF.

B.2.3 CPF BODY AND ACTION-SEMANTIC ERROR

Even with the correct local parents, the CPF body may still be wrong. This part of world-model learning identifies the action semantics and the update equation. For the simplified `ls20` example, a learner may correctly identify position, action 4, and the wall test as parents, but it must still infer that an open destination adds one to x and a blocked destination leaves x unchanged.

The executed input a_t is observed, but its effect is not known in advance. In RDDL terms, the learner knows which action assignment occurred, but it must infer which next fluents depend on that action and how. If the parent set is adequate but an exact next value is wrong, the error lies in the CPF body rather than the parent selection. CEGIS tests this distinction because it compares the values predicted by the executable body, while the effect diagnostic records a coarser summary.

Running example: Learning the CPF body

Suppose the learner has correctly identified

$$\mathrm{Pa}_x(p) = \{x_t(p), y_t(p), \texttt{move_right}_t(p), \texttt{blocked_right}_t(p)\}.$$

It must still learn whether action 4 changes x by $+1$, by -1, or not at all. The correct deterministic body is

$$f_x(p; \mathrm{Pa}_x) = \begin{cases} x_t(p) + 1, & \texttt{move_right}_t(p) \wedge \neg\texttt{blocked_right}_t(p), \\ x_t(p), & \text{otherwise.} \end{cases}$$

B.2.4 LIFTEDNESS AND TYPING ERROR

A CPF is written with variables and then grounded to objects. The lifted template

$$x'(p) = f_x(p; s, a) \tag{20}$$

applies to each compatible player p. More generally, a property update may be shared by a type or concept τ:

$$b'(o) = f_{\tau,b}(o; s, a), \qquad \tau(o) = \tau. \tag{21}$$

Typing defines which objects share a lifted CPF template. If types are known, they are declared RDDL object categories, called sorts. If types are uncertain, $\tau(o)$ may be treated as an attribute or learned concept that selects the update rule.

Liftedness creates a separate error mode. If a template is shared across objects that should have different dynamics, one empirical row can mix incompatible effects even when the local context is

adequate. For example, a moving block and a static wall should not share a position-update template. The repair is to split or revise the type used to select the CPF template. Parent error and typing error can initially produce the same mixed row. The system therefore tests candidate contexts and verifies the resulting program by replay before accepting a structural revision.

Running example: Lifted reuse

If the agent learns that a player moves right when the destination is open, it is represented as a lifted rule:

$$x'(p) = x(p) + 1 \quad \text{if} \quad \texttt{move_right}(p) \wedge \neg \texttt{blocked_right}(p).$$

The same rule can be grounded on each later object assigned the player type.

B.2.5 Grounding gives a factored dynamic model

Once a lifted RDDL model is grounded over a finite object set, each grounded fluent becomes a node in a two-slice dynamic Bayesian network, a graph connecting current-state variables to next-state variables. Grounding the CPFs gives the factored transition

$$P(s_{t+1} \mid s_t, a_t) = \prod_{X \in \mathcal{X}} \prod_{\bar{o} \in \mathrm{Ground}(X)} P(X_{t+1}(\bar{o}) \mid \mathrm{Pa}_X(\bar{o}; s_t, a_t)) \,. \tag{22}$$

Each factor is a point mass in the deterministic experiments.

Current fluents, the action, and fixed instance facts determine intermediate fluents and a local parent assignment. The type selects a shared CPF template, which produces one next-state fluent. Grounding this pattern for each legal object and fluent yields Equation 22. The synthesized Python evaluates these factors together inside `transition_function`.

B.3 Effect signatures

The exact CPF context depends on a fluent family b. A detailed diagnostic could therefore use

$$j_t^{i,b} = (\tau_t^i, b, a_t, c_t^{i,b}). \tag{23}$$

Equation 23 gives the fluent-specific row suggested by the CPF formalism. Borrowing from past MDP literature, a row in this case is simply a unique permutation of its contents. The implementation uses the coarser object row $j_t^i = (\tau_t^i, a_t, c_t^i)$ and groups changed fluent families into one effect signature. This approximation reduces the number of rows and applies the same diagnostic to births, disappearances, movement, visibility, rotation, and pixel changes. The baseline c_t^i is the sorted list of visible-neighbor types within Chebyshev radius r_c. Values from committed ξ features are appended to this list. Python synthesis receives the complete recorded states and may introduce dependencies that are absent from j_t^i. The verifier then evaluates their full predicted values.

For a paired object i, define

$$\Delta_t^i = \{b : b_t(o_i) \neq b_{t+1}(o_i)\}, \qquad e_t^i = \rho(\Delta_t^i) \in \mathcal{E}. \tag{24}$$

Equation 24 maps the set of changed fields Δ_t^i to the categorical effect e_t^i. The implementation excludes size, display-coordinate, name, and tag metadata before constructing Δ_t^i. It sorts the remaining field names and adds `no_change`, `born`, and `gone` categories. Object matching first uses name and position and then pairs remaining records with the same name.

Running example: Effect signatures are diagnostic

Suppose a row contains

$$C_t(j, \texttt{x}) = 6, \qquad C_t(j, \texttt{no_change}) = 4.$$

In a deterministic domain, these counts are consistent with an omitted parent such as $\texttt{blocked_right}(p)$ or with a template that groups incompatible objects. Adding the blocker produces an open-space row with movement and a wall row with no movement. The signature cannot determine the increment or destination because it records which fluent families changed. CEGIS tests those values.

B.4 BAYESIAN UNCERTAINTY AND THE EXPLORATION MATRIX

The belief-state example in Section 3.1.1 is a schematic abstraction. Its displayed property and relation probabilities summarize uncertainty whose underlying attribution is lifted over objects, actions, contexts, and observed effects. The implementation records evidence for these effect classes and grounds model-declared fluents on applicable objects; the illustration omits this indexing and uses illustrative values rather than literal estimator outputs.

Let $C_t(j, e)$ count signatures in row j and let $n_j = \sum_e C_t(j, e)$. A repeated (j, e) pair contributes at most once per transition, which limits multiplicity from repeated instances. The global alphabet size is

$$m = \max\left(m_{\min}, \left|\bigcup_j \{e : C_t(j, e) > 0\}\right|\right). \tag{25}$$

Equation 25 counts effect labels observed in any row and applies the lower bound $m_{\min}$, which keeps the normalized entropy defined before the ledger has observed multiple labels. For $q_j(e) = P(e_t^i = e \mid j)$, we place a symmetric Dirichlet prior on the categorical effect probabilities and update it with the recorded counts (Gelman et al., 2013):

$$\begin{aligned} q_j &\sim \mathrm{Dir}(\alpha_0 \mathbf{1}_m), \\ q_j \mid \mathcal{D}_t &\sim \mathrm{Dir}(\alpha_0 \mathbf{1}_m + \mathbf{c}_j), \\ \widehat{q}_j(e) &= \frac{\alpha_0 + C_t(j, e)}{m\alpha_0 + n_j}. \end{aligned} \tag{26}$$

In Equation 26, q_j is the row's vector of effect probabilities, $\mathbf{c}_j$ is its count vector, $\mathbf{1}_m$ is a vector of m ones, and Dir denotes the Dirichlet distribution. The first line states the symmetric prior, the second updates it with recorded counts, and the third gives the probability used by the diagnostic. Unobserved effect categories in a row retain predictive mass $\alpha_0/(m\alpha_0 + n_j)$. The normalized row and aggregate uncertainties use Shannon entropy (Shannon, 1948):

$$\eta_j = \frac{\mathrm{H}(\widehat{q}_j)}{\log m}, \qquad \eta_t = \frac{\sum_j n_j \eta_j}{\sum_j n_j}. \tag{27}$$

Equation 27 defines uncertainty for a row and its count-weighted aggregate over the ledger. Archived diagnostics label η_t as ontology error. We use aggregate uncertainty because the statistic also responds to missing parents, extraction errors, and hidden state. The diagnostic report orders rows by $n_j \eta_j$. A row can have high entropy because its evidence is sparse or because its recorded outcomes remain mixed. Multiplication by n_j reduces the priority of small rows and retains well-supported mixed rows. Neither entropy nor this ordering determines the cause. A proposed refinement is required to test a missing parent or an object type that combines distinct dynamics. Object-pairing error and hidden state can produce the same observations without being resolved by such a refinement.

The implementation also evaluates a finite set $\mathcal{C}_t$ of type merges and composite-object candidates. If $\mathcal{R}$ denotes the current type assignment, it reports

$$\eta_t^\star = \min_{\mathcal{R}' \in \{\mathcal{R}\} \cup \mathcal{C}_t} \eta_t(\mathcal{R}'), \qquad \Delta\eta_t^\star = \eta_t(\mathcal{R}) - \eta_t^\star. \tag{28}$$

Equation 28 selects the lowest aggregate uncertainty among the current representation and the enumerated candidates. The reduction $\Delta\eta_t^\star$ can underestimate the reduction available from a representation that the candidate generator did not enumerate.

Actor-facing guidance has two parts. The first we consider an "exploration matrix", which has one entry for each object type and action (τ, a). For an entry with $n_{\tau,a}$ observations, of which $s_{\tau,a}$ show the most common effect, we place a uniform prior on the probability of the most common effect and update it with these counts. This gives a $\mathrm{Beta}(1 + s_{\tau,a},\ 1 + n_{\tau,a} - s_{\tau,a})$ posterior. Entries are ordered by one minus a sample from this posterior. The matrix shows the actor which object–action effects are under-sampled or still mixed, including those whose confounder is unknown. This points towards objects of high complexity.

The second part tells the actor which mixed effects the model accounts for. An admitted model-declared fluent $\zeta(o, s)$ refines a mixed action–context class into bins indexed by its value v, with

evidence shared across objects of the same type within each action–context–value bin. For a refined bin with $n_{j,v}$ observations and modal-outcome count $s_{j,v}$, its uncertainty is

$$\text{Refined-bin uncertainty} = 1 - \frac{s_{j,v}}{n_{j,v} + 1}. \tag{29}$$

This value approaches zero after a few consistent observations, as expected in a deterministic environment. When the bins of a mixed entry have low uncertainty, the actor can treat that entry as understood and spend its actions elsewhere. Fluent values observed without interaction evidence are listed as untested, which gives the actor unknown states to probe. Together, the two parts direct exploration toward effects that are complex or poorly understood.

Running example: From effects to exploration priority

The open and blocked transitions produce

$$\begin{aligned} j_{\text{open}} &= (\texttt{player}, 4, c_{\text{open}}), \quad & e_{\text{open}} &= \texttt{x_change}, \\ j_{\text{wall}} &= (\texttt{player}, 4, c_{\text{wall}}), \quad & e_{\text{wall}} &= \texttt{no_change}. \end{aligned}$$

Once the fluent `blocked_right` is admitted, open and blocked contexts form separate bins. With n observations of the same effect in either bin, Equation 29 gives uncertainty $1/(n+1) \to 0$. Once both bins have low uncertainty, the actor treats the mixed $(\texttt{player}, 4)$ entry as understood and explores elsewhere.

B.5 Context refinements and declared fluents

The ξ ledger scores the K_ξ rows with largest $n_j \eta_j$. Candidate features include varying target fields, a bounded target-pixel hash, neighbor types at observed offsets, neighbor multisets at radii in $\mathcal{R}_\xi$, and within-object click offsets for the pointer action. Generic target-field candidates exclude position, raw pixels, collision flags, and display metadata because those values can create instance-specific partitions.

For candidate ξ, write j_v for the subrow with $\xi = v$. The gate is

$$\begin{aligned} \eta_{j|\xi} &= \sum_v \frac{n_{j_v}}{n_j} \eta_{j_v}, \qquad \Delta\eta(j,\xi) = \eta_j - \eta_{j|\xi}, \\ \Delta\eta(j,\xi) \geq \delta_\xi, &\qquad \exists v : \; n_{j_v} \geq n_\xi \;\wedge\; \max_e C_t(j_v, e)/n_{j_v} \geq p_\xi. \end{aligned} \tag{30}$$

The threshold δ_ξ requires a minimum uncertainty reduction. The support threshold n_ξ requires enough samples in at least one subrow, and p_ξ requires that subrow to have a sufficiently common effect. The ledger supplies the candidate features and their statistics to the synthesizer. If the synthesizer proposes a feature, the engine re-evaluates Equation 30 on $\mathcal{D}_t$ before appending the feature value to later contexts.

Running example: Testing the missing parent

The candidate $\xi(p) = \texttt{blocked_right}(p)$ divides the mixed row into

$$\begin{aligned} \xi(p) = \text{false}: &\quad x_t(p) = 2 \to x_{t+1}(p) = 3, \\ \xi(p) = \text{true}: &\quad x_t(p) = 2 \to x_{t+1}(p) = 2. \end{aligned}$$

The gate in Equation 30 tests whether this split lowers effect uncertainty and has sufficient support. CEGIS still checks the predicted coordinates.

A model-declared fluent $\zeta(o, s)$ is an additional state property proposed by the synthesized model as a Python function of object o and state s. It may use one or both arguments and must return a value that can be used as a table key. Returning `None` or raising for an object marks the property as inapplicable. The registry accepts at most N_ζ functions with at most V_ζ observed values each. For this audit, the outcome signature augments e with horizontal and vertical displacement, rotation and visibility changes, and the pixel-hash transition. An object is engaged when it changes, receives a click, or lies within Chebyshev radius r_e of a changed object. A source action-context cell is mixed when it has at least n_{mix} samples, at least B_{mix} outcome signatures, and its most common signature occupies less than fraction p_{mix}.

The engine admits a fluent for a type and cell when it produces at least B_ζ value bins and each supported bin has at least n_ζ samples with modal fraction at least p_ζ. Values observed outside engaged samples are reported as requiring further interaction evidence. The registry supplies admitted fluents to the synthesizer as empirically supported state distinctions and refines the actor-facing classes as described in Appendix B.4.

B.6 VERIFIER COMPARATOR

The comparator V in Equation 7 selects which predicted and recorded state fields replay compares. In runs that learn from rendered observations, V compares the predicted and observed palette value at each retained cell. In runs supplied with object records, V compares dynamic objects by identity, position, visibility, rotation, and internal pixels.

Some games display a timer whose rendering changes after each action but whose quantization does not expose a usable state variable. A synthesized model may propose a mask for this display. The verifier accepts the mask only when it is a continuous horizontal or vertical strip with width at most two cells and area at most twice its length. In rendered-observation runs, masked cells are excluded. In object-record runs, an object is excluded only when its display rectangle lies inside the accepted strip. An invalid mask is ignored. This is the qualification on CEGIS in Equation 7.

The loader statically rejects references to the file containing $\mathcal{D}$. It permits level-entry caches for reset and level-advance transitions, because those entry states are selected by the environment. The reward function may not read these caches. Transition and reward outputs must also match the recorded successor, reward, and termination flag in $\mathcal{D}$.

B.7 SYNTHESIS, REPLAY, AND CRITICISM

The CEGIS procedure (Solar-Lezama et al., 2006; Jha et al., 2010; Alur et al., 2013) receives the replay buffer $\mathcal{D}_t$, preceding Python model, qualitative hypotheses $(\widehat{\mathcal{S}}_{\mathrm{NL}}, \widehat{T}_{\mathrm{NL}}, \widehat{R}_{\mathrm{NL}})$, and available critic findings. It may revise the extractor, transition and reward functions, planner, and qualitative hypotheses. Candidate programs are evaluated on the accumulated replay buffer; failures provide counterexamples for the next revision. Equation 7 defines acceptance on observed transitions.

On scheduled synthesis rounds, an independent critic reviews replay-consistent code for layout-specific constants, omitted interactions, and unsupported generalization. Valid findings prompt repairs followed by another replay check; findings that require new observations are retained as unresolved hypotheses and proposed probes. Replay establishes consistency with the observed transitions, while predictions for unvisited states remain hypotheses.

The synthesis interface specifies object records and declared fluents while leaving code organization flexible. Coupled effects may be implemented jointly across objects; a mechanic need not belong to a single object's method.

For $\widehat{s}_{\ell+1} = \widehat{T}(s_\ell, a_\ell)$, the replay mismatch set is

$$\mathcal{M}_\ell = \{X(\bar{o}) : \widehat{s}_{\ell+1}[X(\bar{o})] \neq s_{\ell+1}[X(\bar{o})]\}. \tag{31}$$

Equation 31 identifies the CPF families involved in a replay failure. The static checks described in Section B.6 reject replay lookup and reward functions that use cached future states.

Running example: Counterexample localization

If the only mismatch is

$$\widehat{x}_{t+1}(p_1) \neq x_{t+1}(p_1),$$

then the first repair target is the CPF for $x'(p)$ rather than the complete transition function. The synthesizer can test whether a parent such as `blocked_right`(p) is missing or whether the update body is wrong.

Checked planning. Once $\mathcal{D}$ contains reward evidence, the planner searches for a sequence that satisfies

$$\widehat{s}_{k+1} = \widehat{T}(\widehat{s}_k, a_k), \qquad \max_{0 \le k < L} \widehat{R}(\widehat{s}_k, a_k, \widehat{s}_{k+1}) > 0. \tag{32}$$

Equation 32 requires the model rollout to contain positive predicted reward. Before live use, the harness verifies plans from stored initial states of completed levels. During live execution, it compares each observed successor with the model prediction. A disagreement adds a counterexample to $\mathcal{D}$ and terminates the remaining plan. This checked execution is the operational basis of goal-directed falsification: a plan reaches reward or returns evidence against the model that proposed it.

Running example: Goal hypothesis

In the simplified `ls20` example, a candidate goal requires an unlocked door and a player at that door:

$$G(s) = \mathbf{1}\{\exists p, d : \neg\texttt{locked}(d) \wedge \texttt{at_door}(p, d)\}\,.$$

The predicate is retained only when it agrees with the recorded reward transitions, just as a transition CPF must agree with replay.

B.8 Online learning algorithm

Algorithm 1 summarizes the interaction loop. Here $\mathrm{NL} = (\widehat{\mathcal{S}}_{\mathrm{NL}}, \widehat{T}_{\mathrm{NL}}, \widehat{R}_{\mathrm{NL}})$, and Synthesize includes the replay and scheduled critic procedure in Appendix B.7. RepairGate uses the batching limits in Appendix C. While repair is deferred, the actor collects additional evidence; reward-directed programmatic planning requires replay consistency and planner eligibility. Observation abstraction reconstructs any required history from the recorded trajectory, and ExploitGoal? includes the planner eligibility checks.

Algorithm 1 OPINE-World online learning loop (schematic)

Input: known action interface $\mathcal{A}$, action budget B
$\omega \leftarrow$ Env.Reset(); $\quad \mathcal{D} \leftarrow \varnothing$
$\mathrm{NL}, \widehat{\mathcal{S}}_{\mathrm{B}}, \widehat{T}_{\mathrm{P}}, \widehat{R}_{\mathrm{P}}, \widehat{P}, \widehat{f}^{-1} \leftarrow$ Initialize$(\omega, \mathcal{A})$
while budget remains and the game is unfinished **do**
  if RepairGate$(\mathcal{D})$ and $\neg\Phi_1(\widehat{f}^{-1}, \widehat{T}_{\mathrm{P}}, \widehat{R}_{\mathrm{P}}; \mathcal{D})$ **then**
    repeat
      $\mathrm{NL}, \widehat{T}_{\mathrm{P}}, \widehat{R}_{\mathrm{P}}, \widehat{P}, \widehat{f}^{-1} \leftarrow$ Synthesize$(\mathcal{D}, \mathrm{NL}, \widehat{\mathcal{S}}_{\mathrm{B}})$
    until $\Phi_1(\widehat{f}^{-1}, \widehat{T}_{\mathrm{P}}, \widehat{R}_{\mathrm{P}}; \mathcal{D})$
  end if
  Reconstruct $\widehat{s}$ from ω and recorded history using $\widehat{f}^{-1}$.
  $c \leftarrow \Phi_1(\widehat{f}^{-1}, \widehat{T}_{\mathrm{P}}, \widehat{R}_{\mathrm{P}}; \mathcal{D})$
  if c and ExploitGoal?$(\widehat{s}, \mathrm{NL}, \widehat{\mathcal{S}}_{\mathrm{B}})$ **then**
    $\pi \leftarrow \widehat{P}(\widehat{s}; \widehat{T}_{\mathrm{P}}, \widehat{R}_{\mathrm{P}})$
  else
    $\pi \leftarrow$ EpistemicExperiment$(\widehat{s}, \mathrm{NL}, \widehat{\mathcal{S}}_{\mathrm{B}})$
  end if
  for $a \in$ ShortPrefix(π), within the remaining action budget **do**
    $(\omega', r) \leftarrow$ Env.Step(a)
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\omega, a, r, \omega')\}$
    $\mathrm{NL}, \widehat{\mathcal{S}}_{\mathrm{B}} \leftarrow$ Update$(\mathcal{D}, \mathrm{NL}, \widehat{\mathcal{S}}_{\mathrm{B}})$
    $m \leftarrow \neg$LiveCheck$_{\Phi_1}(\omega, a, r, \omega')$
    $\omega \leftarrow \omega'$
    if m **then**
      discard the unexecuted suffix of π; **break**
    end if
  end for
end while

C Experimental Details

The studies use different sources of replication, which makes the experimental unit important for interpreting their uncertainty estimates. Table 4 summarizes the unit, evidence, and intervention for each study. All language-model roles use Claude Opus 4.8 with a one-million-token context window.

The main ablations in RQ1 use high effort, while RQ2, RQ3 ablations use low effort. Runs use Docker filesystem and network isolation to prevent cheating.

The online loop uses $K_{\text{critic}} = 3$. After the first prediction failure, model review may be deferred for at most $D_{\text{move}} = 5$ executed moves, $D_{\text{error}} = 3$ accumulated errors, or $D_{\text{plan}} = 3$ completed action plans. These constants are fixed across online conditions. The planner becomes eligible after the first completed level.

Table 4: **Experimental configurations.** The unit column gives the independent or paired unit used in summaries and tests. Learned extraction infers object records from rendered frames.

Study	Unit	Evidence and conditions
RQ1: exploration	25 paired games	The ablation removes $\widehat{\mathcal{S}}_{\text{B}}$.
ϵ-greedy	25 shared games	Descriptive comparison at a 400-action cutoff. The baseline removes $\widehat{\mathcal{S}}_{\text{B}}$ and explores random objects with probability $\epsilon = 0.1$.
RQ2: fixed evidence	5 runs per condition	The conditions receive the same 757-action `g50t` trajectory in batches of six. The actor and planner are disabled. Conditions are the full system, monolithic and type-restricted programs, no $\widehat{\mathcal{S}}_{\text{B}}$, and no persistent NL.
RQ2: revision	10 repairs per condition for M1 and M3	M1 and M3 start from independently synthesized source programs; within each comparison, the 250-action repair trajectory is matched across conditions. M2 measures six level transitions within each of the five fixed-evidence runs.
RQ3: online	5 runs per condition	The full system and the no-persistent-language condition learn the first three `g50t` levels with a 1,000-action cap. The ablation rejects prose, comments, and docstrings from the persisted world model.

C.1 Scope and interpretation of the ablations

Exploration as a capability of the whole system. In an exploration task, a working planner enables the agent to plan efficient routes to informative states, programmatic replay verifies that its understanding of the world is consistent with the observed history, object extraction allows principled tracking of the world state, plan soundness determines exploitation or exploration priorities. The capability of the learned engine therefore can be classified as an exploration capability. This is why RQ1 evaluates the contribution of $\widehat{\mathcal{S}}_{\text{B}}$ through this synthesis, verification, planning, and action-selection loop in tandem.

Limiting Controls. For any experiment, it is possible to generate an almost unlimited list of potential confounders. When designing these experiments, the issue we faced is not whether a variable exists, but whether it is sufficiently important and plausible to threaten a conclusion. In RQ1, improvements to the learned engine and the resulting plans are mechanisms through which Bayesian guidance can improve exploration. In RQ2, different interaction evidence could itself explain a difference in model quality, so the trajectory is held fixed across conditions.

To assess the world-modeling capabilities targeted by RQ2 and RQ3, we evaluate seven recurring challenges: (1) partial observability, (2) history-conditioned state and dynamics, (3) autonomously moving agents, (4) coupled interactions among multiple objects, (5) context-dependent behavior, (6) intermediate states between actions, and (7) the introduction of new objects and mechanics as tasks progress.

RQ2 and RQ3 use `g50t` as a complete case study of these world-modeling challenges. These properties make `g50t` a demanding environment for testing both executable model structure and persistent qualitative hypotheses. These modeling requirements generalize beyond this single domain's isolated visual layout. In addition, the large discovery failure under the Bayesian ablation is informative for examining mechanisms behind the benchmark gains. Each of these runs is prohibitively expensive, and the traces we analyzed under this general case resulted in convincing enough behaviors to motivate this approach.

C.2 Statistical analysis

The tests follow the unit that generated the variation. Actions, levels, and synthesis rounds within a run are not treated as independent observations. All reported tests are two-sided, and the reported p-values are unadjusted.

RQ1 pairs conditions within each game. Per-game RHAE uses the Wilcoxon signed-rank test. Level completion first reduces each game to its paired difference in completed levels and then applies a sign-flip randomization test. The paired cluster bootstrap resamples 25 games with replacement 100,000 times. These tests compare the high-effort full and ablated conditions.

The fixed-evidence and M1 analyses match conditions by run index. Their exact sign-flip tests enumerate the 2^n signs of the paired differences and count assignments whose absolute mean is at least the observed value. M3 and the online RQ3 study use exact two-sample permutation tests over the independent synthesis replications. These tests enumerate assignments of the pooled observations to conditions while preserving the number of runs per condition. M2 is summarized at the level-transition unit, with transitions grouped within their evolving run. We report M2 descriptively because the type-restricted condition has missing intermediate snapshots and the transitions within a run are dependent.

Table 15 reports the completed RQ3 online analysis. Its p-values come from the $\binom{10}{5} = 252$ assignments of five runs to each condition. Repair-test values are reported with the repair results in Appendix F.7.

C.3 Held-out prediction and program classification

Replay accuracy on synthesis evidence cannot measure generalization. The structure study therefore uses a second buffer that is unavailable during synthesis. A seeded random policy generates randomly seeded 100,000 transitions. This buffer is shared by all conditions. The evaluator replays each trajectory, and supplies the learned engine with the ground-truth before-state at each step. Exact match requires its predicted successor to match all object records in the ground-truth successor. This can be called a test set.

A prompt restriction also does not guarantee that synthesis produces the intended source organization. For function k, let c_k be cyclomatic complexity, ℓ_k be source lines, d_k be decision sites, and h_k be maximum nesting depth. Let $c_{(k)}$ denote complexities sorted from largest to smallest. We classify a program P with

$$\begin{aligned} C_{\max}(P) &= \max_k c_k, & C_{95}(P) &= \mathrm{P95}_{\ell_k}(c_k), \\ H_d(P) &= \sum_k \left(\frac{d_k}{\sum_j d_j} \right)^2, & D_{\max}(P) &= \max_k h_k, \\ C_5(P) &= \sum_{k=1}^{5} c_{(k)}. \end{aligned} \tag{33}$$

Radon 6.0.1 computes per-function source lines and McCabe cyclomatic complexity, which counts independent control-flow paths (McCabe, 1976; Lacchia, 2023). $C_{\max}$ identifies the largest function, while C_{95} is the source-line-weighted 95th percentile and C_5 sums the five largest function complexities. H_d applies the Herfindahl–Hirschman index to each function's share of decision sites (Hirschman, 1964). It approaches one when the same function contains most decisions. $D_{\max}$ is the largest Python block-nesting depth computed from the abstract syntax tree. Together, these measurements classify monolithicity. Held-out exact match and revision cost assess model outcomes. We also archive maximum cognitive complexity as an auxiliary source-complexity metric (Campbell, 2018).

The online study learns object records from rendered frames based on exploratory decisions, whereas the controlled synthesis studies provide a fixed action trajectory. The evidence, verifier, synthesis schedule, and evaluation buffers remain fixed across conditions.

Holding the trajectory fixed and disabling the actor and planner in RQ2 isolates the influence of $\widehat{S}_{\mathrm{B}}$ through synthesis, including the induced program structure and learned transition rules. Online,

the synthesized engine also supports action selection, so exploration outcomes reflect the coupled synthesis and exploration loop. RQ2 evaluates synthesis separately under controlled evidence.

The reported diagnostic uses $\alpha_0 = 1$, which gives each effect one symmetric pseudocount. We set $m_{\min} = 2$ so that normalized entropy is defined before two effects have been observed. The base and engagement neighborhoods use $r_c = r_e = 1$ because immediate contact captures the local interactions used by the actor while limiting row fragmentation.

The prompt budget admits $K_\xi = 3$ high-weight rows, and $\mathcal{R}_\xi = \{2, 3\}$ tests nearby relational context beyond the base neighborhood. The observable-refinement gate uses $\delta_\xi = 0.05$, $n_\xi = 3$, and $p_\xi = 0.95$. The declared-fluent registry uses $N_\zeta = 16$, $V_\zeta = 12$, $n_{\text{mix}} = 3$, $B_{\text{mix}} = 2$, $p_{\text{mix}} = 0.9$, $B_\zeta = 2$, $n_\zeta = 2$, and $p_\zeta = 0.9$. The capacity limits bound prompt growth. The support and concentration thresholds require repeated evidence before a state distinction is admitted. Equation 30 and Section B.5 define how these constants are used.

D Extended Related Work

Learning World Models for Planning. Planning in unfamiliar environments requires learning action consequences from limited interaction, while the relevant state variables and goals may also be unknown. Latent methods learn compressed dynamics for rollouts and control (Ha & Schmidhuber, 2018; Hafner et al., 2025). Programmatic methods synthesize executable dynamics for prediction and search (Tang et al., 2024; Ahmed et al., 2025; Piriyakulkij et al., 2025). Language-augmented methods retain NL theories that evolve with observed transitions (Levy et al., 2025), and NL-PDDL plans over a supplied NL action model via lifted regression with LLM entailment (Liu et al., 2026), but does not learn that model from interaction. Yet fitting observed transitions does not establish whether the representation captures distinctions needed for untried actions. Existing approaches also do not unify NL and programmatic models in a way that provides an actionable measure of uncertainty of the learned representation. Additionally, planning over high branching factors becomes computationally prohibitive at long horizons with a general planner, motivating task-specific planners and heuristics.

Exploration and State Discovery. Discovering these omissions requires interactions that distinguish competing explanations of action effects. Information-based methods target uncertainty about model parameters (Lindley, 1956; Houthooft et al., 2016). Surprise-based methods use prediction error or disagreement among transition models (Pathak et al., 2017; Sekar et al., 2020). State-discovery methods add history features to resolve perceptual aliasing (McCallum, 1996) or learn dependencies among supplied variables (Strehl et al., 2007). Yet uncertainty measured within a fixed representation does not by itself identify which missing object property or relation would explain conflicting outcomes. Open-world exploration must therefore also generate and test hypotheses about the representation itself.

ARC-AGI-3 Systems. Recent ARC-AGI-3 systems demonstrate the value of persistent test-time artifacts and structured interaction. Rudakov et al. (2025) heuristically prioritize and enumerate untested frame–action pairs, navigating through recorded transitions to previously observed frontier states. This improves coverage over random exploration but does not abstract equivalent states or prioritize experiments according to their expected information about the underlying mechanics. Its exploration cost consequently scales with the reachable visual state space and depends on frontier states remaining recoverable. Other systems use replay-tested executable models (Rodionov, 2026), multi-agent workspaces (Sarafian et al., 2026), and continual programmatic harnesses (Karten et al., 2026; Furgale et al., 2026). These approaches provide increasingly capable mechanisms for retaining experience, constructing models, and coordinating test-time reasoning, but lack formal approaches to efficient, information-informed exploration.

Our Point of Departure. We introduce a hybrid NL, Bayesian, and programmatic world model that unifies representation revision with targeted exploration. An object-oriented NL proto-representation preserves partial hypotheses, while a Bayesian belief state prioritizes sparsely tested or conflicting object–action effects. The resulting observations support revision of both the proto-representation and executable model, which supplies predictions and reward-directed planning.

E LIMITATIONS

Domain assumptions. ARC-AGI-3 environments are deterministic and use discrete state and action spaces. Accordingly, our Bayesian diagnostic interprets mixed effects as evidence that the learned ontology omits a relevant property, relation, or historical variable. In stochastic environments, mixed effects may instead reflect irreducible transition uncertainty. Extending OPINE-World to stochastic or continuous-time, state, or action spaces would therefore require distributional transition predictions, probabilistic replay consistency, and uncertainty estimates. This is beyond the scope of our system.

Unvisited states. OPINE-World relies on replay consistency and observed effects; therefore, Φ_1 gives no guarantee for unvisited states. While our systematic hypothesis generation and goal-conditioned experiment design could prioritize untested objects that may be necessary for reward, a well-principled system for this failure mode based on goal-conditioned relevance, coverage, and structured hypothesis generation would allow better insights into unexplored states.

Computational cost and evaluation coverage. A complete 25-game evaluation uses Claude Opus 4.8 at high reasoning effort for acting and program synthesis. Including replay and planning, individual environments can consume up to 2 billion cumulative tokens. Due to this cost, we allocated the available compute to maximize benchmark coverage in areas that are pivotal for our claims.

F ADDITIONAL RESULTS AND ARTIFACT AUDITS

The paired game audit links exploration failures to the mechanics left unresolved. The fixed-evidence and revision studies compare how the resulting programs predict and change, while the NL ablation tests the value of retaining explanations between synthesis sessions. Examples are grouped by the finding they support, with the measurements and source excerpts reported together.

F.1 COMPLETE-SYSTEM BENCHMARK COMPARISON

OPINE-World's aggregate advantage includes completing games where the strongest comparator entirely fails (Table 5). The comparison uses matching game builds and scoring rules, including the baseline1 scorecard rather than the earlier run reported in its paper (Rodionov, 2026; Sarafian et al., 2026; Karten et al., 2026; ARC Prize Foundation, 2026c). The per-game results and coverage counts show why both completion and action efficiency matter when interpreting the aggregate score.

Table 5: **Per-game RHAE and aggregate coverage.** OPINE-World, the $\widehat{\mathcal{S}}_{\mathrm{B}}$ ablation, and other systems. Standard errors are computed across the 25 games.

	OPINE-World		Other systems			
Game	Full	No $\widehat{\mathcal{S}}_{\mathrm{B}}$	baseline1	DreamTeam	Continual	OpenClaw
ar25	**100.0**	**100.0**	**100.0**	**100.0**	17.4	8.3
bp35	2.6	2.0	**17.3**	0.8	1.1	0.4
cd82	**100.0**	**100.0**	**100.0**	79.0	0.0	0.0
cn04	**100.0**	47.6	**100.0**	0.0	46.4	0.0
dc22	**82.4**	14.3	36.4	0.0	18.2	4.8
ft09	**100.0**	**100.0**	95.1	**100.0**	70.1	14.3
g50t	61.7	0.0	**100.0**	2.9	10.7	0.0
ka59	62.5	60.6	**63.0**	59.7	38.6	10.7
lf52	4.2	3.5	**34.9**	10.9	0.2	1.8
lp85	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	0.0
ls20	71.2	59.1	**98.3**	14.7	3.6	3.6
m0r0	**100.0**	**100.0**	71.6	47.6	66.8	14.3
r11l	**100.0**	**100.0**	74.3	47.6	4.8	0.0
re86	**100.0**	**100.0**	26.4	58.3	16.7	2.8
s5i5	27.3	7.5	0.7	**38.2**	0.0	2.8
sb26	89.8	**100.0**	**100.0**	94.0	0.0	2.8
sc25	**84.0**	28.6	20.1	36.1	0.0	14.3
sk48	21.3	16.7	**30.2**	6.9	1.1	2.8
sp80	**100.0**	66.0	5.5	0.1	47.6	4.8
su15	**92.1**	50.6	88.7	2.2	0.0	0.0
tn36	**68.4**	3.6	26.2	5.0	0.0	3.6
tr87	**100.0**	**100.0**	86.3	12.6	0.0	14.3
tu93	**100.0**	**100.0**	**100.0**	**100.0**	61.4	6.5
vc33	**92.4**	88.9	19.4	21.4	8.9	10.7
wa30	**100.0**	5.0	**100.0**	13.3	0.0	6.7
Mean $\pm$ SE	**78.4** $\pm$ **6.3**	58.2 $\pm$ 8.3	63.7 $\pm$ 7.4	38.1 $\pm$ 7.6	20.5 $\pm$ 5.7	5.2 $\pm$ 1.0
Levels completed	**160/183**	126/183	146/183	98/183	64/183	28/183
Games completed	**20/25**	12/25	14/25	6/25	3/25	0/25

F.2 ϵ-GREEDY BASELINE TEST

The ϵ-greedy baseline ablates the $\widehat{\mathcal{S}}_{\mathrm{B}}$ exploration layer and replaces each newly proposed action sequence with probability $\epsilon = 0.1$ of blind exploration towards a random object. We pick a 400-action budget cutoff for all comparisons due to budget constraints. The results show a clear conclusion even at this limit (Table 6 and Figure 6). Random exploration is insufficient to match the full system or even the $\widehat{\mathcal{S}}_{\mathrm{B}}$ ablation.

Table 6: **Exploration at 400 actions per game.** RHAE follows Appendix G.

Condition	RHAE	Levels cleared	Games solved
Full system	65.97	141/183	13/25
$\widehat{\mathcal{S}}_{\mathrm{B}}$ ablated	52.96	115/183	8/25
ϵ-greedy ($\epsilon = 0.1$)	39.49	106/183	7/25

F.3 PAIRED LEVEL PROGRESSION

Table 7: **Paired online comparison for RQ1.** Conditions are paired within each of the 25 games.

Measurement	Value
Full-system mean RHAE	78.40
Ablated mean RHAE	58.16
Mean paired RHAE difference	20.24
Standard deviation of paired differences	29.45
Paired standardized effect (d_z)	0.69
RHAE: exact p	4.27×10^{-4}
Full-system levels completed	160/183
Ablated levels completed	126/183
Additional levels completed	34
Completion-rate difference (percentage points)	18.6
Paired game-cluster bootstrap 95% CI (percentage points)	$[8.0, 31.0]$
Level completion: exact p	1.95×10^{-3}

A higher score can reflect either shorter solutions or access to later levels. The full system completes more levels than the $\widehat{\mathcal{S}}_{\mathrm{B}}$ epistemic ablation, with the difference extending beyond the early levels (Table 8). Bayesian guidance therefore improves coverage as well as the aggregate score. Its benefit cannot be explained solely by taking fewer actions on levels that both conditions solve.

Table 8: **Paired progression by level index.** RQ1 completion counts for all 25 games. Eligible gives the number of games containing that level. The final columns count discordant condition pairs. Level positions are sequentially dependent within a game and are not treated as independent samples.

Level	Eligible	Full	No $\widehat{\mathcal{S}}_{\mathrm{B}}$	Full only	Ablation only
1	25	25	24	1	0
2	25	25	23	2	0
3	25	24	20	4	0
4	25	23	16	7	0
5	25	21	15	6	0
6	25	21	14	7	0
7	16	11	8	3	0
8	11	7	5	2	0
9	5	3	1	2	0
10	1	0	0	0	0
Positions	183	160	126	34	0

The progression analysis compares completed-level counts within games because later levels depend on earlier completion. The significance test uses these paired game differences, and the confidence interval resamples games while preserving their level sequences (Table 7).

F.4 HOW ONLINE EXPLORATION FAILURES AROSE

The largest losses under the epistemic ablation occur when the agent fails to discover a required mechanic. It may continue rearranging familiar objects or repeating similar actions without discovering what prevents completion. This low-diversity exploration can be catastrophic because additional combinations of known actions do not reveal the missing interaction. The paired audit identifies where Bayesian guidance changes this behavior by directing attention toward sparsely modeled interactions (Table 9).

The audit examines action traces, hypotheses, synthesis handoffs, and observed frame changes across the paired runs. The failure categories distinguish repeated search over familiar mechanics, overlooked interactions, and failures to transfer previously learned behavior.

Table 9: **Hand-checked exploration behavior.** Patterns found in the online ablation artifacts. The final column summarizes the evidence recorded in the corresponding interaction traces.

Pattern	Games	Archived observation
Repeated or narrow search	su15, g50t, dc22, sk48	The actor repeats similar action sequences or block configurations while unresolved objects and interactions remain untested.
Attention remains on familiar objects	wa30, ar25, m0r0	The actor continues to probe a small set of known objects or mechanics after they stop producing progress.
Untested or misread mechanic	cn04, sc25, tn36	The actor overlooks essential blatantly presented frame information unimpeded.
Missing class-level record	s5i5, ls20	The actor does not use previous tests of the same object class to select a different interaction or transfer a behavioral property.
Incorrect carry-forward model	sp80	Limited probing in earlier levels leaves the transition program and planner with an incorrect mechanic in a later level.

Listing 1: Example 1. Failed docking arrangements in the epistemic ablation of cn04.

```
CONFIGS TRIED, ALL reward=0/done=False:
2. foot-on-body assembly w/ body-overlap: placed (messy pile), NO reward.
3. 7==7 GROW overlay: grew YELLOW to 7 feet, moved G1/G2/G3 so all 7 target feet coincide
    EXACTLY with YELLOW 7 feet (verified pixel-exact) -> gray3 all, NO reward.
```

Listing 2: Example 2. Further docking hypotheses in the same world model.

```
CALL66: 7==7 overlay WITH YELLOW ACTIVE at end -> val3=63 (all 7 YELLOW feet gray3 docked), NO
     REWARD.
Only untried class: CLEAN rotation-only NON-OVERLAPPING interlock (like L4 spirals).
```

Listing 3: Example 3. Later actor notes returning to the docking construction.

```
Step 928
Yellow-docker approach FALSIFIED: reached feet==all-7-comb-tips exactly (via click and via
    translate) with reward=0.
Step 929
Fresh board, yellow active grow0(3 feet).
Step 972
All 3 combs docked onto yellow 7 feet.
```

These traces show combinatorial exploration of familiar docking arrangements while the interaction required for reward remains unresolved. The agent varies the construction without finding a new mechanic, which can be catastrophic for completion when the existing hypothesis is wrong. This is the low-diversity behavior identified in the paired exploration audit. Bayesian guidance addresses it by directing attention toward sparsely modeled interactions instead of relying on further combinations of familiar actions.

The audit also includes cases where Bayesian guidance provides no clear benefit or introduces a cost. Seven pairs show no meaningful difference in the mechanics tested. In tr87, the agent investigates uncertainty unrelated to the goal, while the sb26 review attributes the ablation's advantage to a planner error following a context reset in the full run. Three pairs remain unclassified because their review notes do not give a reason. These cases distinguish useful discovery from additional exploration that consumes budget without resolving a requirement for progress.

A decrease in ontology error does not necessarily mean that the agent is closer to solving the level. Revised extraction or type assignments can regroup earlier observations and raise η, while a new level can introduce poorly sampled interactions. The uncertainty traces describe the model's unresolved effects. Progress depends on whether those effects matter for the goal. Traces for ar25, tr87, and vc33 begin after extraction stabilizes and describe that later part of learning.

F.5 Archived hypothesis-and-test episodes

The full-system traces show how unresolved mechanics become useful experiments. The agent tests a requirement that its current model cannot explain, incorporates the resulting observation, and uses

the revised mechanic in a subsequent plan. The examples below cover goal revision, interactions between objects, and the reuse of a mechanic on a later level.

Listing 4: Example 1. `sp80` full-system reasoning log.

```
PLAN from step72 [barX inactive@32-51, paddle active@40-63]: [click(40,29) activate barX, 1 (
    up ->r24-27), 5 fire]. Predict all 3 cups fill.
## CALL23 (step74 fire, PNG-decoded by subagent): ALL 3 CUPS FILLED SIMULTANEOUSLY (color13,
    ticks18-24) yet REWARD 0.
=> 'all cups filled' is NOT the win condition. HUD row63 = 56 cells left (~84 moves) => budget
     NOT the issue.
```

Listing 5: Example 2. `vc33` full-system reasoning log.

```
- ROOT CAUSE: `_l7_band_cross` only ferried ONE marker (left-side preferred),
  clearing it and IGNORING the far-side marker. When BOTH flanking reservoirs
  hold a marker the env SWAPS them: each glyph crosses to the OTHER reservoir,
  re-centred horizontally at its own rows. Because the two glyphs are congruent
  and at the same rows here, the swap reduces to a bar-COLOUR exchange in place
```

Listing 6: Example 3. `dc22` full-system reasoning log.

```
- CONVEYOR CLICK LOCK (steps 681,690 are NET NO-OPS, before==after): when GREEN
  is riding a navy/teal bar and advancing that bar +2px (or wrapping it from the
  dock) would leave green OFF the bar's new span, the env REFUSES THE WHOLE click
  (every lane frozen) -- the same toggle-locked-under-player rule the L4/L5 navy
  cycles obey. Discrimination vs the FIRING clicks (e.g. step 680: green at c18-19
  on a leftmost-14 bar -> bar slides to span c16-21, green STILL on it -> fires).
```

Listing 7: Example 4. `sc25` full-system world model.

```
- TWO legends: DIAMOND/15 and MIDDLE-COL/6. RESOLVED -- BOTH are active, each
  with its own reaction: solving diamond/15 SHRINKS the handle (step 83); solving
  middle-col/6 fires a PROJECTILE (step 79 missed because the 4x4 straddled the
  arm). So the player must FIRST shrink (diamond) to fit the rows29-30 arm, THEN
  align + fire (middle-col) down the arm to clear the box+gate, THEN drive RIGHT
  into the dock. The CHECK reaction is keyed by legend color, NOT level.
```

These examples show discovery of new behavior rather than repeated combinations of familiar actions. A failed prediction or unsuccessful search leads to a test of the missing mechanic. Bayesian guidance supports this change in attention, while the persistent world model retains the discovered relationship for synthesis and planning. Together, these behaviors explain how exploration can recover progress after the current model stops providing a route to completion.

F.6 Program organization under fixed evidence

Monolithic Code Structure places more control flow in its largest functions than the other conditions (Table 10). The generated programs therefore differ in organization as intended by the interventions.

The online source survey finds that most programs combine centralized transition logic with object-related helpers. Their updates are commonly organized around a mechanic or level while their state remains object-indexed. This agrees with the RQ2 finding that coupled effects are difficult to assign to independent object rules. $\widehat{\mathcal{S}}_{\mathrm{B}}$ Code Structure allows synthesis to retain a shared update where the interaction requires it.

A short public transition function can delegate most of its work to a large helper. Measuring that entry point would therefore miss much of the program's control flow. The structure comparison measures the functions throughout the source, including helpers called by the public interface.

With the same trajectory supplied to synthesis, $\widehat{\mathcal{S}}_{\mathrm{B}}$ Code Structure has the highest mean held-out prediction accuracy. Type-Oriented Code Structure uses more output tokens without recovering that accuracy (Tables 11 and 12). A more prescribed organization therefore does not necessarily produce a better predictive model.

Table 10: **Program-structure classification.** Means and standard deviations over five fixed-trajectory runs per condition. The five diagnostics form the monolithicity classification in Equation 33. They verify that the prompt arms produce distinct program structures.

Condition	$C_{\max}$	C_{95}	H_d	$D_{\max}$	C_5
Full system	38.2 ± 10.2	37.8 ± 10.8	$.048 \pm .009$	5.0 ± 1.2	117.2 ± 16.3
Monolithic	257.0 ± 82.5	257.0 ± 82.5	$.476 \pm .127$	6.6 ± 0.9	337.6 ± 88.9
Type-restricted	49.2 ± 35.5	48.9 ± 35.6	$.056 \pm .039$	4.4 ± 0.5	126.0 ± 39.5
No $\widehat{\mathcal{S}}_{\mathrm{B}}$	51.8 ± 31.3	51.8 ± 31.3	$.070 \pm .039$	5.0 ± 0.7	134.0 ± 31.1
No persistent NL	40.6 ± 15.8	40.6 ± 15.8	$.045 \pm .018$	4.6 ± 0.9	110.0 ± 24.7

Table 11: **Fixed-evidence predictive and source measurements.** Means and standard deviations over five runs per condition. Exact match uses the shared 100,000-transition buffer. Registered state functions are model-declared candidates, while admitted functions pass a local effect-splitting check. Admission count does not measure the accuracy of the complete transition program. Maximum cognitive complexity is an auxiliary source diagnostic.

Condition	Exact match (%)	SLOC	Functions	Max cognitive	Registered	Admitted
Full system	83.9 ± 9.3	1004 ± 84	61.0 ± 7.2	61.6 ± 14.1	10.4 ± 3.4	0.8 ± 0.4
Monolithic	73.2 ± 16.5	915 ± 146	18.2 ± 4.3	549.4 ± 164.2	–	–
Type-restricted	64.4 ± 23.6	1107 ± 227	71.4 ± 7.7	64.8 ± 62.2	8.4 ± 3.6	0.8 ± 0.8
No $\widehat{\mathcal{S}}_{\mathrm{B}}$	71.6 ± 11.9	949 ± 124	48.4 ± 4.9	81.6 ± 40.6	–	–
No persistent NL	66.9 ± 18.5	709 ± 61	70.6 ± 7.6	63.0 ± 38.3	10.8 ± 3.3	2.4 ± 1.7

Table 12: **Fixed-evidence synthesis accounting.** Means and standard deviations over five runs per condition. Output is reported in thousands of tokens. Input and total are reported in millions of tokens.

Condition	Rounds	Output (k)	Input (M)	Total (M)
Full system	33.6 ± 5.7	741 ± 184	66.8 ± 16.6	67.5 ± 16.8
Monolithic	29.0 ± 3.9	631 ± 161	53.2 ± 15.3	53.9 ± 15.4
Type-restricted	34.8 ± 9.9	814 ± 272	70.7 ± 22.7	71.5 ± 22.9
No $\widehat{\mathcal{S}}_{\mathrm{B}}$	33.8 ± 5.0	789 ± 144	65.5 ± 10.6	66.3 ± 10.7
No persistent NL	32.0 ± 4.2	722 ± 193	64.4 ± 15.2	65.2 ± 15.3

No NL admits more candidate state predicates while achieving lower mean held-out prediction accuracy (Table 11). A predicate can separate observed effects locally without improving the model's predictions elsewhere. More admitted predicates should therefore not be read as better abstraction. Persistent language provides another way to retain a tentative distinction before synthesis commits it to a program rule.

Type-Oriented Code Structure can pass the structural check while reusing handlers across types. The inspected program assigns several types a static rule and sends others to a fallback. A registry therefore confirms the dispatch organization, while the source audit shows how much behavior the rules actually distinguish.

F.6.1 Qualitative source audit

Listing 8: Example 1. History retained by $\widehat{\mathcal{S}}_{\mathrm{B}}$ Code Structure.

```
if _sig(state) != _ACC["last_sig"]:
        # State does not continue our last output (fresh level / planner branch
        # / reset). Drop stale history; freeze any ghosts present (unknown tapes).
        _ACC["tape"] = []
        _ACC["ghosts"] = [
            {"tape": [], "idx": 0, "x": g["x"], "y": g["y"]}
            for g in state if g.get("name") == PLAYER and not _has9(g)
        ]
```

Listing 9: Example 2. History retained by No Natural Language.

```
sig_in = _sig(state)
    if _RUNTIME["sig"] is not None and sig_in == _RUNTIME["sig"]:
        rt = {"n": _RUNTIME["n"], "hist": list(_RUNTIME["hist"]),
              "deposits": [dict(d) for d in _RUNTIME["deposits"]],
              "gidx": _RUNTIME["gidx"], "v": _RUNTIME.get("v", (STEP, STEP)),
              "maxis": _RUNTIME.get("maxis")}
    else:
        rt = {"n": 0, "hist": [], "deposits": [], "gidx": 0, "v": (STEP, STEP), "maxis": None}
```

Listing 10: Example 3. The unablated world model records why the gate explanation changed.

```
## RUN-24: latching (co-planar) gate is a TOGGLE
Co-planar gate `qinemeqqdw` oscillates between home and home+slide on each plate
RISING edge (any actor). Falling edge = no move (latched). RECORD/RESET restore
home. Supersedes the "latch-forever" reading (L2/L3 never double-pressed without
a record, so they were consistent with either; L5 s572+s578 double-press forces
the toggle). Momentary above-player gates unchanged. See `gate_ext` accumulator.
```

Both the unablated and No NL engines represent movement history in Python. The difference concerns the record of why a rule was adopted and what remains uncertain. The unablated artifacts retain rejected explanations and the observations that replaced them, giving later synthesis access to reasoning that No NL removes.

Listing 11: Shared context before type dispatch in Type-Oriented Code Structure.

```
ctx = _make_ctx(state, action_id)
    new_state = [TYPE_RULES.get(object_type(o), _static_rule)(o, state, action_id, ctx)
                 for o in state]
```

Type-Oriented Code Structure still computes shared context before dispatching its object rules. The handlers depend on information about other objects despite their separate organization. This supports the RQ2 argument that object-indexed state can coexist with shared transition logic, which the unablated condition allows synthesis to organize around the interaction.

F.7 Revision protocol and results

The advantage of $\widehat{\mathcal{S}}_{\mathrm{B}}$ Code Structure is clearest when an existing mechanic changes locally. M1 and M3 compare repair metrics under evidence matched across conditions. M2 measures revisions as later levels introduce mechanics. This separates a local correction from additions that require broader changes to the model (Figure 7).

Interventions and scope of replication. M1 rotates the sliding doors' orientation-to-displacement mapping by $90°$, changing one existing object mechanic. M3 changes the global timer to decrement every third move instead of every second, increasing the available action allowance. Both interventions preserve the level layouts and goal conditions. Repetition across evidence trajectories tests sensitivity to which transitions reveal that modification, while repetition across independently synthesized starting programs tests sensitivity to the model being repaired; within each comparison, repair evidence is matched across conditions. These repetitions support conclusions about repair under the tested interventions, including evidence for localized repair in M1. Each of these runs is prohibitively expensive, and the observed benefits under this general case were convincing enough to motivate this approach.

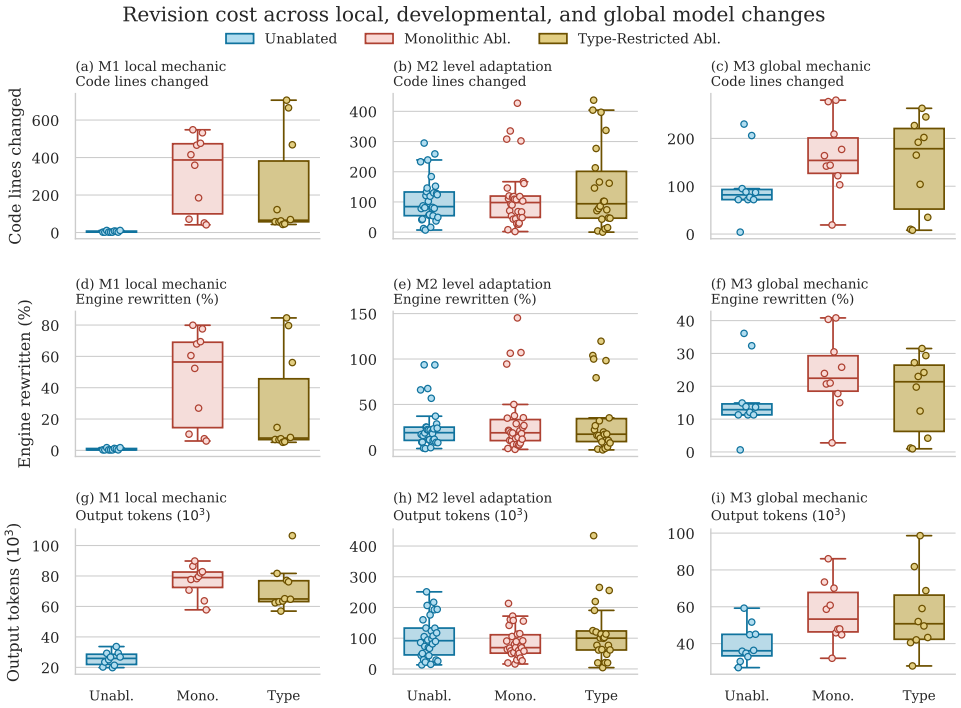


Figure 7: **Full distributions of code and synthesis required by three model revisions.** M1 and M3 show ten repair runs per condition. M2 attributes changes to objects and relations introduced at each level transition in the fixed-trajectory runs. Code lines exclude comments and blank lines. Engine rewrite divides changed code by entering engine size.

Table 13: **Detailed revision measurements.** M1 and M3 are means over ten repairs per condition. M2 reports 30 level transitions for the full and monolithic conditions and 22 transitions with available snapshots for the type-restricted condition. Code lines exclude comments and blank lines. Churn divides changed code by entering engine size.

Change	Condition	Code lines	Churn (%)	Definitions	Rounds	Output (k)
M1: local	Full system	4.8	0.75	1.1	2.00	25.6
	Monolithic	314.4	45.83	9.4	5.00	76.9
	Type-restricted	229.6	27.50	18.2	5.00	71.7
M2: level	Full system	107.5	25.92	11.8	4.70	101.5
	Monolithic	115.7	30.57	4.1	3.93	84.5
	Type-restricted	146.3	34.01	13.4	5.05	117.8
M3: global	Full system	100.3	15.75	10.8	4.00	39.8
	Monolithic	163.7	23.86	7.5	4.00	56.8
	Type-restricted	145.1	17.38	12.1	4.00	56.4

For M1, $\widehat{\mathcal{S}}_{\mathrm{B}}$ Code Structure requires less rewriting, fewer synthesis rounds, and fewer output tokens than the restricted structures (Tables 13 and 14). The source examples below show how the same local intervention can lead to either a small correction or a broad revision.

Table 14: **Tests of revision cost against the full system.** M1 uses exact two-sided sign-flip tests on ten matched repair runs. M3 uses exact two-sample permutation tests on ten independent synthesis replications per condition. All p-values are unadjusted.

Change	Outcome	Monolithic p	Type-restricted p
M1	Changed lines	.00195	.00195
	Engine churn	.00195	.00195
	Synthesis rounds	.00195	.00195
	Output tokens	.00195	.00195
	Changed definitions	.00586	.00195
M3	Output tokens	.0117	.0356
	Changed lines	.0700	.2448

When later levels introduce new mechanics, the revision costs are closer across conditions. The added behavior can require changes across interacting objects even when existing rules have separate helpers. M2 compares the entering engine with the first replay-consistent engine after the new level's evidence, retaining earlier transitions in replay. Its observations are grouped within the evolving programs, and some Type-Oriented Code Structure snapshots are unavailable.

For the global rule change, the conditions take the same number of synthesis rounds, while $\widehat{\mathcal{S}}_{\mathrm{B}}$ Code Structure uses fewer output tokens. The reduction comes from generating less material during repair rather than completing fewer attempts (Table 14).

A small repair is useful when it also incorporates the changed behavior. The M1 prediction check evaluates unseen transitions from the modified world alongside edit size. The unablated model improves after repair, supporting the finding that a local code correction can update the mechanic without a broad rewrite.

F.8 NL INTERVENTION AND ONLINE RESULTS

RQ3 removes persistent NL model content while retaining the actor and synthesis roles. Comments and docstrings are removed from the persisted program, while observations, verifier feedback, and communication artifacts remain available. This tests whether retaining hypotheses and explanations helps later synthesis beyond the executable state and evidence available to both conditions.

The full system needs fewer synthesis rounds during early learning and generally uses fewer actions and tokens than the NL ablation (Table 15). The available source snapshots also show less rewriting. Persistent explanations help later synthesis continue from earlier reasoning while object behavior is still uncertain.

Table 15: **Online NL ablation.** Run values are ordered by run index. Actions count interaction through completion of level 3. Tokens combine synthesis output and billable input and are reported in millions. Changed-code values are available for three runs. Each p-value comes from an exact two-sample permutation test over condition labels.

Outcome	Full-system runs	Ablation runs	Full mean	Ablated mean	Full reduction	Exact p
Actions to clear level 3	169, 277, 237, 176, 227	626, 268, 756, 249, 289	217.2	437.6	50.4%	.0238
Synthesis rounds	14, 16, 16, 10, 13	39, 19, 32, 17, 14	13.8	24.2	43.0%	.0397
Total tokens (M)	31.95, 38.99, 31.73, 29.71, 32.01	109.73, 35.44, 90.84, 40.40, 45.00	32.88	64.28	48.9%	.0159
Changed code lines	629, 839, 958	1,883, 1,021, 1,413	808.7	1,439.0	43.8%	.1000

The token breakdown shows how the difference in synthesis effort contributes to the online cost comparison (Table 16). Changed-code measurements use the runs with available source snapshots, as identified in Table 15.

Table 16: **Token components in the online NL ablation.** Means over five runs per condition, in millions of tokens. Reductions use the ablated condition as the denominator. Tests use exact two-sided permutations of condition labels.

Component	Full mean	Ablated mean	Full reduction	Exact p
Output	.401	.764	47.5%	.0238
Billable input	32.48	63.52	48.9%	.0159

The same pattern extends to prediction when interaction is held fixed. The unablated models have higher mean accuracy than No NL on the shared held-out buffer (Table 11). Persistent explanations can therefore help synthesis interpret the evidence it has already received, in addition to helping the actor choose useful actions.

F.9 Computation and checked planning

Repeated synthesis is a major source of computational cost because successive sessions revisit accumulated evidence. The recorded synthesis work totals 174 hours across the full-system benchmark runs. Separate actor and synthesis contexts limit what must be carried into action selection because the actor receives selected model artifacts and evidence summaries rather than the full synthesis transcripts.

Listing 12: Example 1. Token-position mappings used by the `lp85` planner.

```
for op in ops:
        mp = {}
        for keys in op["cycles"]:
            ids = [idx[k] for k in keys]
            n = len(ids)
            for k in range(n):
                src = ids[(k + 1) % n] if op["ccw"] else ids[(k - 1) % n]
                mp[src] = ids[k]
        perms.append((mp, op["click"]))
```

Listing 13: Example 2. Transition and reward checks in the `ar25` planner.

```
nxt = transition_function(cur, a)
            r, done = reward_function(cur, a, nxt)
            new_path = path + [a]
            if r and r > 0 and done:
                return new_path
```

The learned program also gives the planner a smaller representation of the task. In `lp85`, button presses become permutations of token positions. The planner can search over these positions instead of rendered frames. Candidate solutions remain dependent on the learned transition and reward rules, which are checked during execution. This combination makes the program useful for finding actions as well as predicting them.

G Scoring and Protocol Checks

For game g, let L_g be its number of levels, let c_g be the number completed in order, and let $b_{g\ell}$ and $a_{g\ell}$ be the human and agent action counts for completed level ℓ. The level contribution and total level weight are

$$z_{g\ell} = \begin{cases} \min\left\{(b_{g\ell}/a_{g\ell})^2, 1.15\right\}, & \ell \le c_g, \\ 0, & \ell > c_g, \end{cases} \qquad W_g = \sum_{\ell=1}^{L_g} \ell. \tag{34}$$

Equation 34 assigns zero to incomplete levels while retaining their weight in W_g. The completed-weight cap prevents an early bonus from raising a game above the fraction of level weight completed.

For a set of G games,

$$S_g = 100 \min\left\{ \frac{\sum_{\ell=1}^{L_g} \ell z_{g\ell}}{W_g}, \frac{\sum_{\ell=1}^{c_g} \ell}{W_g} \right\},$$
$$\text{RHAE} = \frac{1}{G} \sum_{g=1}^{G} S_g. \tag{35}$$

Equation 35 averages the capped game scores to obtain RHAE. This is the scoring implementation used by the official scorecards (ARC Prize Foundation, 2026a;c).

The public scorecards provide game-version hashes, baselines, action counts, and per-level scores. Table 17 compares each card with the game versions used by OPINE-World. The April 2026 update replaced the second-best human reference with the median human reference, raised the per-level cap from 1.00 to 1.15, and changed 15 builds. The hash split and the cap inferred from the per-level entries agree for each card (ARC Prize Foundation, 2026b;c).

Table 17: **Public-scorecard protocol audit.** A dash in the published column means that the card contains per-level data but no published total.

System	Matching builds	Different builds	Cap	Published / recomputed
OPINE-World	25	0	1.15	78.40 / 78.40
baseline1	25	0	1.15	63.74 / 63.74
DreamTeam	25	0	1.15	38.06 / 38.06
Continual Harness	25	0	1.15	20.54 / 20.54
OpenClaw	25	0	1.15	5.20 / 5.20
GPT-5.6 Sol (max)	25	0	1.15	7.80 / 7.80
Claude Opus 4.8 (high)	25	0	1.15	1.50 / 1.50
TELL	10	15	1.00	– / 42.92
a-evolve	10	15	1.00	– / 4.28

The scorer reproduces 1,296 per-level values across the five external system cards when each card's cap is applied. We exclude TELL and a-evolve from the main comparison because their game builds and scoring cap differ. The standalone GPT-5.6 Sol and Claude Opus profiles use the 183-level baseline records published with the benchmark. DreamTeam's public card uses the same game builds and scoring rule and contains one run at 38.06, while its paper reports a two-run mean of 38.36 (Sarafian et al., 2026).

G.1 Detailed baseline coverage

Among the 13 games completed by both OPINE-World and baseline1, OPINE-World uses fewer actions on six and baseline1 uses fewer on seven, with similar aggregate actions. OPINE-World also completes seven games that baseline1 does not complete. Table 18 gives the six largest action gaps. The remaining game is `dc22`, where OPINE-World clears six levels in 1,479 actions and baseline1 reaches four levels.

Table 18: **Coverage beyond baseline1.** baseline1 exhausts its budget before completing these games, while OPINE-World completes them. Human gives the reference action count from the public scorecard (ARC Prize Foundation, 2026c).

		baseline1		OPINE-World
Game	Human actions	Actions	Levels	Actions
`re86`	1,255	1,754	4/8	**850**
`tn36`	317	3,058	6/7	**417**
`vc33`	447	1,596	3/7	**427**
`m0r0`	1,107	2,573	5/6	**259**
`sc25`	350	1,107	3/6	**256**
`sp80`	518	786	1/6	**369**
Total	3,994	10,874	–	**2,578**

H AI USE STATEMENT

We used generative AI tools to assist with creating and modifying scientific figures, writing and editing software code, drafting, polishing and proofreading the manuscript, and retrieving and identifying potentially relevant literature. We have not used generative AI tools to generate synthetic datasets, develop the theoretical or conceptual framework, formulate mathematical claims, contribute to or write proofs, propose or refine hypotheses, design the research methodology or experiments, perform translation, clean or reformat datasets, conduct qualitative or thematic analysis, or interpret the experimental results. LLM-generated code was verified and tested for correctness by the authors, all figures were checked against the underlying data, drafted text was edited and fact-checked by the authors, and LLM-identified literature was read and confirmed relevant before citation. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

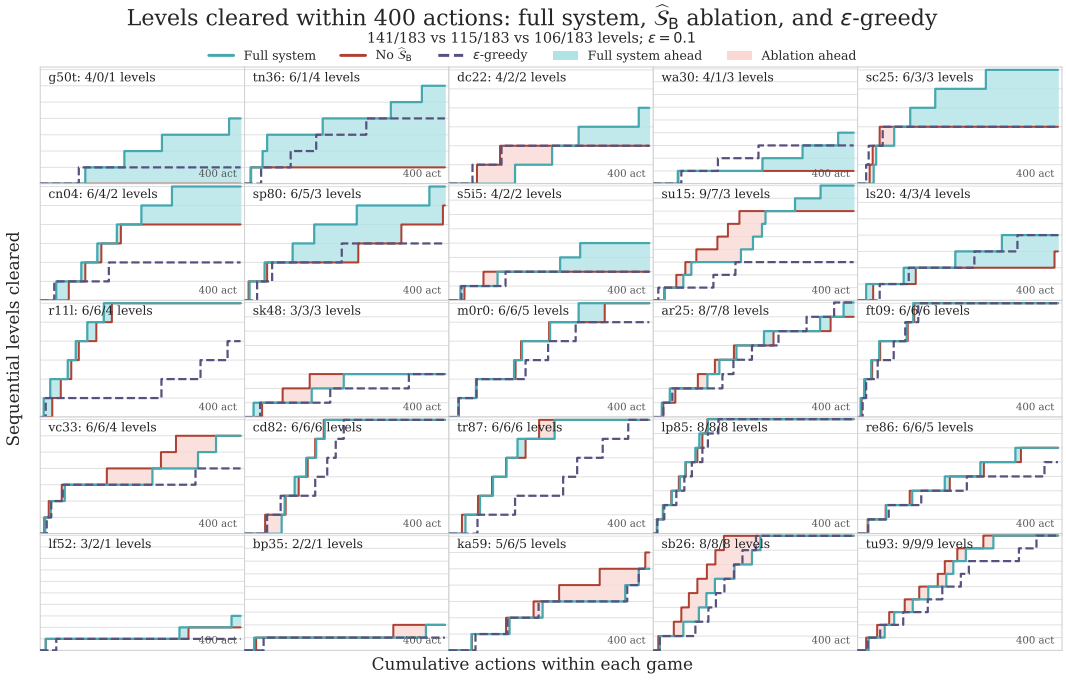


Figure 6: **Levels completed within 400 actions** for the full system, the $\widehat{S}_B$ ablation, and ϵ-greedy ablation.