PROPS: POSTERIOR SAMPLING FOR PROGRAMMATIC WORLD MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Programmatic world models represent an environment's dynamics as code that a language model synthesizes from observed transitions, and recent methods decompose this code into many small "laws". We study how an agent can plan with such laws. Because each law states only part of what an action does, we compile laws into planning rules that each state an action's whole effect. However, since a rule's context can stay partial, two uncertainties remain: 1) whether an action can run at all, and 2) which of the rival rules compiled for an action is right. We introduce Program Posterior Sampling (ProPS), a controller that keeps an availability posterior and a rule posterior over these two unknowns, plans under a sample from both, and updates both from each outcome. For a small library of complete programs, we prove that an exact version of ProPS is optimal under Bayes-adaptive dynamic programming, while a general version trades this guarantee for scale. On Crafter-OO, ProPS outperforms all baselines given the same laws, including a product of experts and an LLM agent that reads the laws as text.

1 INTRODUCTION

Programmatic world models have gained increasing attention as a sample-efficient alternative to parametric world models. While parametric models learn next-state prediction in pixel or latent space from large collections of trajectories (Ha & Schmidhuber, 2018; Hafner et al., 2020), programmatic world models represent dynamics as code that a large language model synthesizes from observed transitions (Tang et al., 2024; Dainese et al., 2024; Piriyakulkij et al., 2025; Khan et al., 2026). Because code expresses conditional, compositional, and counterfactual structure directly, programmatic world models can generalize from limited data and remain interpretable and revisable (Khan et al., 2026).

Recent work decomposes world modeling into writing hundreds of smaller programs. Unlike earlier methods, which refine one monolithic program over many costly iterations (Tang et al., 2024; Dainese et al., 2024), PoE-World (Piriyakulkij et al., 2025) synthesizes context-specific *expert programs* that each fit one or a few transitions, and learns a weight per expert to combine them into a single predictive distribution, a *product of experts*. OneLife (Khan et al., 2026) further factorizes the transition function into *laws*, small programs that each state a *context*, the condition under which the law applies, and an *effect*, its prediction for a small part of the next state, such as the player's health or a single map tile. Each law is thus *partial* (Talvitie & Singh, 2008; Khetarpal et al., 2021), which lets OneLife credit each observed change only to the laws that predicted it. With this credit assignment, the approach scales from gridworlds and Atari to Crafter-OO, an object-oriented version of the survival game Crafter (Hafner, 2022).

In this work, we study how an agent can plan with such laws in complex environments like Crafter-OO. While Khan et al. (2026) show that laws can identify the better of two hand-written plans, in their evaluation no agent selects a goal, constructs a plan, or acts on one. We make the key observation that laws are easy to synthesize and hard to plan with for the same reason: each pairs one precondition with an effect on a small part of the state, whereas an action typically has several effects. Although laws thus have the precondition–effect form of classical planning operators (Khan et al., 2026), a classical planner cannot chain them as written, since each law states only a part of what an action does and requires.

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.

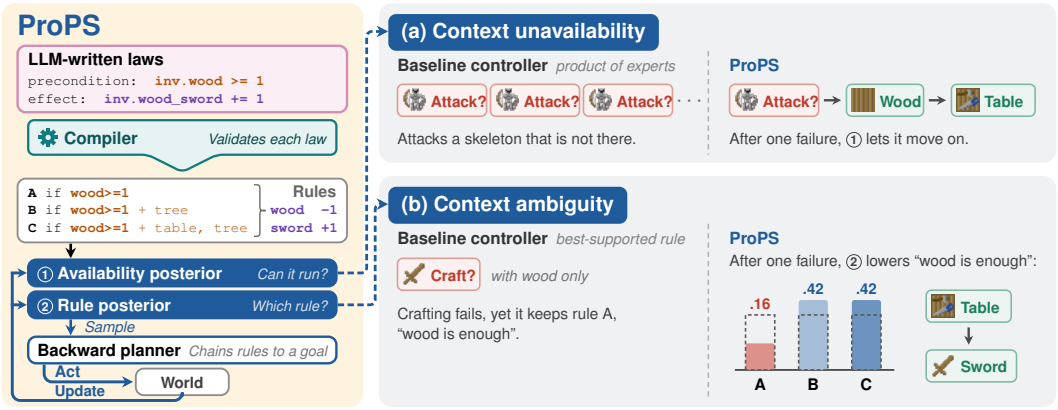


Figure 1: **Overview of ProPS.** Left: a compiler turns LLM-written laws into planning rules, and at each decision ProPS samples from an availability posterior ① and a rule posterior ②, plans under the sample, and updates both from the outcome. (a) When no skeleton is in view, the attack cannot run, and since nothing contradicts its rule, the baseline keeps attacking, whereas ① lets ProPS move on. (b) When crafting fails with wood only, the baseline keeps rule A, whereas ② moves weight to rules B and C, which require more than wood and so are consistent with the failure (dashed: before).

We therefore propose to compile laws into planning *rules* that each state an action's whole effect, keeping a rule only if its laws reproduce the transitions they were written from (§2.2; Figure 1, left). For instance, the rule compiled from two laws for crafting a wood sword, one consuming wood and one producing the sword, states both changes, and its context adds a nearby table and tree that the compiler induces from the data (rule C in Figure 1). To plan, we chain these rules backward from each achievement the agent has not yet earned and pursue the goal whose plan is most likely to succeed, replanning after every action.

Although compiling makes a rule's effect whole, its context, what the action requires, still remains partial, which leaves two ways for planning to fail (Figure 1). The first is **context unavailability**, where an action cannot run. The rule for attacking a skeleton predicts that when the agent faces a skeleton (context) and strikes it, the skeleton takes damage (effect), but it says nothing about whether a skeleton can be reached. When none is in view, the attack fails to run. Since no transition occurs, this failure does not contradict the rule, so a planner that trusts it repeats the attack (red boxes in Figure 1a, left). Under our product-of-experts baseline, 98% of the agent's actions cannot run. The second is **context ambiguity**, where competing rules disagree about the context an action needs. None of the laws for crafting a wood sword mentions the table that crafting requires, and because the compiler induces the rest of each rule's context from one source episode at a time, they compile into three rules that differ only in context (rules A–C in Figures 1 and 2b). When the agent crafts with wood but without a table, the wood-only rule A predicts success, yet crafting produces no sword. A controller that holds its rule or its weights fixed, such as our best-supported rule or product of experts,[1] keeps relying on A and only lowers its predicted success (Figure 1b, left).

To address these challenges, we introduce Program Posterior Sampling (ProPS), a controller that treats what the laws leave unstated as unknowns and infers them from the outcomes of its own actions. Whereas these controllers either repeat an action that cannot run or weaken a fixed rule, ProPS credits each failure to one of two posteriors. For instance, when an attack does not run, ProPS records this in its *availability posterior*, and the attack's availability falls, so the agent stops wasting actions and pursues another reachable goal (Figure 1a, right). When crafting runs but fails, ProPS treats the failure as evidence against the rule it relied on. Its *rule posterior* lowers its belief that wood is enough and raises its belief in rival rules, such as the one that also requires a table (from the dashed to the colored bars in Figure 1b, right). We pose control under this uncertainty as a Bayes-adaptive decision problem (Feldbaum, 1960; Guez et al., 2012), and show that with a small library of complete programs, our exact controller is optimal, whereas choosing the most informative test can fail (§2.1). For open-ended environments, where exact planning does not scale, we draw on posterior sampling (Thompson, 1933; Strens, 2000; Osband et al., 2013). At each decision, ProPS

[1]PoE-World instead refits its weights and synthesizes new experts while it acts, at the cost of further language-model calls (Piriyakulkij et al., 2025).

samples for each action one rule and a probability that it runs, plans under the sample, executes the first action, and updates both posteriors from the outcome (§2.2).

Our contributions are as follows:

- We introduce Program Posterior Sampling (ProPS), a controller that compiles LLM-written laws into planning rules validated against their source transitions, and plans under samples from an availability posterior and a rule posterior that it updates from each outcome, with no language-model call during control (§2).
- We prove that the exact controller, which plans by backward induction over a supplied library of complete programs, is optimal under Bayes-adaptive dynamic programming, and we show, through constructed problems, that choosing the most informative test can fail where valuing a test by what it adds to the task succeeds (§2.1).
- We show on Crafter-OO that planning under a posterior over rival rules, rather than committing to a single rule, improves control. In controlled worlds, testing the disputed rule before committing raises success from 29.9% for controllers bound to one fitted rule to 83.3%, close to an oracle. In open-ended play with the same 161 laws, ProPS outperforms all baselines, including a product of experts, a best-supported rule, and an LLM agent that reads the laws as text. Finally, we provide ablation studies showing that removing either posterior reduces what the agent achieves (§3).

2 Program Posterior Sampling

We now introduce Program Posterior Sampling (ProPS), a controller designed to plan with a programmatic world model whose programs leave open which one governs each action. ProPS plans with *rules*, planning operators compiled from synthesized laws, each of which declares an action's whole effect and the context it needs. It keeps *posteriors* over which rule governs each action and whether each action can run, and revises them from the outcomes of its own actions. We first give an exact controller for a small library of complete programs and prove it optimal (§2.1). We then trade this optimality guarantee for scale and extend the loop to the partial laws of open-ended worlds, where ProPS weights rival rules by their evidence and resamples at every decision (§2.2).

Setting. Every episode follows the same stationary dynamics from its own initial state. At each step the agent observes x_t, a set of named variables such as `inventory.wood` or the material of the tile it faces. It chooses a task-level action (an operator) o, such as chopping a visible tree, which a fixed adapter turns into primitive moves. The adapter also reports whether the action *fired*, that is, ran. An action can fail to run when, for instance, its target is absent. From k source episodes $D^{(1:k)}$, a synthesizer writes *laws* (Khan et al., 2026), which form the controller's starting portfolio $\mathcal{H}_0$. A law $h = (c_h, f_h, S_h)$ is a small program with a *precondition* c_h, the context the law states, and an *effect* f_h that predicts the next values of the variables in its *write scope* S_h. We call such a law *partial* in both its effect and its context, in the sense of local and partial models that make only certain predictions in only certain situations (Talvitie & Singh, 2008; Khetarpal et al., 2021): it predicts only the variables in its scope, and only where its precondition holds, so describing one action can take several laws. A *complete program*, by contrast, predicts the whole next state wherever it is applied, and a *plan* π is a sequence of actions. Each evaluation episode starts in a new world with the same portfolio and allows B decisions, and what the controller learns stays within that episode.

2.1 Exact Control via Task-Conditioned Acquisition

Programs synthesized from the same source episodes can all reproduce them and still disagree about what an action does, so the agent must act without knowing which program is right. We first study this decision in its simplest form, a few complete programs of which one describes the world. Each action then does two things, it moves the agent toward its goal, and its outcome tells the agent something about which program is right. The best action therefore depends on the agent's belief, a probability distribution over the programs, as well as on the state, and on how each possible outcome would change that belief. Planning over the state and the belief together is a finite-horizon *Bayes-adaptive* decision problem (Feldbaum, 1960; Guez et al., 2012), in which a test is worth only the gain in task success its outcome makes possible. Although such problems are intractable in general,

with a few complete programs they can be solved exactly, and the exact controller of ProPS does so. With deterministic outcomes, its belief is the prior restricted to the programs not yet contradicted.

To illustrate, we design controlled worlds (§3.1) in which three such programs disagree about which sword defeats a creature. Crafter attacks with the highest sword held, so crafting one cannot be undone. At each decision the controller can *commit*, executing the best plan under its current belief, or first run a *diagnostic*, a supplied action whose outcome tells the programs apart, such as crafting a sword and striking a cow to read the damage. Formally, we write z for the task state, β for the belief, and m for the remaining budget. The value $\mathcal{V}$ of an augmented state (z, β, m) is defined as

$$\mathcal{V}(z, \beta, m) = \operatorname{lexmax}\{\mathcal{Q}_{\text{commit}}(z, \beta, m), \mathcal{Q}_a(z, \beta, m) : a \in \mathcal{A}_{\text{diag}}(z, m)\}, \tag{1}$$

where lexmax prefers the highest completion probability and then the fewest expected actions. The recursion compares committing now, whose value $\mathcal{Q}_{\text{commit}}$ is taken under the current belief with the whole remaining budget, with each diagnostic a in $\mathcal{A}_{\text{diag}}(z, m)$, those available at z within budget m. The value $\mathcal{Q}_a$ of a diagnostic averages, over its outcomes $y \sim \Pr_\beta(y \mid a)$, the value of acting on the updated belief with the budget left after its cost $c(a)$. A diagnostic therefore wins only when what it reveals improves the later choices by more than the budget it spends. Backward induction over the budget solves this recursion, and we call the resulting policy the *exact policy*.

We now show the optimality of this policy and its separation from the standard criterion of selecting the most informative test, the one expected to reduce uncertainty about the programs most (Lindley, 1956; Chaloner & Verdinelli, 1995). Although the recursion is the natural candidate for optimality, the objective is lexicographic, so the proof must also cover policies that choose their actions at random. Although this test is the usual criterion, it can concern something the task never uses, such as a creature the goal does not name, so the two can disagree. The theorem below establishes both: the recursion is optimal, and on a family of problems the most informative test fails where the recursion succeeds, by a margin growing with the number of task-relevant alternatives.

Theorem 1 (Optimality and task-blind separation)**.** *Let the hypothesis and action sets be finite, every diagnostic have finitely many outcome branches with exact hypothesis-conditioned kernels and branches sufficient for Bayesian updating, action costs be positive integers, and the physical successor be determined by the state, the diagnostic, and its outcome. For the lexicographic objective (completion probability, negated expected actions):*

(i) Equation 1 attains the optimal value over all history-dependent policies, including randomized policies.

(ii) For every $K \geq 2$ and $N > K$, a deterministic two-decision problem (one test, then one commitment) with unit-cost tests and a uniform prior on $\{1, \ldots, K\} \times \{1, \ldots, N\}$ admits success 1 under the exact policy and $1/K$ under cost-aware greedy information gain.

Appendix A.1 gives the proofs and the construction behind part (ii). The theorem covers only the exact decision problem, and the compiled controller below replaces its backward induction with a sample.

2.2 Compiled Control from Synthesized Laws

We now extend the loop to the partial laws a synthesizer produces (Piriyakulkij et al., 2025; Khan et al., 2026). Here complete programs, a joint posterior, and exact backward induction are unavailable, and the agent faces a second unknown, whether each action can run at all (Figure 1). The resulting controller applies posterior sampling to a factored hypothesis space (Osband & Van Roy, 2014): the rival rules of each action form one factor, a sample picks one rule per factor, and a backward-chaining planner acts under the sample. Only the compiler is specific to synthesized laws. We next describe the controller in four steps, 1) compiling laws into rules, 2) the two posteriors, 3) planning under a sample, and 4) updating from the outcome.

(1) Compiling laws into rules. Several laws may describe one action, each predicting one piece of its effect, such as wood -1 when a sword is crafted (Figure 2a). A planner needs instead *rules* that each declare an action's whole effect and the context it needs. Formally, a rule is a tuple $\kappa = (o, g, \mathcal{O} \cup \{\emptyset\}, S)$ of an action o, a guard g, a set $\mathcal{O}$ of explicit *outcome clauses*, each one observable change such as wood -1 with wood sword $+1$, a residual token $\emptyset$ that covers any other change so that every outcome can be decoded, and a write scope S (rule C in Figure 2a). The

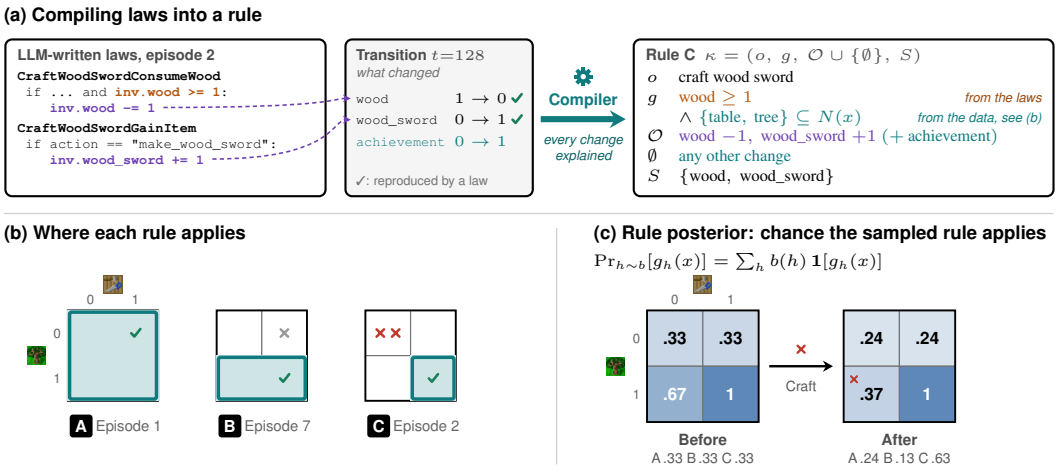


Figure 2: **Rule compilation and the rule posterior,** for crafting a wood sword, one of 16 compiled actions. (a) Two laws from source episode 2 each state a context and one piece of the effect, and together reproduce every change of their transition (✓), so the compiler emits a rule whose guard adds context induced from the data (teal). (b) Each rule's guard (teal) comes from one source episode's successes (✓) and failures (✗; grey: without wood), over whether a table (columns) and a tree (rows) lie within one tile. (c) When crafting fails beside a tree without a table, in one evaluation world, weight moves to rule C, which did not apply there.

compiler works on the source transitions (x, o, y) the laws were written from. For each distinct change an action produced, it takes the observed change as the rule's outcome. It compiles the rule only if the laws written for that action in the episode, replayed on the transition, reproduce every inventory and target coordinate the change altered (Appendix B.2, Algorithm 2). The guard joins the context these laws state with context the compiler induces. A nearby material or a tool joins it when present at every success of the action in the episode and absent at the most failures, so an episode without failures adds nothing (rule A in Figure 2b). A rule thus differs from a law on both axes: its effect is *whole*, and its context adds what the data show, although the induced part can still be incomplete or wrong.

Because induction runs within each source episode, episodes that failed differently yield rival rules for one action. Identical rules from different episodes merge into one. Writing h for a rule and κ_h for its tuple, the rival rules of an action form its *factor* f, whose rules $\mathcal{H}_f$ the rule posterior chooses among, and $\mathcal{F}$ is the set of factors. The 161 law records of our study, 86 of them distinct, compile into 47 rules in 16 factors, one per action. Guards and outcomes use the same coordinates, so the planner can chain rules, one rule's outcome satisfying another's guard (Appendices B.3 and B.2).

(2) Rule and availability posteriors. Over the compiled rules, ProPS maintains two posteriors, one over which rule governs each action and one over whether each action can run. The *rule posterior* $b = (b_f)_{f \in \mathcal{F}}$ holds a distribution b_f over the rules $\mathcal{H}_f$ of each factor f. It starts uniform and is updated by replaying the source transitions in order. We assume that beliefs about different factors are independent and that an action's outcome is evidence only about its own factor. A choice of one rule per factor, $(h_f)_f$, then has weight $\prod_f b_f(h_f)$, which stands in for the joint posterior β of the exact controller. Replay settles some factors, such as placing a table (Appendix Table 4), and not others, such as crafting a wood sword, whose three rules stay at one third (Figure 2c). Each rule also counts how often each outcome clause c occurs, $n_\kappa(c)$, from source replay on, with the Dirichlet ($\alpha = 1$) predictive

$$q(\kappa, c) = \frac{n_\kappa(c) + 1}{\sum_{c' \in \mathcal{O}_\kappa \cup \{\emptyset\}} (n_\kappa(c') + 1)}. \tag{2}$$

Thus b holds the uncertainty over which rule is right, and q the stochasticity of what a rule delivers. The *availability posterior* of action o is $\mathrm{Beta}(1 + f_o, 1 + n_o - f_o)$, where n_o counts the times o was selected in the episode and f_o the times it fired.

(3) Planning under a sample. At each decision, ProPS samples both posteriors, drawing one rule $h_f \sim b_f$ per factor and, from each action's availability posterior, a probability ρ_o that the action runs. It chains the sampled rules backward from each unearned achievement into plans (Fikes &

Nilsson, 1971) and uses the sampled probabilities to discount plans through actions that may not run. Such plans use at most one rule per factor, so each describes a possible world, and we call them *hypothesis-coherent*. Step i of a plan π of L steps requires clause c_i of rule h_i, which occurs with probability $q_i = q(\kappa_{h_i}, c_i)$ (Equation 2). We define the score of π under a belief b as

$$J(\pi; b, m) = W(\pi; b)\,\mathrm{Succ}(1, m), \qquad W(\pi; b) = \prod_{f \in \mathrm{factors}(\pi)} b_f(h_f), \tag{3}$$

where m is the remaining budget and $\mathrm{Succ}(1, m)$ is the probability of completing all L steps within m actions when a failed step is retried at the same q_i (Appendix B.4). Under the rule posterior b itself, W would multiply the weights of the rules a plan names and rank a chain through contested factors below a one-step plan, even when each step succeeds under any of its rival rules. Scoring each step by the total weight of the rules that predict it removes this penalty deterministically (Appendix Table 11). Sampling removes it as well, since the sampled rules form a point mass $\tilde{b}$ with $W(\pi; \tilde{b}) = 1$. ProPS ranks plans by J_{comp}, the score under the sample discounted by availability:

$$J_{\mathrm{comp}}(\pi; \tilde{b}, \rho, m) = J(\pi; \tilde{b}, m) \prod_{i=1}^{L} \left[1 - (1 - \rho_{o_i})^m\right], \tag{4}$$

where each term of the product is the chance that a step's action runs within m attempts. Because each term spends the whole remaining budget, the product is a ranking proxy rather than a joint probability. The agent executes the first action of the highest-scoring plan over all unearned achievements, found by branch-and-bound search, and replans at the next decision (Figure 3 in Appendix B.1). Unlike the exact policy of Equation 1, each plan is a fixed sequence that ignores what later outcomes will reveal, and replanning after every outcome partly compensates for this, as in open-loop feedback control (Bertsekas, 2001).

(4) Updating from the outcome. After each action, the adapter returns a receipt y_t, the observations before and after the action, and whether the action fired. Its availability counts always update, n_o by one and f_o by one if it fired, whereas the rule posterior learns only from fired actions, since an action that did not fire yields no transition that a rule predicts. For a fired action, the *applicable* rules $\mathcal{A}_t$ are the rules of its factor whose guards hold at x_t, and the other rules *abstain*. Each applicable rule κ decodes y_t on its write scope S_κ into the outcome $\tau(y_t; \kappa)$, where Π_S keeps only the variables in S,

$$\tau(y_t; \kappa) = \begin{cases} c, & \text{if clause } c \in \mathcal{O}_\kappa \text{ matches the change } \Pi_{S_\kappa} y_t, \\ \emptyset, & \text{if no clause matches,} \end{cases} \tag{5}$$

and increments that outcome's count n_κ in Equation 2. With $b_t(h)$ the weight of rule h at step t, $q_h(c) = q(\kappa_h, c)$, the applicable mass $B_t = \sum_{h \in \mathcal{A}_t} b_t(h)$, and its evidence-weighted mass $L_t = \sum_{h \in \mathcal{A}_t} b_t(h) q_h(\tau(y_t; h))$, all computed before the update, the ordinary update is

$$b_{t+1}(h) = \begin{cases} b_t(h) q_h(\tau(y_t; h)) B_t / L_t, & h \in \mathcal{A}_t, \\ b_t(h), & \text{otherwise.} \end{cases} \tag{6}$$

This means that the update routes evidence only to rules that made a prediction, as OneLife's credit assignment does (Khan et al., 2026), and conserves their total weight, so a lone applicable rule keeps its weight when it fails. Because rules decode different scopes, the update compares each rule's own predictive score, which makes it an evidence weighting rather than a posterior under one shared likelihood, and we call b the rule posterior for its role. The update has one more step, *confirmation*, which by convention treats an abstaining rule's false guard as predicting that nothing changes. When every applicable rule decodes to the residual while a rival abstained, confirmation renormalizes the whole factor with multipliers $q_h(\emptyset)$ for applicable rules and 1 for abstainers, so the rules whose guards excluded the failure gain weight (Appendix B.1). In Figure 2c, crafting fails beside a tree without a table, so A and B, which applied, lose weight to C, which abstained. The gain from this step over the ordinary update is small (Appendix G.4). These updates are scoped and never delete a rule, so every rule keeps positive weight (Proposition 1, Appendix A.2). These guarantees do not extend to identification or calibration. A false rule may survive indefinitely, so the open-ended experiments include deletion as a variant. Resampling at every decision forgoes the per-episode commitment on which the analysis of posterior sampling relies (Strens, 2000; Osband et al., 2013; Osband & Van Roy, 2017). Hypothesis-coherent plans keep that commitment only within each scored plan. Algorithm 1 (Appendix B.1) summarizes the loop.

3 EXPERIMENTS

We evaluate ProPS on Crafter-OO in two settings, 1) controlled worlds where a small library disagrees about an irreversible commitment (§3.1), and 2) the open-ended game, where every controller receives the same adapter, budget, and synthesized laws (§3.2).

Domains. Crafter (Hafner, 2022) is an open-ended survival game with 22 achievements over collection, crafting at stations, and combat, which Crafter-OO (Khan et al., 2026) modifies by exposing typed object-oriented states. The open-ended setting uses unmodified Crafter-OO, while the controlled setting changes which sword tier harms a designated creature.

Metrics. In the controlled setting, Success is the share of instances in which the agent earns the achievement for defeating the creature within its budget. AUC averages that share over deadlines from 1 to 48 actions, so that earlier success counts more, and Actions is the mean number of actions in the successful instances. In the open-ended setting, we measure the number of distinct achievements earned in one episode, reported as a mean with its standard error over the same 48 worlds for every controller, with the number of worlds that earn each of six early achievements.

3.1 CONTROLLED TESTS OF PROGRAM AMBIGUITY

We first isolate the effect of program ambiguity on irreversible action selection. We supply a library of three complete programs, each naming a different sword tier as the one that defeats the designated creature, and 48 procedural worlds vary which program is true. Crafting a sword cannot be undone. The source evidence is four unarmed fights, which all three programs reproduce. Before committing, a controller can test a tier by crafting it and striking a cow placed in the world, whose damage the programs predict differently. Under one common controller, we compare the programs that WorldCoder (Tang et al., 2024), PoE-World (Piriyakulkij et al., 2025), and OneLife (Khan et al., 2026) fit with their released code (**WorldCoder fit** and **PoE-World / OneLife fit**), a controller that commits to a random tier (**Random tier**), and an ablation that retains all three programs but never tests (**Retain all, never test**). All share the library, the recipes, the navigation and combat routines, and a 48-action budget, and an **Oracle** plans with the true program (details in Appendix D.1).

A posterior over programs resolves what one fitted program cannot. Table 1 shows that the exact controller of ProPS, which tests the disputed rule before it commits, succeeds on 83.3% of instances, 2.1 points below the oracle, while retaining all three programs without testing succeeds on 48.6%. The programs fitted by PoE-World/OneLife only succeed on 29.9%, on par with committing to a random tier, as ambiguous evidence gives a fitted program nothing that separates the tiers. WorldCoder's program predicts that no sword plan succeeds, so the common controller attempts no plan. Note that its optimism constraint would instead keep a program under which some sword plan succeeds, and replace it when contradicted, which amounts to committing to a tier without testing, which is equivalent to retaining all three programs without testing. In contrast, our exact controller resolves the rule before 98.6% of its commitments. Due to space limitations, Appendices G.2 and G.3 report two studies that compare our exact controller (valuing a test by its effect on the task) with information gain (valuing it by what it reveals about the programs). With one creature, cost-aware information gain ties the exact controller. When the goal names one of two creatures, task-blind information gain tests the one with more programs and succeeds on 26.6% of runs, against 81.9% for the exact controller, but matches it once told the goal's creature, so what matters is conditioning tests on the task (Theorem 1).

Table 1: Results in the controlled setting, over 144 instances (48 worlds × three hidden programs).

Controller	Success (%)	AUC	Actions
Random tier	28.5	0.114	29.72
WorldCoder fit	0.0	0.000	—
PoE-World / OneLife fit	29.9	0.124	29.14
Retain all, never test	48.6	0.187	30.54
ProPS, exact (ours)	**83.3**	**0.279**	32.93
Oracle (true program)	85.4	0.343	29.72

3.2 OPEN-ENDED CONTROL FROM LEARNED PORTFOLIOS

We next evaluate control from rules learned in separate episodes. Every controller gets the same 161 law records, synthesized with OneLife's pipeline using Gemini 2.5 Flash from eight episodes, and plays disjoint target worlds with the same observations and 64 decisions (details in Appendix D.2).

Table 2: Results on open-ended Crafter-OO, with rules compiled per source episode. The middle columns count the worlds (of 48) earning each of six early achievements of the 22, and Achievements is the mean number of distinct achievements ± standard error. The last group adds our availability posterior to each belief over the rules, and ProPS is the rule posterior with it.

Controller	Place table	Wood pickaxe	Collect stone	Collect coal	Collect drink	Wood sword	Achievements
LLM agents							
ReAct (no memory)	29	8	4	1	3	6	2.33 ± 0.19
ReAct + reflection memory	46	23	14	0	8	0	3.27 ± 0.18
ReAct + law memory	46	35	24	5	5	3	3.98 ± 0.25
Planning-based							
Product of experts	0	0	0	0	0	0	1.06 ± 0.04
Best-supported rule	7	5	5	0	44	0	2.27 ± 0.12
Rule posterior (ours)	44	43	40	17	44	7	5.96 ± 0.29
Planning-based, with our availability posterior							
Product of experts	47	40	35	14	38	14	5.88 ± 0.25
Best-supported rule	28	17	13	5	30	10	3.29 ± 0.28
ProPS (ours)	48	43	39	17	37	16	**6.90** ± 0.27

Baselines. We compare against both planning-based and LLM agent baselines. **Product of experts** fits one weight per rule by maximum likelihood, following OneLife and PoE-World, and holds the weights fixed. **Best-supported rule** keeps for each action the rule with the highest replayed weight. We additionally evaluate these with our availability posterior, so that they differ from ProPS only in their belief over the rules and thus serve as ablations of its rule posterior. We also compare with LLM agents based on ReAct (Yao et al., 2023). **ReAct + law memory** reads the 161 law records as text, **ReAct + reflection memory** reads reflections on the source episodes in the manner of Reflexion (Shinn et al., 2023), and **ReAct (no memory)** has neither (Appendix D.4). Finally, ProPS is our full method, and **Rule posterior** refers to ProPS without its availability posterior.

ProPS is the most effective controller given the same laws. Table 2 shows that ProPS outperforms all baselines at 6.90 achievements, against 5.88 for the product of experts and 3.29 for the best-supported rule, both with our availability posterior, and at most 3.98 for a ReAct-based agent. The best-supported rule commits to whichever rule replay ranked first, even where replay left rivals tied. Reading the laws lifts ReAct by 1.65 achievements over no memory, but it stays below even the rule posterior alone. An LLM agent can use the laws only as advice, whereas an executable portfolio turns the outcome of each action into an update of the rules' weights.

A posterior over rival rules outperforms a single fixed rule. In the last group of Table 2, replacing the rule posterior by the best-supported rule costs 3.60 achievements, and by a product of experts 1.02. Even without the availability posterior, the rule posterior far exceeds both and loses to the single rule in none of the 48 worlds. Under rival hypotheses a contradicted rule is also abandoned sooner, as the wood sword shows (Figure 1b). With the availability posterior, the best-supported rule keeps rule A, which omits the table, and crafts to no effect 40.6 times per world, as it keeps no rival that the failure could favor. The product of experts can only lower its predicted success, because a rule whose guard is false contributes nothing to its prediction, and fails 6.8 times. ProPS instead treats a false guard, by convention, as predicting no change (confirmation, §2.2), so one failed attempt moves A from 0.33 to 0.16 and raises B and C. It fails only 2.8 times per world and places a table in all 48 worlds.

The rule posterior helps most when the rules are ambiguous. Most of the 3.60 achievements the rule posterior adds over the best-supported rule depend on our compiler's guard induction, which runs within each source episode (§2.2). Inducing guards from all eight episodes together instead, which we call joint induction, collapses the sword rules into the correct one, leaving the rule posterior less to resolve, and the baselines come closer. With the availability posterior, the best-supported rule rises from 3.29 to 6.23 and the product of experts from 5.88 to 6.69, while ProPS keeps the highest mean, 7.02, by margins within the variation across the 48 worlds (Appendix Table 11). A rule posterior is thus worth about 3.5 achievements over a single rule when the rules are ambiguous about a mechanism the task needs, and less than one when the compiler removes that ambiguity, as

eight of sixteen factors keep rival rules (Appendix D.3). No belief over a fixed library resolves the remaining failures (Appendix H.1).

Ablations. Removing the availability posterior lowers ProPS from 6.90 to 5.96, and adding it lifts both baselines, the product of experts most, from 1.06 to 5.88, as the share of its actions that cannot run falls from 98% to 44% (Table 2, Figure 1a). For the product of experts, the counts alone recover that gain, so the posterior helps it by remembering which actions could not run (Appendix G.5). Within the rule posterior, whose removal costs 1.02 to 3.60 as reported above, the gain comes from retaining and revising the rival rules. Keeping one rule per factor, or executing a plan without replanning, loses more than two achievements, whereas scoring each step by the total weight of the rules that predict it in place of a sample, or deleting a contradicted rule, changes little (Appendix G.4).

4 Related Work

Programmatic world models. WorldCoder (Tang et al., 2024) and CodeWorldModels (Dainese et al., 2024) synthesize a single program and refine it on the transitions it mispredicts. PoE-World (Piriyakulkij et al., 2025) and OneLife (Khan et al., 2026) decompose the world model into many context-specific programs, and fit one weight per program. WALL-E (Zhou et al., 2025) learns action rules as code from mismatches between predicted and real trajectories, and plans by model-predictive control. AutoManual (Chen et al., 2024) and SPRING (Wu et al., 2023) give a language agent rules as text. These systems act on one model at a time. In contrast, we carry the competing candidates into action selection and let the outcomes of the agent's own actions revise them.

Learning operators from experience. Our rules, with their residual outcome, have the form of the noisy deictic rules of Pasula et al. (2007). DOORmax (Diuk et al., 2008) learn deterministic precondition–effect schemas by elimination. Schema Networks (Kansky et al., 2017) learn factored precondition–effect schemas for games, and Guan et al. (2023) and Ahmed et al. (2025) have a language model write planning operators or bilevel models. Each of these learners commits to one consistent model. Our compiler instead produces operators of this kind from synthesized partial laws, validates each against the transitions its laws were written from, and keeps every validated alternative.

Bayes-adaptive control and posterior sampling. Control that treats model identity as part of the state is dual control (Feldbaum, 1960) and the Bayes-adaptive MDP (Duff, 2002; Guez et al., 2012). EMPA (Tsividis et al., 2026), the nearest prior controller, plans under its most probable program-structured theory of a game and explores by novelty and expected information gain. The value of information (Howard, 1966) rates a test by the decision it improves, information gain (Lindley, 1956) by what it reveals, as Piriyakulkij et al. (2024) do for rules that a language model proposes, and information-directed sampling (Russo & Van Roy, 2014) trades regret against information. Posterior sampling for reinforcement learning (Strens, 2000; Osband et al., 2013) commits to one draw per episode, and Arumugam & Griffiths (2026) apply it with hypotheses that a language model proposes. ProPS (§2.2) instead plans with compiled partial laws, weighs rival rules by evidence rather than an exact posterior, and draws again at every decision, as Thompson sampling (Thompson, 1933) does.

5 Conclusion

We address planning with a programmatic world model whose laws are partial, each predicting one small part of the next state in the one context it was written from. We introduce Program Posterior Sampling (ProPS), a controller that compiles such laws into planning rules and infers what the laws leave unstated from the outcomes of its own actions, keeping an availability posterior over whether each action can run and a rule posterior over which of an action's rival rules is right. For a small library of complete programs, we prove that an exact version of ProPS is optimal under Bayes-adaptive dynamic programming, where choosing the most informative test can fail. On Crafter-OO, given the same laws, ProPS achieves more than a product of experts, a single best-supported rule, and an LLM agent that reads the laws as text, and each posterior alone lifts what a controller achieves. More broadly, what a synthesizer leaves unstated need not be settled before an agent acts, because the agent can plan with rival rules and let the outcomes of its own actions decide among them.

AI USE STATEMENT

In this work, we used generative AI tools (Anthropic's Claude Code and OpenAI's Codex) to assist with formulating mathematical claims and writing and checking their proofs, and in initial ideation and code implementation. Additionally, we used them to produce early drafts of some text, to edit and polish the paper, and to identify related work. All AI-assisted generated code was manually verified and tested for correctness. All references suggested by generative AI tools were manually surveyed, and all proofs were manually verified. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

To support reproducibility, we will release the code of ProPS for both experimental settings, together with the synthesized laws and compiled rules, after blind review. Crafter-OO and OneLife's synthesis pipeline, on which our experiments build, are already public (Khan et al., 2026). The appendices describe the full details of rule compilation, planning, both posteriors, and every compared controller (Appendices B and D), and give all prompts (Appendix E) and proofs (Appendix A).

REFERENCES

Zergham Ahmed, Joshua B. Tenenbaum, Chris Bates, and Samuel J. Gershman. Synthesizing world models for bilevel planning. *Transactions on Machine Learning Research*, 2025. ISSN 2835-8856. URL https://openreview.net/forum?id=m9V4JHLJrD.

Dilip Arumugam and Thomas L. Griffiths. Toward efficient exploration by large language model agents. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=M3vwnscpL2.

D.P. Bertsekas. *Dynamic Programming and Optimal Control*, volume 1 and 2. Athena Scientific, 2 edition, 2001.

Kathryn Chaloner and Isabella Verdinelli. Bayesian experimental design: A review. *Statistical Science*, 10(3):273–304, 1995. ISSN 08834237. URL http://www.jstor.org/stable/2246015.

Minghao Chen, Yihang Li, Yanting Yang, Shiyu Yu, Binbin Lin, and Xiaofei He. Automanual: Constructing instruction manuals by llm agents via interactive environmental learning. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 589–631. Curran Associates, Inc., 2024. doi: 10.52202/079017-0019. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/0142921fad7ef9192bd87229cdafa9d4-Paper-Conference.pdf.

Nicola Dainese, Matteo Merler, Minttu Alakuijala, and Pekka Marttinen. Generating code world models with large language models guided by monte carlo tree search. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=9SpWvX9ykp.

Carlos Diuk, Andre Cohen, and Michael L. Littman. An object-oriented representation for efficient reinforcement learning. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, pp. 240–247, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605582054. doi: 10.1145/1390156.1390187. URL https://doi.org/10.1145/1390156.1390187.

M.O.G. Duff. *Optimal Learning: Computational Procedures for Bayes-adaptive Markov Decision Processes*. University of Massachusetts at Amherst, 2002. URL https://books.google.co.kr/books?id=YrmjYgEACAAJ.

Alexander A. Feldbaum. Dual control theory. I. *Automation and Remote Control*, 21(9):874–880, 1960.

Richard E. Fikes and Nils J. Nilsson. Strips: a new approach to the application of theorem proving to problem solving. In *Proceedings of the 2nd International Joint Conference on Artificial Intelligence*, IJCAI'71, pp. 608–620, San Francisco, CA, USA, 1971. Morgan Kaufmann Publishers Inc.

Lin Guan, Karthik Valmeekam, Sarath Sreedharan, and Subbarao Kambhampati. Leveraging pretrained large language models to construct and utilize world models for model-based task planning. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=zDbsSscmuj.

Arthur Guez, David Silver, and Peter Dayan. Efficient bayes-adaptive reinforcement learning using sample-based search. In F. Pereira, C.J. Burges, L. Bottou, and K. Weinberger (eds.), *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012. URL https://proceedings.neurips.cc/paper_files/paper/2012/file/35051070e572e47d2c26c241ab88307f-Paper.pdf.

David Ha and Jürgen Schmidhuber. Recurrent world models facilitate policy evolution. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018. URL https://proceedings.neurips.cc/paper_files/paper/2018/file/2de5d16682c3c35007e4e92982f1a2ba-Paper.pdf.

Danijar Hafner. Benchmarking the spectrum of agent capabilities. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=1W0z96MFEoH.

Danijar Hafner, Timothy Lillicrap, Jimmy Ba, and Mohammad Norouzi. Dream to control: Learning behaviors by latent imagination. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=S1lOTC4tDS.

Ronald A. Howard. Information value theory. *IEEE Transactions on Systems Science and Cybernetics*, 2(1):22–26, 1966. doi: 10.1109/TSSC.1966.300074.

Ken Kansky, Tom Silver, David A. Mély, Mohamed Eldawy, Miguel Lázaro-Gredilla, Xinghua Lou, Nimrod Dorfman, Szymon Sidor, Scott Phoenix, and Dileep George. Schema networks: Zero-shot transfer with a generative causal model of intuitive physics. In Doina Precup and Yee Whye Teh (eds.), *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pp. 1809–1818. PMLR, 06–11 Aug 2017. URL https://proceedings.mlr.press/v70/kansky17a.html.

Zaid Khan, Archiki Prasad, Elias Stengel-Eskin, Jaemin Cho, and Mohit Bansal. One life to learn: Inferring symbolic world models for stochastic environments from unguided exploration. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=UQ36IrVCw2.

Khimya Khetarpal, Zafarali Ahmed, Gheorghe Comanici, and Doina Precup. Temporally abstract partial models. In A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan (eds.), *Advances in Neural Information Processing Systems*, 2021. URL https://openreview.net/forum?id=LGvlCcMgWqb.

D. V. Lindley. On a Measure of the Information Provided by an Experiment. *The Annals of Mathematical Statistics*, 27(4):986 – 1005, 1956. doi: 10.1214/aoms/1177728069. URL https://doi.org/10.1214/aoms/1177728069.

Ian Osband and Benjamin Van Roy. Near-optimal reinforcement learning in factored mdps. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Weinberger (eds.), *Advances in Neural Information Processing Systems*, volume 27. Curran Associates, Inc., 2014. URL https://proceedings.neurips.cc/paper_files/paper/2014/file/0f0b653ef2261da4d9655441deb6cc55-Paper.pdf.

Ian Osband and Benjamin Van Roy. Why is posterior sampling better than optimism for reinforcement learning? In Doina Precup and Yee Whye Teh (eds.), *Proceedings of the 34th International*

Conference on Machine Learning, volume 70 of *Proceedings of Machine Learning Research*, pp. 2701–2710. PMLR, 06–11 Aug 2017. URL https://proceedings.mlr.press/v70/osband17a.html.

Ian Osband, Daniel Russo, and Benjamin Van Roy. (more) efficient reinforcement learning via posterior sampling. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'13, pp. 3003–3011, Red Hook, NY, USA, 2013. Curran Associates Inc.

Hanna M. Pasula, Luke S. Zettlemoyer, and Leslie Pack Kaelbling. Learning symbolic models of stochastic domains. *J. Artif. Int. Res.*, 29(1):309–352, July 2007. ISSN 1076-9757.

Wasu Top Piriyakulkij, Cassidy Langenfeld, Tuan Le, and Kevin Ellis. Doing experiments and revising rules with natural language and probabilistic reasoning. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 53102–53137. Curran Associates, Inc., 2024. doi: 10.52202/079017-1682. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/5f1b350fc0c2affd56f465faa36be343-Paper-Conference.pdf.

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 *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=obwRcksFZw.

Daniel Russo and Benjamin Van Roy. Learning to optimize via information-directed sampling. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Weinberger (eds.), *Advances in Neural Information Processing Systems*, volume 27. Curran Associates, Inc., 2014. URL https://proceedings.neurips.cc/paper_files/paper/2014/file/90720a2fcc41f9332e6a1558da327089-Paper.pdf.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/1b44b878bb782e6954cd888628510e90-Paper-Conference.pdf.

Malcolm J. A. Strens. A bayesian framework for reinforcement learning. In *Proceedings of the Seventeenth International Conference on Machine Learning*, ICML '00, pp. 943–950, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc. ISBN 1558607072.

Erik Talvitie and Satinder Singh. Simple local models for complex dynamical systems. In D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou (eds.), *Advances in Neural Information Processing Systems*, volume 21. Curran Associates, Inc., 2008. URL https://proceedings.neurips.cc/paper_files/paper/2008/file/f76a89f0cb91bc419542ce9fa43902dc-Paper.pdf.

Hao Tang, Darren Yan Key, and Kevin Ellis. Worldcoder, a model-based LLM agent: Building world models by writing code and interacting with the environment. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=QGJSXMhVaL.

William R. Thompson. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3/4):285–294, 1933. ISSN 00063444. URL http://www.jstor.org/stable/2332286.

Pedro Tsividis, Joao Loula, Jake Burga, Juan Pablo Rodriguez, Sergio Arnaud, Nate Foss, Andres Campero, Ajay Subramanian, Thomas Pouncy, Samuel J. Gershman, and Joshua B. Tenenbaum. Human-level learning of complex novel tasks as theory-based modelling, exploration and planning. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 384(2320):20240529, 05 2026. ISSN 1364-503X. doi: 10.1098/rsta.2024.0529. URL https://doi.org/10.1098/rsta.2024.0529.

Yue Wu, So Yeon Min, Shrimai Prabhumoye, Yonatan Bisk, Russ R Salakhutdinov, Amos Azaria, Tom M Mitchell, and Yuanzhi Li. Spring: Studying papers and reasoning to play games. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 22383–22687. Curran Associates, Inc., 2023. doi: 10.52202/075280-0983. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/46c2a9a6f2b2be68682013eb1173c801-Paper-Conference.pdf.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

Siyu Zhou, Tianyi Zhou, Yijun Yang, Guodong Long, Deheng Ye, Jing Jiang, and Chengqi Zhang. WALL-e: World alignment by neurosymbolic learning improves world model-based LLM agents. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=DorAT49sxj.

A PROOFS

We prove the two formal results of §2, Theorem 1 on the exact controller (Appendix A.1) and Proposition 1 on the belief updates of ProPS (Appendix A.2).

A.1 PROOF OF THEOREM 1

We use the notation of §2.1, with situations $s = (z, \beta, m)$. A policy π may choose each action from the whole history and may randomize. Its value $V^\pi(s) = \big(p^\pi(s), -e^\pi(s)\big)$ pairs the probability $p^\pi(s)$ that it ends in a successful commitment with the expected number of actions $e^\pi(s)$ it spends. Values are compared lexicographically as in Equation 1, $(p, u) \succ (p', u')$ if $p > p'$, or if $p = p'$ and $u > u'$. We write $\mathcal{V}^*(s)$ for the best value any policy attains from s, and $\mathcal{V}(s)$ for the value of Equation 1.

One property of this order is used twice. If $V_i \preceq W_i$ for every i and the weights $w_i \geq 0$ sum to one, then $\sum_i w_i V_i \preceq \sum_i w_i W_i$, because the first coordinates decide unless they tie on every term of positive weight, and then the second coordinates decide. Taking every W_i to be $\max_{\succ,j} V_j$ shows that an average of values is never better than the best of them.

Proof of part (i), optimality. Every action costs at least one unit of budget, so a run from s takes at most m actions and the situations reachable from s form a finite tree. Under the assumptions of the theorem, the outcome probabilities $\Pr_\beta(y \mid z, a) = \sum_{h \in \mathcal{H}} \beta(h) \Pr(y \mid z, a, h)$, the updated belief $\beta_y(h) \propto \beta(h) \Pr(y \mid z, a, h)$, the next task state z_y, and the remaining budget $m - c(a)$ depend on the history only through (z, β, m), the action, and its outcome. A policy therefore loses nothing by choosing from the current situation, and we argue by induction on m.

If $m = 0$, or no diagnostic fits in the budget, the only available action is to commit, so $\mathcal{V}(s) = \mathcal{Q}_{\text{commit}}(s) = \mathcal{V}^*(s)$. Now let $m \geq 1$, and assume $\mathcal{V} = \mathcal{V}^*$ at every situation with budget below m. A policy π that runs diagnostic a first continues after outcome y with some policy π_y from $s_y = (z_y, \beta_y, m - c(a))$, so

$$V^\pi(s) = \sum_y \Pr_\beta(y \mid z, a)\, V^{\pi_y}(s_y) \;+\; \big(0, -c(a)\big) \;\preceq\; \mathcal{Q}_a(s)$$

by the induction hypothesis, $V^{\pi_y}(s_y) \preceq \mathcal{V}(s_y)$, and the property above. Running a and then continuing optimally after each outcome attains this bound, and committing at once attains $\mathcal{Q}_{\text{commit}}(s)$. Every policy starts with some available action, and randomizing that choice yields an average that is never better than its best term, so

$$\mathcal{V}^*(s) = \operatorname{lexmax}\big\{\, \mathcal{Q}_{\text{commit}}(s),\ \mathcal{Q}_a(s) : a \in \mathcal{A}_{\text{diag}}(z, m) \big\} = \mathcal{V}(s).$$

Breaking ties by action cost in Equation 1 only chooses among actions with the same probability of success, so it cannot lower that probability. □

Proof of part (ii), separation. Fix $K \geq 2$ and $N > K$. The hidden program is $\theta = (r, n)$, uniform on $\{1, \ldots, K\} \times \{1, \ldots, N\}$, so $H(\theta) = \log(KN)$. The budget is $m = 2$ and every action costs one. Diagnostic a_r reveals r, diagnostic a_n reveals n, and commitment c_j, for $j \leq K$, succeeds exactly when $j = r$. The problem is deterministic and its tests cost the same.

Because r and n are independent under the prior, the information gains are $\mathrm{IG}(a_r) = \log(KN) - \log N = \log K$ and $\mathrm{IG}(a_n) = \log(KN) - \log K = \log N$. The costs are equal and $N > K$, so cost-aware greedy information gain runs a_n. The belief about r then stays uniform, and the agent must commit with its last action, succeeding with probability $1/K$.

By part (i), the exact policy values each first action by its best continuation. After a_r, committing to c_r succeeds with probability one, whereas running a_n first or committing at once succeeds with probability $1/K < 1$. The exact policy therefore runs a_r and always succeeds, and the gap $1 - 1/K$ tends to one as K grows. Information gain (Lindley, 1956) thus tests the nuisance coordinate n, which has the higher entropy, whereas valuing a test by the decision it improves (Howard, 1966) tests the coordinate r that the commitment requires. □

A.2 Scoped, non-deleting revision

We now state and prove the properties of the updates of §2.2 that the main text relies on. An action that did not run changes no rule weight, an action that ran changes only its own factor, and no rule ever loses all of its weight. We call the extra step of §2.2, which renormalizes a whole factor when every applicable rule misses, *confirmation*.

Proposition 1 (Scoped, non-deleting revision)**.** *Assume that the variables in each rule's write scope are observed, that the adapter reports correctly whether each action ran, and that every rule starts with positive weight. Under the updates of §2.2:* (i) *an action that did not run changes no rule weight and no outcome count;* (ii) *an action that ran changes only its own factor;* (iii) *a rule whose guard fails keeps its outcome counts, and its weight changes only under confirmation;* (iv) *if nothing else writes a rule's scope, a rule whose clauses list every change its action can make on that scope matches one of those clauses whenever its guard holds; and* (v) *every rule keeps positive weight after any finite sequence of updates.*

We use the notation of §2.2, with $q_h = q(\kappa_h, \cdot)$ and $\tau_h = \tau(y_t; \kappa_h)$. The rules of the executed action's factor $\mathcal{H}_{o_t}$ whose guards fail at x_t *abstain*. Confirmation replaces the ordinary update, Equation 6, when $\mathcal{A}_t$ is nonempty, every applicable rule decodes to the residual, and some rule abstained. It then renormalizes the whole factor,

$$b_{t+1}(h) = \frac{b_t(h)\, w_h}{\sum_{h' \in \mathcal{H}_{o_t}} b_t(h')\, w_{h'}}, \qquad w_h = \begin{cases} q_h(\emptyset), & h \in \mathcal{A}_t, \\ 1, & \text{otherwise.} \end{cases} \tag{7}$$

The multiplier 1 for an abstaining rule is a convention. It treats the outcome as fully consistent with a rule whose guard excluded the state, and it is not a likelihood computed from the observation. Under either update, each applicable rule increments the count of its decoded outcome, and the abstaining rules keep their counts.

Proof of Proposition 1. Every predictive probability is positive, $q_h(c) \geq 1/\big(|\mathcal{O}_{\kappa_h}| + 1 + \sum_{c'} n_{\kappa_h}(c')\big) > 0$, because the pseudocount 1 in Equation 2 keeps the numerator at least one. Observing the scope variables makes every decode well defined, and a correct report of whether the action ran makes the cases below exhaustive.

(i) An action that did not run skips the belief update (Algorithm 1), so no outcome count changes, the availability counts being kept separately, and $b_{t+1} = b_t$.

(ii) A rule of another factor is never applicable, so both updates leave its counts and its weight unchanged.

(iii) Let h abstain. It decodes nothing, so its counts are unchanged. If no rule is applicable, or some applicable rule decodes to a clause, the ordinary update applies. It leaves $b_t(h)$ unchanged and keeps the total weight of the applicable rules at B_t, since $\sum_{h' \in \mathcal{A}_t} b_t(h')\, q_{h'}(\tau_{h'})\, B_t / L_t = B_t$. If every applicable rule decodes to the residual, confirmation applies with multipliers $q_{h'}(\emptyset) \in (0, 1)$ for the applicable rules and 1 for the abstaining ones, so the abstaining rules gain weight in proportion to

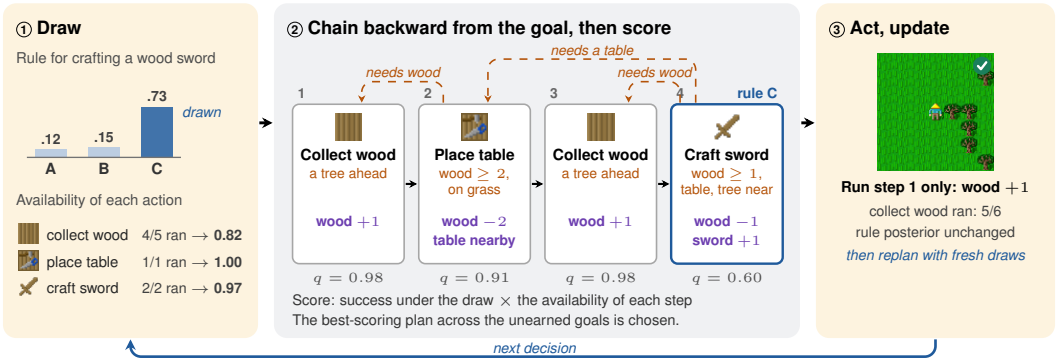


Figure 3: How ProPS plans, at one decision of a recorded episode. (1) It draws one rule per action from the rule posterior, here rule C for crafting a wood sword, and one availability per action from its counts. (2) Under the drawn rules it chains backward from the goal, so that each unmet requirement becomes a preceding step, and it scores the plan by its success under the sample times the availability of each step. (3) It runs only the first step, updates both posteriors from the outcome, and replans at the next decision with fresh samples. The frame is replayed from the episode (creatures are not drawn).

their previous weights and the applicable rules lose it. This describes how weight moves, and says nothing about whether an abstaining rule's guard is correct.

(iv) If nothing else writes the scope of h, the clauses of h list every change its action can make on it, and the guard of h holds when the action runs, the observed change $\Pi_{S_{\kappa_h}} y_t$ matches one of those clauses, so $\tau_h \neq \emptyset$ by Equation 5.

(v) Under the ordinary update, an applicable rule is multiplied by $q_h(\tau_h) > 0$ and by B_t/L_t, which is positive because $B_t \geq b_t(h) > 0$ and L_t is a finite sum of positive terms. Under confirmation, every rule of the factor is multiplied by a positive w_h and divided by a finite positive normalizer. All other weights are unchanged. By induction from positive initial weights, every rule keeps positive weight after any finite sequence of updates, in exact arithmetic. □

B Implementation

We describe how ProPS is implemented in the order of its loop, starting with the loop itself and its confirmation step (Appendix B.1), then how laws are compiled into rules (Appendix B.2) and the conditions these rules satisfy (Appendix B.3), and finally how plans are built and executed through the action adapter that every controller shares (Appendix B.4).

B.1 The ProPS loop

Algorithm 1 states the loop of §2.2 in full, and Figure 3 shows one decision of a recorded episode. Availability counts are kept per semantic action and start at zero in each target episode. A decision counts as an attempt of the action its plan starts with, and as a success when the adapter reports that the action ran.

Confirmation. The ordinary update, Equation 6, changes only the rules whose guards hold, so a rule whose narrower guard correctly excluded a state gains nothing from that state. Confirmation, Equation 7, moves weight to such rules when every applicable rule misses, with a multiplier that is a convention (Appendix A.2). Without it, the rule posterior attains 5.71 achievements rather than 5.96 (Appendix G.4).

B.2 Compilation details

Inputs and outputs. The compiler reads only the source trajectories, the code of the synthesized laws, and their fitted weights. The adapter supplies the object schema, the semantic actions, and

Algorithm 1 The ProPS loop: sampled planning, execution, and the two belief updates. The search maximizes Equation 4.

Require: Rules $\mathcal{H}_0$ compiled from the source episodes, with weights b and outcome counts from source replay; decision budget B

Initialize the map memory and $(n_o, f_o) = (0, 0)$ for each action o
for $t = 0, \ldots, B-1$ while the episode lasts **do**
  Observe x_t; update the map memory; set $m = B - t$
  Draw one rule $h_f \sim b_f$ for every factor f, forming $\tilde{b}$
  Draw $\rho_o \sim \mathrm{Beta}(1 + f_o, 1 + n_o - f_o)$ for every action o
  Chain rules backward from each unearned goal into the plan set Π_t, using at most one rule per factor
  if $\Pi_t = \emptyset$ or $\max_{\pi \in \Pi_t} J_{\mathrm{comp}}(\pi; \tilde{b}, \rho, m) \leq 0$ **then**
    Explore toward unseen tiles (Appendix B.4); **continue**
  end if
  $\pi_t \leftarrow \arg\max_{\pi \in \Pi_t} J_{\mathrm{comp}}(\pi; \tilde{b}, \rho, m)$, breaking ties by fewer expected attempts, then a fixed order
  Execute the first action o_t of π_t through the adapter; receive the receipt y_t and whether the action ran, f_t
  $n_{o_t} \leftarrow n_{o_t} + 1$; $f_{o_t} \leftarrow f_{o_t} + f_t$
  if $f_t = 1$ **then**
    $\mathcal{A}_t \leftarrow$ the rules of o_t's factor whose guards hold at x_t; if empty, **continue**
    $\tau_h \leftarrow \tau(y_t; \kappa_h)$ for each $h \in \mathcal{A}_t$ (Equation 5)
    Update b by Equation 6, or by confirmation (Equation 7) when every $\tau_h = \emptyset$ and some rule of the factor abstained
    $n_{\kappa_h}(\tau_h) \leftarrow n_{\kappa_h}(\tau_h) + 1$ for each $h \in \mathcal{A}_t$
  end if
end for

which coordinates mark an entity as present, and no recipes, damage values, or rewards. For Crafter-OO the state holds the inventory counts, achievements, vitals, and the typed objects in the local 9×9 view. A guard is a conjunction of tests, each comparing one coordinate with a value by $=, \neq, \geq, >, \leq, <, \in$ or $\notin$, possibly with alternative terms. Collecting compiles to an inventory change, crafting to consuming items and producing one under a guard, placing to a guarded change of the faced tile, and a first achievement to an effect inside an ordinary outcome clause. Table 3 lists what each of the eight source episodes contributes, and Appendix C shows compiled rules next to the laws they come from.

Procedure. For each distinct change that an action produced in a source episode, the observed change proposes the rule's outcome, and the laws must certify it (§2.2). For each law with positive weight that names the action, the compiler (1) reads the action and the typed fields from the law's precondition and effect, (2) binds the material or entity the agent faced, (3) runs the law on a copy of the recorded state, (4) compares every predicted coordinate the controller can see with the recorded next state, and (5) adds laws greedily, each time the one that reproduces the most coordinates still unexplained, until together they reproduce every changed inventory item and target coordinate (Algorithm 2). A change that no set of laws reproduces yields no rule.

Induced context. A rule's guard requires the items its change consumes and the inventory context its laws state, and adds context that the compiler induces from one source episode's attempts of the action. There, the successes are the attempts that produced the rule's change and the failures are the attempts that changed nothing. A non-walkable material within one tile of the agent joins the guard if it was nearby at every success and absent at some failure, and of these materials the compiler keeps those absent at the most failures, ties included. For an interaction, it instead adds the one tool requirement, a minimum count of a pickaxe or a sword, that every success met and that rules out the most failures. For a crafting action, materials that a crafting law's precondition names and that were adjacent at every success replace the induced ones. An episode without failures therefore adds no context, and episodes whose failures differ yield rival rules for one action (Figure 2b).

Identity and counts. Two rules are the same when they share the action, the exact guard, the write scope, and the effect, whatever the law's class name, source episode, or text. Source successes and attempts are counted once per source transition before they initialize the outcome counts of

Algorithm 2 Compiling synthesized laws into rules. Each source episode is compiled on its own (joint induction pools the attempts of all eight), and the rules of all episodes are then merged.

Require: Source episodes $D^{(1:k)}$, each with its recorded transitions and its synthesized laws with fitted weights

for each source episode $D^{(i)}$ **do**
  for each action o and each distinct change Δ of o in $D^{(i)}$ that gains a non-vital item, damages or removes a target, or unlocks an achievement **do**
    $S \leftarrow$ the attempts of o that produced Δ; $F \leftarrow$ the attempts of o that changed nothing
    $R \leftarrow$ the inventory items and target coordinates that Δ changes, or its achievements if it changes nothing else
    for each $s \in S$ and each law ℓ of positive weight whose code names o **do**
      Bind the material or entity faced at s; if ℓ's precondition holds, run ℓ on a copy of the recorded state, and let C_ℓ be the changed coordinates it predicts correctly
    end for
    $L \leftarrow \emptyset$; while a coordinate of R is unexplained, add to L the law whose C_ℓ explains the most of them (ties by higher weight, then by more inventory fields read)
    if a coordinate of R stays unexplained **then**
      continue (no rule for Δ)
    end if
    $g \leftarrow$ the items Δ consumes, the inventory context the preconditions in L state, and the target health values they admit
    if o interacts with an entity **then**
      Add to g the tool threshold met by all of S that rules out the most attempts in F
    else
      Add to g the non-walkable materials within one tile at every attempt in S and absent at the most attempts in F, ties kept; for a crafting action, materials that a crafting law's precondition names and that were adjacent at every attempt in S replace them; for a placement, require a faced material seen in S
    end if
    Emit a rule (o, g, Δ) for the physical change, if any, and one rule per achievement that an attempt in S unlocked for the first time
  end for
end for
Add to every guard a test that each entity the rule reads or writes is present
Merge rules that share the action, guard, write scope, and effect, and group the rules of each action into its factor
Initialize the outcome counts from source successes and attempts, each counted once per source transition

Equation 2, so two synthesized classes that describe the same transaction cannot vote twice. Each rule records its source episode, laws, supporting transitions, and the coordinates it covers.

Achievements. The compiler adds an achievement to a rule's outcome only if some source transition with the same action and effect first unlocked that achievement, and the goals the planner pursues are exactly the achievements declared this way. A final check requires every coordinate the planner uses to come from a law, and every declared achievement to belong to exactly one outcome clause.

Absent entities. A rule can mention an entity that is gone when the action runs. The adapter declares which coordinate marks an entity as present (for Crafter-OO, `entity.{target}.exists`), and the compiler adds to each rule's guard a test that every entity the rule reads or writes is present. Interacting with an entity that is gone decodes to the residual with no change.

One added coordinate. Rules can chain only if guards and effects use the same coordinates (§2.2). Our compiler's guard induction reads the nearby materials, so if its effect extraction did not compare them, a rule could require a nearby table while no rule could produce one, and no craftable item could ever be reached. We therefore add the nearby materials to the coordinates that effect extraction compares. The addition names no station, recipe, or action. The effect it exposes was already observed, since 31 of the 40 table placements attempted with two wood in the source episodes put a table within the guard's radius. OneLife's own code, including its observable extractor, is unchanged. Compiling with or without a hand-written mapping from `place_X` to a station X yields identical libraries.

Table 3: What each source episode contributes. Laws counts the fitted OneLife laws, of which Positive have positive weight. Records counts the executable law records admitted to the portfolio, and Matched the source-trace records matched to them, after two records were rejected for reading coordinates outside the shared vocabulary. No target-world outcome enters this table.

Source episode	Steps	Laws	Positive	Records	Matched
0	146	30	30	12	12
1	600	70	56	25	28
2	187	81	76	20	21
3	357	48	45	15	21
4	168	60	54	23	22
5	232	57	48	18	22
6	183	65	55	24	23
7	192	73	68	24	22
Total	2,065	484	432	161	171

B.3 Conditions on the compiled rules

The compiler enforces five conditions that let rules serve as planning operators.

(R1) Rules with the same compiled form are one rule, with one record of evidence.

(R2) Every observed outcome decodes to one of a rule's clauses or to its residual.

(R3) Evidence changes only the rules that made a prediction, which Equation 6 ensures by keeping the total weight of the applicable rules.

(R4) Every declared effect yields a defined next state, so backward chaining can compose rules into plans.

(R5) Guards and effects use one vocabulary of coordinates, so one rule's effect can satisfy another's guard (Appendix B.2).

B.4 PLANNING, EXPLORATION, AND THE ACTION ADAPTER

Building plans. The planner indexes the rules by their declared effects and chains backward from each unearned goal. A guard that does not yet hold becomes an earlier step if some rule can produce it, becomes a task for the adapter if it names the action's own target, such as the tree that collecting wood needs, and otherwise rejects the plan. The planner adds the items a plan consumes, one complete alternative of any guard with alternatives, and the nearby stations it needs. It rejects cycles, plans longer than twelve steps, and plans that use two rules of one factor, since such a plan describes no possible world. A rule that removes an entity applies only at a health value at which the source episodes saw the entity removed, and at other values the planner can deal damage first and replan from the new observation.

Scoring plans. Step i of a plan of L steps succeeds with probability q_i, the predictive probability (Equation 2) of the clause the step requires. The retry surrogate assumes that a failed step is retried at the same q_i until the budget runs out, which gives the completion probability

$$\mathrm{Succ}(i, m) = q_i \,\mathrm{Succ}(i+1, m-1) + (1 - q_i)\,\mathrm{Succ}(i, m-1),$$

with $\mathrm{Succ}(L+1, m) = 1$ for every m and $\mathrm{Succ}(i, 0) = 0$ for $i \le L$. Equation 3 multiplies $\mathrm{Succ}(1, m)$ by the weights of the rules the plan names.

Exploration. When the first step of the chosen plan cannot run yet, the controller moves toward the visible entity or station it needs. When no plan scores above zero, it explores with a fixed frontier policy over its memory of the map, which holds the tiles and entities seen in earlier observations, visit counts, and the agent's displacement inferred from its own moves. It has no access to absolute coordinates or to the simulator's path planner.

The action adapter. We supply the semantic actions rather than learn them. Each is one of four kinds: explore, interact with a visible material or entity, craft a named item, or place a named item. For Crafter-OO, the adapter computes from each new 9×9 observation a shortest path over the visible walkable tiles to a tile next to the target, then faces the target, and each primitive action calls Crafter's own step function once. The adapter reports that an action ran only when the intended interaction, craft, or placement took place, so failed navigation or targeting reports that it did not run and cannot change a rule's weight. The controller sees only the observation before and after.

Every controller in Table 2, the LLM agents included, reaches Crafter through this adapter, with the same pathfinding and the same cap of one to nine primitive steps per action.

Why the actions are supplied. The synthesized laws describe the outcomes of interactions. Each of the 161 law records concerns one interact (89), craft (50), or place (22) action, and none states how to reach a tile where the interaction can take place. Planning over Crafter's seventeen primitive actions would therefore need laws about movement, which this synthesis does not produce. We supply the adapter, hold it fixed across every controller, and claim nothing for it.

C EXAMPLES OF COMPILED RULES

We show compiled rules from the portfolio of §3.2 next to the laws they come from.

Box C.1 shows the law `PlayerPlaceTableWoodCost` exactly as synthesized in source episode 0, with OneLife weight 0.500, and below it the record compiled from it in that episode, trimmed to the fields the planner and the update read. The law predicts only that placing a table costs two wood. The record keeps the law's condition on wood and adds the context the compiler induces from the episode's attempts (Appendix B.2). The agent faced grass at the one successful placement, and a tree was nearby then but absent at failed attempts, so the tree enters the guard although placing a table does not need one. The effect is the whole observed change, in which the law certifies the wood while the nearby table and the achievement are observed in the same transitions. The counts are this episode's, one success in four attempts.

Box C.1 Place Table Law and Its Compiled Rule

```python
class PlayerPlaceTableWoodCost:
    def __init__(self):
        """Initialize with configurable parameters."""
        pass

    def precondition(self, current_state: WorldState, action: str) -> bool:
        """Return True if this law should apply to the given state and action."""
        return action == "place_table" \
            and current_state.player.inventory.wood >= 2

    def effect(self, current_state: WorldState, action: str) -> None:
        """Apply the law by modifying the world state."""
        current_state.player.inventory.wood = DiscreteDistribution(
            support=[current_state.player.inventory.wood - 2]
        )
```


```
{"key": ["place", "place_table", null, null],
 "inventory_guards": [{"item": "wood", "minimum": 2}],
 "nearby_materials": ["tree"],
 "placement_front_materials": ["grass"],
 "effect": {"inventory_delta": {"wood": -2},
            "nearby_materials_add": ["table"],
            "achievements_add": ["place_table"]},
 "support": 1, "attempts": 4, "eligible_attempts": 1,
 "primitive_cap": 1,
 "law_provenance": [{"law_name": "PlayerPlaceTableWoodCost",
                     "law_index": 8, "weight": 0.49999937415122986,
                     "fields": ["inventory.wood"]}]}
```


Table 4 shows how source replay weighs three rival rules for placing a table.

Table 4: Three of the four rules for placing a table, which form one factor. Counts give the source transitions decoded to the rule's explicit clause, to another clause, and to the residual $\emptyset$. q is the predictive probability of the explicit clause (Equation 2), and Weight is the rule's weight after replaying the source transitions from a uniform start (Equation 6). E denotes the shared effect, in which wood decreases by two, a table appears nearby, and the achievement is recorded. The broad guard of h_a decodes nine transitions to the residual and loses nearly all of its weight, while the narrower guards decode one. Each grid shows which faced tiles (grass, sand, stone) a guard accepts, with rejected tiles dimmed.

Rule	Guard	Faced tile	Explicit effect	Counts	q	Weight
h_a	wood ≥ 2		E	31/0/9	.74	.002
h_b	wood ≥ 2, faced grass		E	30/0/1	.91	.49
h_c	same as h_b		E; faced tile becomes table	30/0/1	.91	.49

Table 5 shows the rule for mining stone that each source episode compiles. The laws of five episodes require a pickaxe and those of three do not. An episode with failed attempts adds the pickaxe requirement that every success met (Appendix B.2), so episode 1 yields the correct guard from a law that states none. An episode without failures keeps its law's condition, so episodes 2 and 6 yield a rule under which stone can be mined by hand. Replay and the rule posterior then weigh these rival guards against each other.

Table 6 shows the rule for crafting a wood sword that each source episode compiles, the factor behind Figure 1b. Every law behind these rules states only the action and at most one wood, and

Table 5: The rule for mining stone compiled in each source episode. Law states gives the condition the synthesized law states besides facing stone, Attempts the successful and total attempts to mine stone in that episode, and Guard the compiled rule's guard besides facing stone. Episodes with failed attempts induce a wood-pickaxe requirement, and the others keep the law's condition.

Source episode	Law states	Attempts	Guard
0	any pickaxe	4 of 5	wood pickaxe ≥ 1
1	nothing	5 of 26	wood pickaxe ≥ 1
2	nothing	5 of 5	none
3	any pickaxe	10 of 10	any pickaxe
4	any pickaxe	3 of 4	wood pickaxe ≥ 1
5	any pickaxe	5 of 5	any pickaxe
6	nothing	3 of 3	none
7	any pickaxe	1 of 1	any pickaxe

none mentions a table or a tree. Episodes 1, 4, 5, and 6 crafted the sword at their only attempt, so induction has no failure to explain, and they compile rule A, which omits the table. In episode 2, a table and a tree were nearby at the one success and absent at both failures, which yields rule C, the only rule that requires the table. In episode 7, the one failure lacked wood, which the guard already requires, but a tree was nearby at the success and absent at the failure, so rule B requires a tree and no table. Episodes 0 and 3 crafted no sword and compile no rule. Rules with the same compiled form share one identity (Appendix B.2), so the factor holds three rules, A, B, and C.

Table 6: The rule for crafting a wood sword compiled in each source episode, the factor of Figure 1b. Law states gives the strongest condition the episode's laws state besides the action, Attempts the successful and total attempts to craft the sword in that episode, and Guard the compiled rule's guard. A dash marks an episode without a successful craft. Only episode 2's failures reveal the table.

Source episode	Law states	Attempts	Guard	Rule
0	—	0 of 0	no rule	—
1	wood ≥ 1	1 of 1	wood ≥ 1	A
2	wood ≥ 1	1 of 3	wood ≥ 1, table and tree nearby	C
3	—	0 of 2	no rule	—
4	wood ≥ 1	1 of 1	wood ≥ 1	A
5	wood ≥ 1	1 of 1	wood ≥ 1	A
6	wood ≥ 1	1 of 1	wood ≥ 1	A
7	wood ≥ 1	1 of 2	wood ≥ 1, tree nearby	B

Box C.2 shows the two laws that episode 6 synthesized for attacking a skeleton, shortened by removing comments, docstrings, and an empty constructor, and below them the records compiled from them. The damage law predicts that an attack takes two health from any skeleton, and the defeat law that a skeleton with one health is removed and the achievement recorded. Each law certifies the coordinates it predicts, so the two laws compile to two rules of one factor. The damage rule admits every health from 0 to 9, and the removal rule only health 1, the one value at which a removal was observed and the value the defeat law states. A third record adds the achievement to the removal (Algorithm 2). Both rules read the skeleton, so their guards require it to be present. Such a rule is correct yet cannot run while no skeleton is in view, the failure of Figure 1a that the availability posterior addresses.

Box C.2 Skeleton Laws and Their Compiled Rules

```
class SkeletonDamageFromPlayerAttack:
    def __init__(self, damage_amount: int = 2):
        self.damage_amount = damage_amount

    def precondition(self, current_state: WorldState, action: str) -> bool:
        if action != "do":
```

```
            return False
        target_material, target_entity = current_state.get_target_tile()
        if isinstance(target_entity, SkeletonState):
            return True
        return False

    def effect(self, current_state: WorldState, action: str) -> None:
        target_material, target_entity = current_state.get_target_tile()
        if isinstance(target_entity, SkeletonState):
            new_health = max(0, target_entity.health - self.damage_amount)
            target_entity.health = DiscreteDistribution(support=[new_health])

class SkeletonDefeatLaw:
    def precondition(self, current_state: WorldState, action: str) -> bool:
        if action != "do":
            return False
        target_tile_content = current_state.get_target_tile()
        if target_tile_content and target_tile_content[1] \
                and isinstance(target_tile_content[1], SkeletonState):
            targeted_skeleton: SkeletonState = target_tile_content[1]
            if targeted_skeleton.health == 1:
                return True
        return False

    def effect(self, current_state: WorldState, action: str) -> None:
        target_tile_content = current_state.get_target_tile()
        targeted_skeleton: SkeletonState = target_tile_content[1]
        current_state.objects = DiscreteDistribution(
            support=[[obj for obj in current_state.objects
                      if obj.entity_id != targeted_skeleton.entity_id]])
        current_state.player.achievements.defeat_skeleton = \
            DiscreteDistribution(
                support=[current_state.player.achievements.defeat_skeleton + 1])
```


```
{"key": ["interact", null, "skeleton", null],
 "target_health_values": [0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
 "effect": {"target_damage": 2},
 "support": 1, "attempts": 1, "eligible_attempts": 1,
 "law_provenance": [{"law_name": "SkeletonDamageFromPlayerAttack",
                     "law_index": 14, "weight": 0.49999889731407166,
                     "fields": ["target.health"]}]}
{"key": ["interact", null, "skeleton", null],
 "target_health_values": [1],
 "effect": {"target_removed": true},
 "support": 1, "attempts": 1, "eligible_attempts": 1,
 "law_provenance": [{"law_name": "SkeletonDefeatLaw",
                     "law_index": 60, "weight": 0.49999889731407166,
                     "fields": ["achievement.defeat_skeleton",
                                "target.exists", "target.health"]}]}
```


D EXPERIMENTAL SETUP

We describe the controlled setting (Appendix D.1), the open-ended setting (Appendix D.2), the controllers that use the compiled rules (Appendix D.3), and the LLM agents (Appendix D.4).

D.1 CONTROLLED SETTING

Each of the 48 worlds is a full procedural Crafter-OO world with one change, the damage each sword tier deals to a designated creature, which decides which of the three candidate programs is true. Each world also holds the stations the candidate plans need, a fixed set of resources, and two creatures with different roles. The designated creature, a zombie in 24 worlds and a skeleton in the

other 24, is the goal, and success requires the game's own achievement for defeating it. A cow that stays in place is a test target, since the three programs predict different damage when it is struck with a given sword tier. A diagnostic tests one sword tier: the agent crafts it, strikes the cow once, and reads the damage dealt, which rules out the programs that predicted otherwise. We supply the three programs and give every controller the same recipes, plans, action interface, navigation and combat routines, source transitions, and 48-action budget, so the rows compare how controllers use programs rather than how each system synthesizes them.

Controllers. *Random tier* commits to one of the three tiers uniformly at random. *WorldCoder fit* runs WorldCoder's released implementation, which keeps one accepted program and refines it after each wrong prediction (we do not use its reward synthesis, optimism constraint, exploration, or planner). The common controller reads the program's damage table. *PoE-World / OneLife fit* represents each candidate as an expert over the whole observation (PoE-World) or as a law that writes only the creature's health (OneLife), fits the weights with each release's maximum-likelihood fitter, commits to the mode of the fitted product, and refits when a new outcome could change a later decision. The two rows coincide in this deterministic setting. *Retain all, never test* keeps every program consistent with the evidence but never tests. It commits to the first program in the library and moves to the next only when an attack on the creature contradicts it. *ProPS, exact* solves Equation 1 by backward induction over the programs still consistent, the highest sword tier crafted, the world state, and the remaining budget. *Oracle* gives the true program to the same planner.

D.2 Open-ended setting

The open-ended setting uses unmodified Crafter-OO. OneLife's pipeline explores eight source episodes and synthesizes laws from them (Table 3), which yield 161 law records and 47 rules in 16 factors, and every controller receives the same records. Each controller then plays 48 target worlds whose seeds are disjoint from the source episodes, with 64 decisions and at most 2,000 primitive actions. The endpoint is the number of distinct achievements, of 22, earned in the episode, reported as a mean over the 48 worlds with its standard error, the standard deviation over the worlds divided by $\sqrt{48}$. Every controller sees the same 9×9 view, inventory, achievements, and vitals, acts through the adapter of Appendix B.4, and receives no reward. No controller is told what an action does. Each action appears as a record of its kind, its target, and its step cap, and only actions with a visible target are offered. The LLM agents may choose any action without a plan, including `place_stone`, `place_furnace`, `make_iron_pickaxe`, `sleep`, and `wait`, whereas ProPS plans toward the 13 goals its source episodes cover.

D.3 Planning-based methods

Every controller in this subsection uses the rules, planner, adapter, exploration, goals, and budgets of ProPS, makes no language-model call, and differs from ProPS only in the operation its name gives. A variant marked "+ availability posterior" adds ProPS's availability posterior unchanged.

Product of experts. As in OneLife and PoE-World (Khan et al., 2026; Piriyakulkij et al., 2025), this controller fits one weight θ_i per rule by maximizing the likelihood of the source transitions under the normalized product $P(y \mid s, a) \propto \prod_i p_i(y \mid s, a)^{\theta_i}$ over the rules whose guards hold. We use PoE-World's released settings, namely L-BFGS with strong-Wolfe line search, learning rate 1, four outer steps, weights initialized at 0.5 and clamped to $[0, 10]$, an L_1 penalty of 1, and removal of rules whose weight is at most 0.01. The fit keeps 41 of the 47 rules, or 37 of 41 under joint guard induction, and the weights stay fixed during control. Two choices favor this controller. It normalizes once per factor over the joint outcomes its rules name, where OneLife normalizes each observable separately, and its rules keep outcome counts that are updated after each action, so the predicted success of a step falls after a failure. It is thus the product of experts used inside our controller, and it reproduces neither OneLife, which evaluates no control, nor PoE-World, which synthesizes and refits while it acts.

Best-supported rule. This controller starts from the belief ProPS starts from, replays the same source transitions with the same update, keeps the rule with the highest weight in each factor, and holds these 16 rules fixed. OneLife's fitted weights decide which laws are compiled, namely the 432

with positive weight, but not which rule this controller keeps, because 237 of the 484 fitted weights never moved from their initial value of 0.5, their laws never having fired in their episode. ProPS and this controller therefore differ only in whether each factor collapses to one rule before the target episode.

Ablations. The *rule posterior* is ProPS without the availability posterior. − *Confirmation* drops the confirmation step of §2.2, and − *Both* drops it and the availability posterior. The variants of Table 11 start from the rule posterior without confirmation. *Marginal step support* scores each step by the total weight of the rules that predict it instead of drawing one rule per factor. *Full-belief assignment score* scores plans by Equation 3 under the full belief instead of a draw, and freezing that belief as well gives the same result in every world. − *Retention* keeps one rule per factor, and − *Replanning* executes the selected plan instead of replanning. *Deletion* removes a rule when it decodes an outcome to the residual, instead of down-weighting it, and closes a factor with no rule left.

Joint guard induction. The Joint column of Table 11 runs the induction of Appendix B.2 over the attempts of all eight source episodes together rather than one episode at a time. This yields 41 rules in 16 factors, and the three rules for crafting a wood sword collapse into the correct one. Eight of the sixteen factors still keep more than one rule, because their rules differ in what the language model wrote.

D.4 LLM AGENTS

We evaluate three ReAct agent variants with different memories. We use Gemini 2.5 Flash, called through LiteLLM at temperature 0 with low reasoning effort and at most 3,072 output tokens. Each decision prompt holds the observation, the available actions, the remaining budget, and the full transcript of the episode so far, and the reply is one action under a schema. An invalid reply is retried up to five times and then counts as a no-op. No-memory is the basic ReAct agent without extra memory. The law-memory variant receives the 161 law records as text, with guards, effects, source counts, and provenance, and in the Joint column of Table 11 it receives the records compiled with joint guard induction. The reflection-memory row receives eight reflections, which the same model writes from the transcripts of the eight source episodes, one each, with no code, fitted weights, or information about the target worlds. Because each reflection sees one episode, this variant bounds what per-episode verbal memory contributes rather than what every reflection architecture could. Prompts are provided in Appendix E.

E PROMPTS

We give the prompts of the LLM agents of Appendix D.4. Each decision prompt is one instruction followed by the current request as JSON (Box E.1). The instruction is the same for the three variants, except that the law-memory agent's adds instructions about its law records, and the request carries the law records only for that agent and the reflections only for the reflection agent. At the first decision the prompts hold 9,001, 20,524, and 184,059 characters for the agents with no memory, reflection memory, and law memory. With jointly induced guards, the law-memory agent receives the same instruction with the joint records. The reflections are written once, one per source episode, with the instruction of Box E.2, which also shows the reflection written from source episode 0. It states correct recipes next to lessons that are false in Crafter, such as losing health when moving between sand and grass.

The laws themselves come from OneLife's synthesis pipeline, whose synthesis prompt we use as released. OneLife's paper gives it in full, together with the prompt of its exploration policy (Khan et al., 2026, Appendix G).

Box E.1 ReAct Decision Prompt (Law Memory, First Decision) and Reply

```
You directly control one unmodified native Crafter-OO life. The request includes
`persistent_law_memory`, the exact executable-law option facts learned from eight
```

prior public source lives. They may be incomplete or mutually incompatible. Use them as evidence, infer any prerequisites yourself, and still choose one ordinary current operator. No program or action is selected for you. The request may include `persistent_verbal_reflections`, bounded post-episode natural-language memories from prior public source episodes under the same game dynamics. If present, they may be incomplete or wrong: use them as fallible advice, infer prerequisites yourself, and still choose one current operator. No executable law, program, projected state, or action is selected for you. Use the current public observation and complete in-episode ReAct transcript to earn unachieved game achievements. Return one JSON object with `reason`, a concise public decision rationale, and `operator_id`, exactly one current operator-table identifier. The selected semantic operator executes now and its real outcome appears before your next decision. Do not invent an identifier.
CURRENT_REQUEST={"action_attempt_index": 0, "control_policy": {...},
 "decision_index": 0,
 "objective":
   "Survive and unlock as many of the 22 Crafter achievements as possible.",
 "observation": {"achievements": {"collect_coal": 0, "collect_diamond": 0, ...},
   "daylight": 0.797, "facing": [0, 1],
   "inventory": {"coal": 0, "diamond": 0, "drink": 9, "energy": 9, ...},
   "local_entities": [{"attributes": {}, "dx": -4, "dy": 0, "entity_id": 22,
     "health": 3, "name": "cow"}],
   "local_tiles": [{"dx": -4, "dy": -4, "material": "grass"}, ...],
   "sleeping": false, "step": 0},
 "operator_table": [{"operator": {"action": "make_iron_pickaxe",
     "direction": null, "kind": "craft", "primitive_cap": 1, "repeat": 1,
     "target": null}, "operator_id": "op-10859deae5"}, ...],
 "persistent_law_memory": {
   "interpretation": "Prior public executable mechanism and event ...",
   "kind": "exact_pooled_executable_mechanism_event_memory_v1",
   "learned_options": [{"attempts": 22,
     "compiler_kind": "one_life_executable_mechanism_v1",
     "effect": {"inventory_delta": {"stone": -1, "stone_pickaxe": 1,
       "wood": -1}, ...},
     "inventory_guards": [{"item": "stone", "minimum": 1},
       {"item": "wood", "minimum": 1}],
     "key": ["craft", "make_stone_pickaxe", null, null],
     "nearby_materials": ["stone"], ...}, ...],
   "memory_hash": "d6d83da7...", "source_lives": 8},
 "proposal_index": 0, "protocol": "native-intent-ehc-v8-proposer",
 "react_memory": {"assistant_turn_count": 0, "assistant_turns": [],
   "policy": "complete_in_episode_obs_reason_act_transcript",
   "semantic_action_outcome_count": 0},
 "recent_semantic_history": [], "remaining_actions": 2000,
 "response_schema": {"additionalProperties": false,
   "properties": {"operator_id": {"enum": ["op-001ae716c6", ...],
       "type": "string"}, "reason": {"maxLength": 384, "type": "string"}},
   "required": ["reason", "operator_id"], "type": "object"},
 "semantic_history_policy": "complete", "semantic_history_summary": {...},
 "unachieved": ["collect_coal", "collect_diamond", ...]}

{"reason":"To begin progressing towards achievements, I will collect wood from a nearby tree. Wood is a fundamental resource for crafting tools and a crafting table, which are prerequisites for many other achievements.","operator_id":"op-d8d924b62c"}

Box E.2 Reflection Prompt and the Reflection of Source Episode 0

Write one post-episode verbal memory for an agent that will act in a new procedural episode governed by the same Crafter-OO mechanics. The source episode

```
was open-ended exploration, not an attempt at one fixed goal. Infer only what the
public transcript supports. Record useful action prerequisites and effects, failed
assumptions, stochastic or uncertain outcomes, and reusable strategy. Distinguish
one observation from repeated evidence; repeated failures are evidence too. Do not
invent recipes, probabilities, map facts, hidden state, executable code, or
actions for the future episode. Return exactly the strict JSON object requested.

SOURCE_REQUEST={...}
```

```
Collecting wood from a tree by using the 'Do' action is successful and grants 1
wood. Placing a table requires 2 wood and consumes them, granting the
'place_table' achievement. Crafting a wood pickaxe requires 1 wood and consumes
it, granting the 'make_wood_pickaxe' achievement. Mining stone with a wood pickaxe
using the 'Do' action is successful and grants 1 stone, also granting the
'collect_stone' achievement. Crafting a stone pickaxe requires 1 wood and 2 stone,
consuming them and granting the 'make_stone_pickaxe' achievement. Repeated
attempts to 'Make Stone Pickaxe' without the required materials (wood and stone)
or without being adjacent to a crafting table will fail. Repeated attempts to
'Place Table' without the required wood will fail. Repeated attempts to 'Do' on a
'path' or 'grass' tile do not yield resources. Repeated attempts to 'Move North'
into a 'table' tile (position [32,25]) from position [32,26] are unsuccessful,
indicating that a table acts as an impassable obstacle. Movement in other
directions (East, West, South) is successful even when North is blocked. Drinking
from 'water' using the 'Do' action increases the 'drink' vital by 1 and grants the
'collect_drink' achievement. Moving from 'sand' to 'grass' or vice versa can cause
health loss. Vitals (drink, energy, food, health) decrease over time or with
certain actions. Sleeping sets health to 0 and ends the episode.
```

F LLM USAGE AND COST

Table 7 lists every language-model call behind the open-ended results. All calls use Gemini 2.5 Flash, and costs are as billed, at \$0.30 and \$2.50 per million input and output tokens, with thinking tokens counted as output. ProPS calls the model only to build its portfolio, once for all target worlds, and makes no call while it acts. Its planning takes a median of 0.65 seconds per decision on a CPU, 25 seconds at the 95th percentile, and 4.0 hours for all 48 worlds. Each ReAct agent calls the model at every decision, and the law records give the law-memory agent about 7.6 times the input tokens of the agent without memory, which accounts for most of its cost.

Table 7: Language-model calls and their cost. The portfolio rows run once and serve every controller that uses the laws. The synthesis row includes 117 failed attempts, which were billed and retried, and its token counts cover the 373 that completed. The target-world rows cover the 48 worlds.

	Calls	Input tokens	Output tokens	Cost (USD)
Built once from the eight source episodes				
Exploration	2,066	7.8M	1.7M	6.53
Law synthesis	490	4.6M	1.6M	8.35
Reflections, for the reflection memory	8	0.66M	0.01M	0.22
While acting in the target worlds				
ProPS and the other planning controllers	0	–	–	0
ReAct (no memory)	2,960	27.1M	2.6M	14.62
ReAct + reflection memory	2,981	37.4M	2.7M	18.04
ReAct + law memory	3,064	205.0M	2.9M	68.66
ReAct + law memory, joint records	2,978	198.4M	2.8M	66.48

G Additional Experiments

We report further results for §3: the controlled results with confidence intervals (Appendix G.1), exact valuation and information gain across action budgets (Appendix G.2), a test with two creatures in which the goal decides which one matters (Appendix G.3), the full ablations (Appendix G.4), and what each part of the availability posterior contributes (Appendix G.5).

G.1 Controlled results with intervals

Table 8 adds 95% intervals to Table 1, and a second condition in which the source evidence identifies the program. There every controller with a model reaches the oracle, so the controllers differ only where the programs disagree. Under ambiguous evidence, ProPS exceeds the OneLife fit by 0.156 in success AUC (95% interval $[0.128, 0.184]$), higher in 42 of the 48 worlds, and the fits succeed about as often as committing to a random tier.

Table 8: Results in the controlled setting with 95% intervals from resampling worlds, over 144 instances per condition (48 worlds $\times$ three hidden programs). Actions is the mean over successful instances, with their number in parentheses, and Diag. counts diagnostics per instance. Random tier gives the exact expectation over its three choices.

Controller	Success	AUC	Actions	Diag.
Ambiguous source evidence				
No model, no plan attempted	.000 [.000, .000]	.000 [.000, .000]	— (0)	.000
Random tier	.285 [.259, .308]	.114 [.100, .129]	29.72 (41)	.000
WorldCoder fit	.000 [.000, .000]	.000 [.000, .000]	— (0)	.000
PoE-World fit	.299 [.264, .326]	.124 [.107, .139]	29.14 (43)	.000
OneLife fit	.299 [.264, .326]	.124 [.107, .139]	29.14 (43)	.000
Retain all, never test	.486 [.410, .562]	.187 [.151, .225]	30.54 (70)	.000
ProPS, exact (ours)	**.833** [.764, .896]	**.279** [.246, .311]	32.93 (120)	1.653
Oracle (true program)	.854 [.771, .924]	.343 [.298, .386]	29.72 (123)	.000
Identified source evidence				
No model, no plan attempted	.000 [.000, .000]	.000 [.000, .000]	— (0)	.000
Every controller with a model, and the oracle	.854 [.771, .924]	.343 [.298, .386]	29.72 (123)	.000

G.2 Exact valuation and information gain across budgets

A second bank of 48 worlds with the same three programs runs each policy at budgets of 24 to 48 actions (Table 9). Target feedback only never tests and learns only from its attacks on the creature. Random diagnosis tests a random tier that separates the remaining programs, and greedy information gain tests the tier with the largest expected information gain per action, the cost-aware criterion of §3.1. Random tests do not explain the gain, since averaged over budgets ProPS exceeds random diagnosis by 0.069 in success AUC. On this symmetric problem greedy information gain ties ProPS (difference 0.0004), since both test the lowest tier first. ProPS differs only in declining tests when the budget is short, which it does on 221 decisions.

G.3 Goal-crossed test

Each of 48 worlds holds two creatures whose damage is unknown, a zombie with the three programs of §3.1 and a skeleton with four, and every world and program pair is run once with each defeat goal (Table 10). Task-blind information gain probes the skeleton, which has more programs, whatever the goal, and succeeds on 0.266. ProPS probes the creature the goal names and succeeds on 0.819, higher in all 48 worlds. Information gain that is told which creature the goal concerns reaches 0.816, so a correct relevance scope largely substitutes for exact valuation here.

Table 9: Results in the controlled setting across action budgets, over 144 instances per row. Diag. counts diagnostics per instance.

Budget	Policy	Success	AUC	Diag.
24	Target feedback only	.069	.007	.00
	Random diagnostic	.035	.004	1.26
	Greedy information gain	.049	.005	1.67
	ProPS, exact (ours)	.069	.007	.02
32	Target feedback only	.243	.053	.00
	Random diagnostic	.167	.028	1.31
	Greedy information gain	.458	.076	1.67
	ProPS, exact (ours)	.424	.078	1.21
40	Target feedback only	.306	.098	.00
	Random diagnostic	.312	.080	1.31
	Greedy information gain	.757	.186	1.67
	ProPS, exact (ours)	.736	.184	1.61
48	Target feedback only	.361	.138	.00
	Random diagnostic	.542	.173	1.31
	Greedy information gain	.861	.291	1.67
	ProPS, exact (ours)	.861	.291	1.67

Table 10: Results of the goal-crossed test, with the same worlds and beliefs under both defeat goals (1,152 runs per row). The Zombie and Skeleton columns give the success rate under each goal.

Policy	Success	Zombie	Skeleton	AUC
Target feedback only	.435	.417	.453	.121
Random diagnostic	.216	.293	.139	.033
Task-blind information gain	.266	.444	.089	.041
Information gain told the goal's creature	.816	.903	.729	.158
ProPS, exact (ours)	**.819**	**.910**	**.729**	**.159**

G.4 Ablations

Table 11 gives every ablation and variant of ProPS defined in Appendix D.3, the rule baselines with and without the availability posterior, and ReAct + law memory, under both guard inductions. Without confirmation, drawing one rule per factor (5.71) and scoring by marginal step support (5.65) perform alike, whereas scoring plans under the full belief (2.79), keeping one rule per factor (3.21), or executing the selected plan without replanning (2.23) each lose more than two achievements. Deleting a contradicted rule matches down-weighting it when every rule is kept (5.77) but not with one rule per factor (1.85), for a reason Appendix H.2 explains.

Table 11: Ablation results on open-ended Crafter-OO, as mean distinct achievements ± standard error over the worlds of Table 2, with guards induced from each source episode separately (Per episode) and from all eight together (Joint). The second group removes components of ProPS, the third starts from the rule posterior without confirmation and changes one further component (Appendix D.3), the fourth gives the other beliefs over the rules with and without the availability posterior, and the last gives the LLM agent that reads the law records. A dash marks a variant not run with jointly induced guards.

Controller	Per episode	Joint
ProPS (ours)	6.90 ± 0.27	7.02 ± 0.30
Components of ProPS		
− Availability posterior (rule posterior)	5.96 ± 0.29	—
− Confirmation	6.48 ± 0.20	—
− Both (rule posterior without confirmation)	5.71 ± 0.26	6.27 ± 0.29
From the rule posterior without confirmation, one further change		
Marginal step support	5.65 ± 0.35	—
Full-belief assignment score, with or without revision	2.79 ± 0.23	—
− Retention (one rule per factor)	3.21 ± 0.21	—
− Replanning (plan held)	2.23 ± 0.12	—
Deletion (retained portfolio)	5.77 ± 0.27	—
Deletion, one rule per factor	1.85 ± 0.06	—
Other beliefs over the rules		
Product of experts + availability posterior	5.88 ± 0.25	6.69 ± 0.31
Product of experts	1.06 ± 0.04	1.06 ± 0.04
Best-supported rule + availability posterior	3.29 ± 0.28	6.23 ± 0.26
Best-supported rule	2.27 ± 0.12	5.17 ± 0.25
LLM agent		
ReAct + law memory	3.98 ± 0.25	4.54 ± 0.27

G.5 Counts and draw of the availability posterior

The availability posterior has two parts, the counts behind its mean and the random draw (Table 12). For the product of experts and the rule posterior without confirmation, the counts alone recover the gain of the full posterior, so the posterior helps them by remembering which actions could not run. For the product, the full posterior lowers the share of selected actions that could not run from 98% to 44% over the 48 worlds. The best-supported rule is the exception. Its most common failure is an attempt to craft that runs and produces nothing, which the counts record as available, so only the draw moves it to other goals.

H Failure Cases

We show the rules that per-episode compilation gets wrong (Appendix H.1) and a mismatch in the observation that contradicts a correct rule (Appendix H.2).

Table 12: Results on open-ended Crafter-OO with the availability posterior's counts, its draw, both (Full), or neither, as mean distinct achievements over the worlds of Table 2. Counts uses the posterior mean $(1+f_o)/(2+n_o)$, Draw samples $\mathrm{Beta}(1,1)$ and ignores the counts, and Full samples $\mathrm{Beta}(1+f_o, 1+n_o-f_o)$.

	Mean distinct achievements				Paired difference			
Controller	None	Counts	Draw	Full	Counts − none	Draw − none	Full − counts	Full − draw
Product of experts	1.06	6.21	3.08	5.88	5.15	2.02	−0.33	2.79
Best-supported rule	2.27	1.29	4.23	3.29	−0.98	1.96	2.00	−0.94
Rule posterior without confirmation	5.71	6.35	5.21	6.48	0.65	−0.50	0.13	1.27

H.1 Wrong rules

Compiling each source episode on its own evidence produces three kinds of wrong rule (Table 13). A rule can miss a requirement. Every episode's rule for crafting a stone sword omits the table, because no episode failed to craft one for lack of a table, so the factor holds a single rule without it. Every controller pays for this rule, with 4.2 to 19.2 failed attempts per world under joint induction, and no belief over a fixed library can recover a rule the library lacks. A rule can also carry a spurious requirement, when induction from few attempts adds whatever material happened to be nearby, or a coincident effect, when a law certifies a change that happened on the same step for another reason.

Table 13: Wrong rules in the portfolio, with the source episodes that produce them. Guard or effect gives what the rule requires besides the action's own items, and Error what Crafter actually requires or does.

Action	Episodes	Guard or effect	Error
Craft stone sword	1, 3, 4, 6	no station	missing table
Mine stone	2, 6	no tool	missing pickaxe
Craft stone pickaxe	0	nearby stone	stone instead of table
Craft wood sword	7	nearby tree	tree instead of table
Mine stone	6	effect adds one health	coincident effect

H.2 A mismatch in the observation

Deletion fails with one rule per factor (Appendix G.4) because of a mismatch in the observation rather than in the dynamics. The rule for collecting wood with the most source support writes that the chopped tree becomes grass. The source record anchors the target tree to the tile the agent chopped, which does become grass, but the target-side observation re-resolves the target to the nearest tree, which after a chop is another tree, so every collection contradicts the rule. With one rule per factor, deletion empties the factor at the first chop and every goal that needs wood is lost, whereas with both rules kept the other rule, which makes no claim about that tile, carries the factor.