Autonomous Assessment of Generalizability of AI Agent Capabilities

Daniel Bramblett[1], **Rushang Karia**[1], **Adrian Ciotinga**[1], **Pulkit Verma**[2], **YooJung Choi**[1], **Siddharth Srivastava**[1]

[1]Arizona State University, AZ, USA
[2]Indian Institute of Technology Madras, India
{drbrambl,rkaria,acioting,yj.choi,siddharths}@asu.edu, pulkitv@cse.iitm.ac.in

Abstract

Safe deployment of black-box AI (BBAI) systems such as foundation model agents requires methods for evaluating their capabilities in novel settings. We define an agent's capability as its ability to achieve a short term objective and formalize the problem of learning models that predict whether, with what effects, and under what conditions, an agent can perform a capability. We introduce Monte Carlo Query Synthesis (MCQS), an active query-synthesis method for learning symbolic stochastic capability models of BBAIs. MCQS models capabilities as conditional probability distributions over outcomes and formulates capability evaluation as an active learning problem over policies. We use Monte Carlo tree search to synthesize queries that maximally distinguish between extremal capability hypotheses: the lattice meet and join corresponding to the most pessimistic and optimistic models consistent with observed behavior. Executing these queries yields trajectories that prune inconsistent hypotheses. We prove soundness, completeness, and convergence properties under standard realizability and sampling assumptions. Experiments with multiple BBAI systems show that MCQS learns accurate capability models more efficiently than baseline query strategies, enabling systematic characterization of agent capability boundaries with fewer interactions.

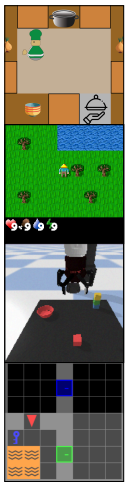
1 Introduction

We consider the problem of efficiently learning interpretable capability models of black-box AI systems (BBAIs) capable of planning and learning. Modern BBAIs such as Large Language Models (LLMs), Vision-Language Models (VLMs), etc. accept high-level instructions and perform long-horizon sequential decision-making (Ha et al. 2023; Liu et al. 2023; Black et al. 2025). However, it is difficult to predict what such BBAIs can and cannot do. For instance, our experiments show that spurious correlations in training data can induce RL agents to exhibit "superstitious" behaviors, such as opening an unnecessary door when the goal is to retrieve a key. Such configuration-dependent effects complicate safe deployment and limit the reliable use of AI systems.

Prior work on the topic is largely restricted to agents with pre-determined capabilities that can be modeled in relatively simple modeling languages (e.g. QACE (Verma, Karia, and Srivastava 2023)). Another related direction of work, on action-model learning (Sec. 5), learns models describing how an agent's primitive actions affect its environment. In contrast, we focus on the relatively under-studied problem of learning *capability models* that characterize the types of planning problems an agent can solve, the conditions under which it can solve them, and their possible outcomes. This is a challenging open problem because we cannot assume prior knowledge of a BBAI's internal representations and planning algorithms. Algorithms for action model learning do not address this setting and existing approaches for capability learning cannot be used in stochastic environments involving agents with non-stationary policies or evolving capabilities. However, solving this problem is essential for determining the limits of generalizability of planning behavior in popular BBAIs. E.g., Fig. 1 shows some of the discovered agent capabilities and their surprising limitations found using the approach presented in this paper. Each capability may involve the execution of multiple actions.

We formalize capability discovery and learning as an active learning problem over a hypothesis class $\mathcal{H}$ of capability models. Each model $h \in \mathcal{H}$ induces a distribution over state-trajectories resulting from BBAI's execution of its capabilities. Unlike standard active learning, the query space here is the space of policies that map states to capabilities, which grows exponentially with the state space.

Our main contribution is a class of Monte Carlo Query Search (MCQS) algorithms for constructing queries for this policy-level active learning problem. The key challenge is the combinatorial structure of both the policy space and the hypothesis class. We address this via two observations. First, the set of hypotheses consistent with observed BBAI executions (state trajectories) constitutes a subset lattice. We show that it suffices to track its extremal elements, i.e., the join and meet of all possible consistent hypotheses ($h_\vee$ and $h_\wedge$ respectively), which can be computed efficiently without enumerating individual hypotheses. Second, inspired by Bell experiments (Bell 1964), we define discriminating policies as those that induce disparate trajectory distributions under the hypotheses $h_\vee$ and $h_\wedge$. Concretely, we measure a policy's ability to discriminate among the two competing hypotheses in the form of expected divergence between trajectory distributions induced by each hypothesis, and treat this divergence measure as a maximization objective for pol-



(a)

Agent	Capability	Conditions & Outcomes
GPT-5.4 on Minigrid	Pick up blue key (in SW)	From NW or SW, achieves intent with $\sim$0.7 probability; side effect: opens unnecessary door in NW with $\sim$0.5 probability.
	Open blue door	From SW, always achieves intent when not holding key; but probability of success drops to 0.8 when holding key.
	Go to SW	From NW, achieves the intent with $\sim$0.2 probability when unnecessary door is closed, but only $\sim$0.1 when it is open; in the latter case, the agent ends up in SE.
CSteve1 on Crafter	Move away from tree	Achieves intent by cutting down tree.
	Achieve drink water	In some scenarios moves away from water instead (probability $\sim$0.7).
SayCan	Stack green on yellow	Achieves intent ($\sim$0.6); fails by picking up entire tower ($\sim$0.4); numerous error outcomes (unstack/knock-down).
LAO* on Blocksworld	Place block A	Always prefers placing on C if clear; otherwise on B.

(b)

Figure 1: (a) Evaluated BBAIs and environments from top-down: Overcooked, Crafter, Saycan, and Minigrid; (b) some of the salient capabilities discovered by MCQS. See Sec. 4.2 for details.

icy computation. Together, these ideas lead to a principled approach for computing policies that are highly informative.

We make minimal assumptions, only requiring access to a *representation function* that maps environment states into a user-interpretable symbolic space. While a substantial body of work studies the problem of learning representation functions (Shah, Nagpal, and Srivastava 2025; Konidaris, Kaelbling, and Lozano-Perez 2018; Ahmetoglu et al. 2022; Peng et al. 2024; James, Rosman, and Konidaris 2020), we focus on the complementary problem of learning capability models given a fixed representation.

The main contributions of this work are: (1) a capability-modeling framework for BBAIs, with a formal learning objective (Sec. 2); (2) MCQS, an approach for capability discovery and learning via informative query synthesis (Sec. 3); (3) theoretical results showing that MCQS converges in the limit (Sec. 4); and (4) empirical results demonstrating its scope and effectiveness across diverse agents and environments (Sec. 4).

2 Formal Framework

We evaluate BBAIs operating in a stochastic, fully observable environment $\mathcal{E}$ characterized by environment states X and low-level actions A. We assume access to a simulator $\mathcal{S}_\mathcal{E}$ for $\mathcal{E}$ that supports standard functionality: resetting to an initial state, reverting to any previously encountered state $x \in X$, stepping the simulator with an action $a \in A$ to obtain the next state and outcome, and querying the set of available actions. We otherwise assume neither explicit knowledge of $\mathcal{E}$ nor additional functionality from $\mathcal{S}_\mathcal{E}$.

Environment states X are often uninterpretable to users, motivating a high-level symbolic representation for expressing both states and capability models. We assume an interpretable symbolic state space S defined over a vocabulary V consisting of a set of predicate symbols P and a set of objects O, where ground atoms are formed by applying predicates in P to objects in O. We use a representation function $\alpha : X \to S$ where $\alpha(x)$ is the set of ground atoms that hold in x. As noted earlier (Sec. **??**), several approaches are being developed for learning such representations, yet the problem of using them for capability model learning remains understudied and forms the focus of this paper.

Capability Name: c_2; **Intent:** clean(l_1)

Conditional Effect r_n**:**
Condition: charged(robot) $\vee$ at(charger, robot)
Effects: 0.75 : clean(l_1) $\wedge$ at(charger, robot)
0.25 : $\neg$charged(robot)

Conditional Effect r_{n+1}**:**
Condition: $\neg$charged(robot) $\wedge$ $\neg$at(charger, robot)
Effects: 0.10 : at(charger, robot) $\wedge$ charged(robot)
0.90 : request_help(user)

Figure 2: Simplified example of a capability model.

2.1 Capability Learning Problem

Intuitively, each capability corresponds to a latent intent-conditioned option (Sutton, Precup, and Singh 1999). We consider intents $\mathcal{I}$, where each intent is a conjunction of ground atoms over V. We use "goal" for a user-assigned objective and "intent" for the agent's current short-term objective, possibly pursued as a step toward a goal. Each intent $i \in \mathcal{I}$ is associated with a latent option consisting of an initiation set, policy, and termination function. Let $\Delta(A)$ denote the set of probability distributions over A. Formally:

Definition 1. *Let $\mathcal{A}$ be a BBAI and Ω be a set of options. A* capability function *of $\mathcal{A}$ is a mapping $\kappa : \mathcal{I} \to \Omega$ such that $\kappa(i) = \langle S_0^i, \mu_i, \beta_i \rangle$, where $S_0^i \subseteq S$ is the initiation set, $\mu_i : S \to \Delta(A)$ is a policy for achieving intent i, and $\beta_i : S \to \{0, 1\}$ is the termination function.*

We write the capability for intent i as $c = \kappa(i)$. Let C be the set of capabilities under consideration. Our objective is to learn a model for each $c \in C$ approximating its initiation set S_0^i and the effects induced by executing μ_i from states in S_0^i until termination under β_i.

To be able to express models of real BBAIs, we model capabilities using conditional probabilistic effects, where conditions may include conjunctions and disjunctions, and outcomes are stochastic. Fig. 2 shows a simplified example for a robot vacuum cleaner's capability for achieving the intent of cleaning a location.

We use grounded rather than lifted models for capabilities because agent capabilities exhibit non-liftable asymmetries (Sec. 4.2). Formally,

Definition 2. *A* capability model h *is a set of conditional-effect rules. Each rule* $r \in h$ *is a tuple* $\langle \mathrm{cond}(r), \mathrm{effects}(r) \rangle$*, where* $\mathrm{cond}(r)$ *is a Boolean formula over* V *and* $\mathrm{effects}(r) = \{(p_j, \eta_j)\}_j$ *is a probability distribution over effects. Each effect* η_j *is a conjunction of literals over* V*,* $p_j \in (0, 1]$*, and* $\sum_j p_j = 1$*.*

An effect $\eta = \langle \eta^+, \eta^- \rangle$ consists of sets of positive and negative literals, with $\mathrm{apply}(s, \eta) = (s \setminus \eta^-) \cup \eta^+$. The effect observed in transition $\langle s, c, s' \rangle$ is $\eta(\langle s, c, s' \rangle) = \langle s' \setminus s, s \setminus s' \rangle$. A transition $\langle s, c, s' \rangle$ is consistent with capability model h, denoted as $h \models \langle s, c, s' \rangle$, if there exists $r \in h$ such that $s \models \mathrm{cond}(r)$ and some $(p_j, \eta_j) \in \mathrm{effects}(r)$ satisfies $s' = \mathrm{apply}(s, \eta_j)$. Otherwise, $h \not\models \langle s, c, s' \rangle$. A capability model h_c for c defines a conditional probability distribution over c's outcomes: $\Pr_{h_c}(s' \mid s, c) = (1/|\mathcal{R}_s|) \sum_{r \in \mathcal{R}_s} \sum_{(p_j, \eta_j) \in \mathrm{effects}(r)} p_j \mathbf{1}[s' = \mathrm{apply}(s, \eta_j)]$ where $\mathcal{R}_s = \{r \in h_c : s \models \mathrm{cond}(r)\}$. If $|\mathcal{R}_s| = 0$, then $\Pr_h(s' \mid s, c) = 0$. We denote $h_C^\dagger$ as the set of capability models, one for each $c \in C$, and omit the subscript when clear from context.

A desired property of a learned capability model is that it is sound and complete with respect to a dataset of observed transitions $\mathcal{D}$. Formally,

Definition 3. *Let* $\alpha : X \to S$ *be a representation function. A* capability-transition dataset $\mathcal{D}_{\mathcal{A},\mathcal{E}}$ *is a multiset of triples* $\langle s, c, s' \rangle$ *where executing capability* c *using BBAI* $\mathcal{A}$ *from state* $x \in X$ *results in* $x' \in X$*, with* $s = \alpha(x)$ *and* $s' = \alpha(x')$*.*

Definition 4. *A model* h_c *is* sound w.r.t. $\mathcal{D}$ *if* $\forall s, s' \in S$*,* $h_c \models \langle s, c, s' \rangle \Rightarrow \langle s, c, s' \rangle \in \mathcal{D}$*, and* complete w.r.t. $\mathcal{D}$ *if* $\forall s, s' \in S$*,* $\langle s, c, s' \rangle \in \mathcal{D} \Rightarrow h_c \models \langle s, c, s' \rangle$*.*

We omit "w.r.t. $\mathcal{D}$" when clear from context. Let $\mathcal{D}^\star$ denote all transitions in $S \times C \times S$ induced by the unknown true model $h^{\dagger\star}$. Requiring soundness w.r.t. $\mathcal{D} \subset \mathcal{D}^\star$ is overly restrictive, since models that generalize beyond observed transitions may violate it. Instead, we seek models that are complete w.r.t. $\mathcal{D}$ and sound w.r.t. $\mathcal{D}^\star$. We use the variational distance (VD) to evaluate a hypothesis model $h^\dagger$ w.r.t. $h^{\dagger\star}$ (Pasula, Zettlemoyer, and Kaelbling 2004): $VD(h^\dagger, h^{\dagger\star}) = (\sum_{(s,c,s') \in \mathcal{D}^\star} |\Pr_{h^\dagger}(s' \mid s, c) - \Pr_{h^{\dagger\star}}(s' \mid s, c)|)/|\mathcal{D}^\star|$.

Given a BBAI $\mathcal{A}$, the capability learning problem is to use a representation function and a simulator to discover $\mathcal{A}$'s capabilities and learn associated models that minimize VD from $\mathcal{A}$'s (unknown) true capability model.

3 Monte Carlo Query Synthesis

In this section, we introduce Monte Carlo Query Synthesis (MCQS), a query-synthesis algorithm for discovering a BBAI's capabilities and learning capability models. The complete algorithm, including the algorithm block, is provided in Appendix E.1. MCQS reduces model uncertainty through falsification: it maintains a set of hypothesis models consistent with $\mathcal{D}$, synthesizes policies whose predicted outcomes disagree across those hypotheses, passes the policy as a *query* to the BBAI, which then executes the policy and in the process creates a new trajectory that is added to $\mathcal{D}$. Our approach works for capability definitions provided as a list of intents that the agent may be able to achieve. Alternatively, we can start with an empty set of capabilities and discover candidate capabilities before learning models for them.

We discover BBAI capabilities by letting it perform random walks in the simulator $\mathcal{S}_\mathcal{E}$ and collecting the resulting effects. We further expand this set by observing state-capability transitions added to $\mathcal{D}$. For each observed effect η, we extract every grounded atom $p(o_1, \ldots, o_k)$ that is added or removed and introduce the corresponding addition or deletion as an intent defining a new capability. If any grounding of a predicate p appears in η, we assume that other type-consistent groundings of p are likely achievable as well. Accordingly, MCQS constructs capabilities for all intents $p(o'_1, \ldots, o'_k)$ where $(o'_1, \ldots, o'_k)$ has the same type signature as the observed atom $p(o_1, \ldots, o_k)$.

3.1 Hypothesis Space Lattice for Query Synthesis

The central challenge in actively learning capability models is maximizing the information gained from each query. We show that the hypothesis space forms a lattice and, when restricted to hypotheses consistent with the state-capability transition dataset $\mathcal{D}$, admits two extremal hypotheses that bound all consistent hypotheses.

For state space S and capability set C, let $T = S \times C \times S$ denote the set of possible transitions. Let compound hypothesis $h^\dagger$ denote a compiled set of hypotheses, one per capability in C, and let $\mathcal{H}$ denote the set of all compound hypotheses. $h^\dagger$ represents the subset of T consistent with it: $T_{h^\dagger} = \{t \in T : h^\dagger \models t\}$. Two hypotheses $h_1^\dagger$ and $h_2^\dagger$ are equivalent, $h_1^\dagger \equiv h_2^\dagger$, iff $T_{h_1^\dagger} = T_{h_2^\dagger}$. This extension of $\models$ to compound hypotheses is well-defined because $h^\dagger$ contains exactly one hypothesis per capability. Consequently, the hypothesis space is partially ordered by inclusion: $h_1^\dagger \preceq h_2^\dagger \iff T_{h_1^\dagger} \subseteq T_{h_2^\dagger}$. This induces a subset lattice over $\mathcal{H}$.

Instead of considering the intractable and combinatorial hypothesis space exhaustively, we restrict our attention to hypotheses that are complete with respect to $\mathcal{D}$ (Def. 4). This restricted space has a unique minimal element: the lattice-meet of all complete hypotheses, which classifies only the transitions in $\mathcal{D}$ as consistent. This model is related to prior

work on "safe models" (Stern and Juba 2017); we call it the *pessimistic model* $h^\dagger_\wedge$ to emphasize that it permits only directly observed transitions.

As noted earlier, soundness w.r.t. $\mathcal{D}$ is too restrictive for filtering hypotheses. We therefore relax it to soundness modulo unseen capability executions: a model $h^\dagger$ is *possibly sound* w.r.t. $\mathcal{D}$ iff, whenever $h^\dagger \models \langle s, c, s'\rangle$, either $\langle s, c, s'\rangle \in \mathcal{D}$, or the following conditions hold: (a) $\forall s'', \langle s, c, s''\rangle \notin \mathcal{D}$, and (b) $\exists s_1, s_2, \langle s_1, c, s_2\rangle \in \mathcal{D}$ s.t. $\eta(s, s') = \eta(s_1, s_2)$. This notion of soundness allows for the possibility that, once an execution of c in s is observed, $h^\dagger$'s prediction will prove sound. We define the *optimistic model* $h^\dagger_\vee$ as the join of all possibly sound models w.r.t. $\mathcal{D}$.

Two properties of $\mathcal{D}$ ensure that pessimistic and optimistic models bound the true model. $\mathcal{D}$ is *transition complete* iff for all s, c whenever there exists s' such that $\langle s, c, s'\rangle \in \mathcal{D}$, then for all s'' s.t. $h^{\dagger\star} \models \langle s, c, s''\rangle$, $\langle s, c, s''\rangle \in \mathcal{D}$. We say that $\mathcal{D}$ is *effect complete* iff for all s, c, whenever $h^{\dagger\star} \models \langle s, c, s'\rangle$, then there exists $\langle s_1, c, s_2\rangle \in \mathcal{D}$ such that $\eta(s, s') = \eta(s_1, s_2)$. These conditions ensure the bounds (Appendix A includes proofs for all theoretical results):

Lemma 1. *Let C be a set of agent capabilities with true model $h^{\dagger\star}$ expressible over vocabulary V, and let $h^\dagger_\wedge$ and $h^\dagger_\vee$ be the pessimistic and optimistic models constructed from transition data $\mathcal{D}$. If $\mathcal{D}$ is transition and effect complete, then $h^\dagger_\wedge \preceq h^{\dagger\star} \preceq h^\dagger_\vee$.*

Furthermore, $h^\dagger_\wedge$ and $h^\dagger_\vee$ can be computed efficiently from $\mathcal{D}$ as follows. For each capability c, let $\mathcal{D}_c = \{\langle s, c, s'\rangle : \exists s, s' \langle s, c, s'\rangle \in \mathcal{D}\}$ denote the observed transitions involving c, and let $\eta_c(s, \mathcal{D}) = \{\eta(\langle s, c, s'\rangle) : \exists s' \langle s, c, s'\rangle \in \mathcal{D}_c\}$ denote the effects observed from state s under c. We partition states by equality of these effect sets: $s_1 \sim_c s_2$ iff $\eta_c(s_1, \mathcal{D}) = \eta_c(s_2, \mathcal{D})$, yielding partitions Φ_c. Each block $S_\varphi \in \Phi_c$ has a common effect set E_φ and induces one conditional effect rule. Let $\ell(s) = \bigwedge_{p\in s} p \wedge \bigwedge_{p\notin s} \neg p$ denote representation of a state as a formula.

We then derive the optimistic and pessimistic models using BDDs by extending effect partitioning (Mordoch et al. 2024) to stochastic effects. A set S_φ where a certain effect occurs can be captured with a pessimistic condition $\mathrm{pcond}(S_\varphi) = \bigvee_{s\in S_\varphi} \ell(s)$ or an optimistic condition $\mathrm{ocond}(S_\varphi) = \neg[\bigvee_{s\in \Phi_c \setminus S_\varphi} \ell(s)]$. The corresponding rule is $r_\varphi = \langle \mathrm{cond}(S_\varphi), (\Pr(\eta), \eta) : \eta \in E_\varphi\rangle$, where cond is either pcond or ocond represented as a BDD and $\Pr(\eta)$ is estimated from $\mathcal{D}_c$ by MLE. Collecting all rules for all capabilities using the pessimistic conditions yields $h^\dagger_\wedge$, while using optimistic conditions yields $h^\dagger_\vee$. This gives two hypotheses bounding the space of consistent hypotheses, enabling MCQS to search for queries maximizing their disagreement.

3.2 MCQS Using Lattice-Based Learning

We now present the overall process for query synthesis based on the lattice theory developed above. For hypotheses $h^\dagger_1$ and $h^\dagger_2$ over the capability set C, a capability policy $\pi : S \to C$ is *informative* from an initial state $x_0 \in X$ if executing π from x_0 results in disparate outcome distributions under $h^\dagger_1$ and $h^\dagger_2$. Once such a policy is identified, we send it as a query to the agent. Since the environment may be stochastic, a query requires multiple independent executions of π from the same initial state, thereby increasing the probability of observing an informative outcome. Formally:

Definition 5. *A* query *is a tuple $\langle x_0, \pi, n\rangle$, where $x_0 \in X$ is an initial environment state, $\pi : S \to C$ is a policy mapping represented states to capabilities, and $n \in \mathbb{Z}^+$ is the number of executions of π.*

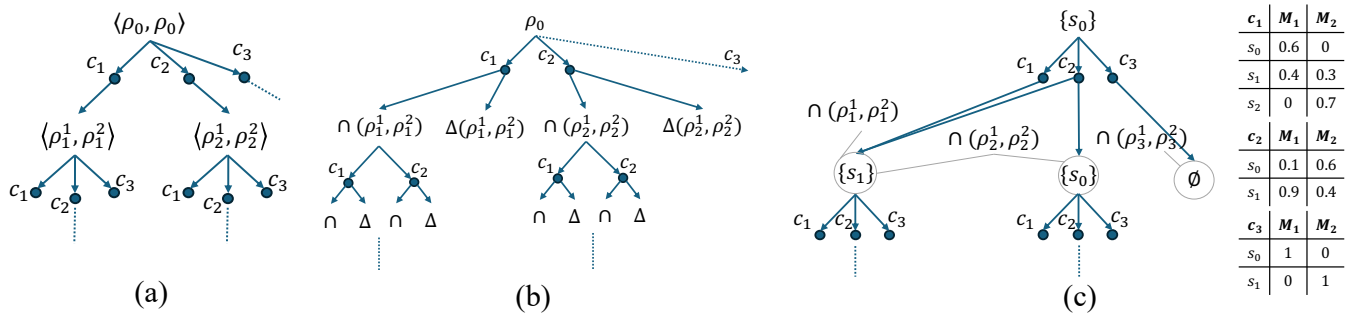
The query-synthesis objective is to identify a policy whose execution induces different predicted state distributions under two capability-set hypotheses. For a capability c_i, let $\tau^h_i(\rho)$ denote the state distribution predicted by model h after executing c_i from an initial distribution ρ: $\tau^h_i(\rho)(s') = \sum_{s\in S} \Pr_h(s' \mid s, c_i)\rho(s)$. We write $\tau^h_{1,\ldots,k}(\rho)$ for the distribution induced by executing the capability sequence $c_1, \ldots, c_k$ from ρ. Let $\hat\delta$ be a distance function over distributions and δ a threshold. An informative policy π is one under which $\hat\delta(\tau^{h^\dagger_1}_\pi(\rho_0), \tau^{h^\dagger_2}_\pi(\rho_0)) > \delta$.

We formulate this objective as a Markov decision process over pairs of predicted state distributions. Let ρ_1 and ρ_2 denote the state distributions maintained under $h^\dagger_1$ and $h^\dagger_2$, respectively. States for the query synthesis MDP are meta-states of the form $\langle \rho_1, \rho_2\rangle$; its actions select a capability $c \in C$; applying c on a meta-state updates each distribution according to its corresponding model. Formally:

Definition 6. *The* distinguishing MDP $\mathcal{P}^\dagger(s_0, h^\dagger_1, h^\dagger_2)$, *where $s_0 \in S$ is the initial state and $h^\dagger_1$ and $h^\dagger_2$ are capability models, is a tuple $\langle \mathcal{S}^\dagger, A^\dagger, \mathcal{T}^\dagger, R^\dagger, s^\dagger_0\rangle$, where $\mathcal{S}^\dagger = \{\langle \rho_1, \rho_2\rangle \mid \rho_1, \rho_2$ are probability distributions over $S\}$; $s^\dagger_0 = \langle \rho_{1,0}, \rho_{2,0}\rangle$ with $\rho_{1,0}(s_0) = 1$ and $\rho_{2,0}(s_0) = 1$; $A^\dagger = C$ is the set of capabilities; $\mathcal{T}^\dagger(\langle \rho_1, \rho_2\rangle, c_i) = \langle \tau^{h^\dagger_1}_i(\rho_1), \tau^{h^\dagger_2}_i(\rho_2)\rangle$; and $R^\dagger(\langle \rho_1, \rho_2\rangle) = \hat\delta(\rho_1, \rho_2)$.*

Solving a distinguishing MDP amounts to finding a policy π that maximizes predicted disagreement: $\arg\max_\pi \hat\delta(\tau^{h^\dagger_1}_\pi(\rho_0), \tau^{h^\dagger_2}_\pi(\rho_0))$. To find π, we use MCTS with UCT (Kocsis and Szepesvári 2006; Świechowski et al. 2023). Let $Q(s^\dagger, c)$ denote the value for applying capability c in state $s^\dagger$, and let $N(s^\dagger)$ and $N(s^\dagger, c)$ denote node and edge visit counts. With exploration constant λ_{uct}, we define $\mathrm{UCT}(s^\dagger, c) = Q(s^\dagger, c) + \lambda_{\mathrm{uct}}\sqrt{\ln N(s^\dagger)/N(s^\dagger, c)}$. During selection, expansion, and policy extraction, MCTS selects $\arg\max_{c\in C} \mathrm{UCT}(s^\dagger, c)$. Retaining the exploration bonus in the query encourages policies targeting rarely explored branches. We develop two MCQS variants that differ in state-distribution representations and rewards.

MCQS-Exact (MCQS-E, Fig. 3(a)) solves the distinguishing MDP directly by maintaining explicit predicted distributions over represented states, encoding each state as a bit-vector with one bit per literal and annotating it with its probability mass. Bit masks test whether a state satisfies each condition in the conditional-effects rules and apply the corresponding effects, yielding an exact but compact representation of the distribution pairs propagated by



c_1	M_1	M_2
s_0	0.6	0
s_1	0.4	0.3
s_2	0	0.7

c_2	M_1	M_2
s_0	0.1	0.6
s_1	0.9	0.4

c_3	M_1	M_2
s_0	1	0
s_1	0	1

Figure 3: Overview of MCQS. ρ_j^i denotes $\tau_j^{M_i}(\rho_0)$; $\cap()$ and $\Delta()$ represent the intersection and the symmetric difference of sets of support (SoS) of their input distributions, respectively. (a) MCTS formulation of the Distinguishing MDP (Def.6); (b) an SoS based representation that replaces each pair of outcome distributions into the intersection and symmetric difference of their supports, and does not generate children for the symmetric difference as they are already distinguished. (c) a sample-based approximation of the SoS representation, where each node represents a sample from two distributions' support sets. Tables on the right show outcome probabilities under the two models.

MCTS (Appendix C). MCQS-E uses total variation distance δ_{TV} for $\hat{\delta}$ since it is symmetric, bounded, and rewards policies assigning probability mass to states likely under one model but unlikely under the other. For node n representing distinguishing-MDP state $s^\dagger$, backpropagation uses the best expected disagreement along the path: $Q(n,c) = \max(\delta_{\text{TV}}(s^\dagger), V(\mathcal{T}^\dagger(s^\dagger,c)))$ and $V(n) = \max_{c\in C} Q(n,c)$.

MCQS with set-of-support (SoS) factorization uses the observation that a state in the symmetric difference of the two models' support sets, denoted by $\Delta(\cdot)$ in Fig. 3(b), is deterministically falsifying: observing such a state in the execution rules out the model whose support excludes it. Such states therefore need not be propagated through MCTS; only states in the *intersection* of the two supports, denoted by $\cap(\cdot)$, carry residual ambiguity. This observation motivates MCQS-S, which propagates only intersection states and approximates them via sampling.

MCQS-S is sample-based MCQS (Fig. 3(c)). It builds on the SoS formulation by replacing nodes that represent intersections of support sets with state nodes sampled from those intersections. At each node with state $s \in S$, MCQS-S selects a valid capability from $C_s = \{c \in C \mid \exists r \in \text{cond\_effects}(c),\ s \models \text{cond}(r)\}$. It then simulates the result of executing capability c by sampling a successor state from the intersection of the two predicted support sets: $s' \sim 0.5\Pr_{h_1}(s' \mid s,c) + 0.5\Pr_{h_2}(s' \mid s,c)$. We use $\Delta(\rho_1,\rho_2)$ to refer to the symmetric difference of the support sets of ρ_1 and ρ_2. Following intuition behind SoS factorization, we define an approximation of TV distance that focuses on states in the symmetric difference: $\hat{\delta}(\rho_1,\rho_2) = \delta_{SD}(\rho_1,\rho_2) = E[I_{\Delta(\rho_1,\rho_2)}] = \sum_{s\in\Delta(\rho_1,\rho_2)} 0.5\rho_1(s) + 0.5\rho_2(s)$. This is used in sample based Q estimates as follows.

During tree traversal, Q is computed as $Q(s,c) = R(s) + \sum_{s'} \Pr(s' \mid s,c)V(s')$, where $V(s) = \max_{c\in C_s} Q(s,c)$. Let c_i be the capability leading to node n with state s, and let $\langle\rho_1,\rho_2\rangle$ be the distribution pair at n's parent. Then $R(s) = 1$ if $s \in \Delta(\tau_i^{h_{C,1}}(\rho_1), \tau_i^{h_{C,2}}(\rho_2))$, and $R(s) = 0$ otherwise. When a state-capability sequence s,c,s',c' is observed during one traversal (including the rollout from a leaf), Q is updated using just the observed state s' rather than the expectation over all states.

3.3 Robust Query Synthesis and Execution

The distinguishing-MDP formulation specifies an ideal query-synthesis objective. We now describe the optimizations MCQS uses to execute queries robustly, revisit informative state-capability pairs, and update the observation dataset. Both MCQS-E and MCQS-S allow query execution to adapt to observed outcomes (Appendix E.2). Additionally, to encourage exploration, we sample the initial state for query synthesis and execution based on frequency (Appendix E.4).

To encourage state-capability revisitation, MCQS uses a ξ-exploration bonus that assigns probability mass to potentially missing effects. Let $\mathcal{D}_{SC}(s,c)$ denote the observations collected for state-capability pair (s,c). We estimate the probability of a missing effect as $\Pr_m(s,c) = 1/(1+|\mathcal{D}_{SC}(s,c)|)$. When evaluating a query, MCQS assigns probability $1-\xi\Pr_m(s,c)$ to the observed effect distribution and probability $\xi\Pr_m(s,c)$ to unseen distinguishing effects. Thus, larger values of ξ encourage revisiting previously tried state-capability pairs, while smaller values prioritize discovering new state-capability pairs.

The dataset $\mathcal{D}$ is updated as follows. We define the *temporal length* of an execution trajectory as the number of symbolic state changes and limit agent execution per capability using a bound Θ on this temporal length. When executing a capability c results in a trajectory $\overline{x} = \langle x_0, x_1, \ldots, x_k\rangle$ that abstracts into a represented state trajectory $s_0, s_1, \ldots, s_j$ ($j < \Theta$) where all consecutive identical states are removed, $\langle s_0, c, s_j\rangle$ is added to $\mathcal{D}$.

4 Results

4.1 Theoretical results

We prove that under realizability and sufficient exploration, MCQS converges to the true capability model in the limit of

infinite samples; proofs and results appear in Appendix A. If MCTS returns optimal policies, we can prove the following.

Theorem 1. *Let C be a set of agent capabilities whose true model $h^{\dagger\star}$ is expressible over vocabulary V, and let $h_i^\dagger$ denote the model learned by MCQS after iteration i. If $\xi > 0$, then $\mathrm{VD}(h_i^\dagger, h^{\dagger\star}) \xrightarrow[i\to\infty]{\text{a.s.}} 0$ over all reachable transitions induced by capabilities in C.*

We additionally prove that there exists a finite set of transitions whose observation is sufficient for the pessimistic and optimistic models to become equivalent.

Theorem 2. *Let $\mathcal{D}$ be the transitions observed by MCQS. If the true agent model $h^{\dagger\star}$ is expressible over vocabulary V, then there exists a finite set of transitions T_δ such that the pessimistic and optimistic models computed from $\mathcal{D} \uplus T_\delta$, denoted by $h_\wedge^\dagger$ and $h_\vee^\dagger$, respectively, satisfy $h_\wedge^\dagger \equiv h_\vee^\dagger$.*

We prove that $h_\wedge^\dagger$ is sound and complete with respect to its dataset, while $h_\vee^\dagger$ is complete (Theorem 3). For a transition and effect complete dataset, $h_\wedge^\dagger \equiv h_\vee^\dagger$ guarantees equivalence to the true model (Theorem 4) and structural equivalence of any complete and possibly sound model to the true model (Theorem 5). Proofs in Appendix A.

4.2 Empirical Evaluation

Our empirical evaluation addresses four questions: *(E1) Does MCQS learn accurate capability models more efficiently than other approaches? (E2) Can MCQS reveal useful information about the limits and capabilities of SOTA BBAIs? (E3) Do the learned capability models accurately predict reachability?* and *(E4) Is maximizing disagreement alone sufficient for learning accurate capability models?* Hyperparameter configurations and detailed results are presented in Appendices E.5 and F.

We assess §E1 by comparing models learned with MCQS-S, MCQS-E, the closest existing approach, and a random-querying ablation baseline across several agents and environments. For §E2, we inspect MCQS's learned models for notable BBAI capabilities and limitations. For §E3, we evaluate whether models learned by MCQS accurately predict outcomes of capability sequences. For §E4, we ablate the revisitation reward by setting $\xi = 0$.

Agents We evaluate six agents across several environments. The HDDLGym (La, Mon-Williams, and Shah 2025) RL agent operates in the Overcooked domain (Carroll et al. 2019). The ReAct agent uses GPT-5.4-nano (OpenAI 2026) in a MiniGrid environment (Chevalier-Boisvert et al. 2023). Both CSteve (Park, Cho, and Ahn 2025) and Qwen CSteve operate in a Crafter environment (Hafner 2022). Finally, the LAO* agent operates on PDDLGym domains (Silver and Chitnis 2020), including Blocksworld and First Responders. As an application, we also evaluate the LLM-based robotic agent SayCan (Ahn et al. 2022).

Baselines To our knowledge, no prior method performs active capability discovery and learning for BBAIs, so we compare against the closest existing approach, QACE (Verma, Karia, and Srivastava 2023), for §E1, and two MCQS ablations for §E1 and §E4. QACE requires the agent's capabilities and intents as input; we supplied those learned by MCQS. QACE learned an accurate model for LAO* on Blocksworld, but produced no model for the other problems within 50 hours due to the limited expressiveness of its model class. The *Random Query* ablation replaces MCTS query synthesis with a uniformly sampled capability sequence $\pi = (c_1, c_2, \ldots, c_{30})$, where $c_i \sim \mathrm{Uniform}(C)$, leaving all other components unchanged and isolating the contribution of query synthesis. For §E4, we ablate MCQS by setting $\xi = 0$.

Evaluation metrics and methodology We evaluate how accurately a learned model $h^\dagger$, constructed from $\mathcal{D}_{h^\dagger}$, predicts observed transition dynamics using weighted sampled variational distance $\mathrm{VD}s(h^\dagger, h^{\dagger\star})$. To construct an independent evaluation dataset $\mathcal{D}'$, we execute the BBAI $\mathcal{A}$ using the same environment and representation function, collecting 100,000 sequences of 10–30 capabilities (5,000 for MiniGrid). Let $N_e(s, c, s' \mid \mathcal{D})$ denote the number of occurrences of transition $\langle s, c, s'\rangle$ in $\mathcal{D}$ and $N_e(s, c \mid \mathcal{D}) = \sum_{s'} N_e(s, c, s' \mid \mathcal{D})$. Let $T_{\mathcal{D}'}$ denote the unique transitions observed in $\mathcal{D}'$, and $c_T = \sum_{\langle s,c,s'\rangle \in T_{\mathcal{D}}} N_e(s, c, s' \mid \mathcal{D}_{h^\dagger})$ be the number of transitions observed in $\mathcal{D}_{h^\dagger}$. We define $\mathrm{VD}_s(h^\dagger, h^{\dagger\star})$ as $\mathrm{VD}_s(h^\dagger, h^{\dagger\star}) = \frac{1}{c_T} \sum_{\langle s,c,s'\rangle \in T_{\mathcal{D}'}} N_e(s, c, s' \mid \mathcal{D}_{h^\dagger}) \times \left| \frac{N_e(s,c,s' \mid \mathcal{D}')}{N_e(s,c \mid \mathcal{D}')} - \Pr_{h^\dagger}(s' \mid s, c) \right|$.

We evaluate learned pessimistic models because they are safer for unseen states. To focus on achievable intents, we remove conditional-effect rules that fail to achieve their intent and discard capabilities never observed as achievable. We use temporal length ∞ for MiniGrid, SayCan, and Overcooked, and 1 otherwise.

We perform five independent model-learning runs per problem. MCQS stops after a timeout or 25 consecutive queries without a newly discovered transition or significant model change (an effect probability changing by more than 5%). To balance revisitation with hard-to-reach capability discovery, we dynamically set ξ (Sec. 3.3) based on the number of queries since the last discovery, up to 10^{-5}. Timeouts are at most two days.

Results and Analysis We now present our analysis of results addressing the evaluation questions E1–E4 listed above.

§E1 (Learning efficiency) Fig. 4 compares model agreement (1 - weighted VD) over the number of simulator steps. On MiniGrid, MCQS-S reaches 91% model agreement with $18.6\times$ fewer simulator steps than the random-query baseline. On First Responders, MCQS-S reaches over 99% with $18.1\times$ fewer steps. On Overcooked, in 10,000 simulator steps, MCQS-E achieves nearly double the model agreement of the random-query baseline.

Crafter's coarse representation function aggregates grounded states sharing the same symbolic state, increasing VD and the simulator steps required to reduce it. This trend is consistent across both CSteve variants, so we report only one. Despite this challenge, MCQS-E reaches 50% model agreement with $10.5\times$ fewer simulator steps than random

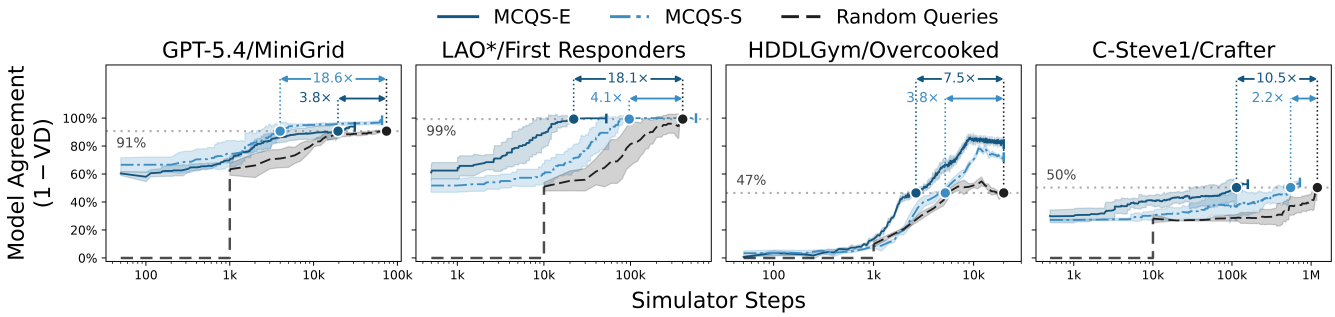


Figure 4: Model agreement (1 - weighted VD) of MCQS-E and MCQS-S on four evaluation problems. Shaded regions indicate one standard deviation over multiple runs. Annotated arrows indicate the simulator step-count ratio between each MCQS approach and the Random Query ablation baseline.

querying, suggesting that MCQS remains effective under imperfect representation functions.

§E2 (Limits and capabilities of SOTA BBAIs) Fig. 1b highlights unintuitive limitations discovered by MCQS-E that are readily interpretable and can inform BBAI development. GPT-5.4 reliably picks up a key when requested, but often unnecessarily traverses to and opens a door; moreover, asking it to pick up the key and then unlock the door is less successful than directly requesting it to unlock the door. SayCan is highly stochastic, with many desired effects occurring with low probability; for example, it stacks a green block on a yellow block only 6% of the time, often misidentifying the yellow block as green. Finally, LAO*'s place-block-A capability reveals a preference likely induced by internal tie-breaking and stationary policies.

These results also illustrate the challenges of lifting accurate capability models for BBAIs. We considered lifted models, but emergent agent capabilities often lack symmetries under object substitution. For example, the LAO* agent prefers placing blocks on C rather than B to clear its hand, while the MiniGrid agent can retrieve a key from NW or SW, but not from other quadrants. Lifting loses such object-specific dependencies, producing exaggerated capabilities that misrepresent the agent's true behavior.

§E3 (Cross evaluation through reachability) We evaluated MCQS-S models on 100 predicted-reachable 5-capability sequences, executing each 100 times on the BBAI. Models achieved $81\% \pm 3.6\%$ accuracy on MiniGrid, 100% on First Responders, and $78\% \pm 16\%$ on Crafter. These results demonstrate that learned capability models can predict downstream performance and support planning.

§E4 (Ablation w.r.t. visitation parameter) With $\xi = 0$, the resulting models produced by MCQS achieved higher than 98% model agreement across three PDDLGym problems: Blocksworld, Tireworld, and First Responders. This demonstrates the effectiveness of rewarding disagreement alone. The revisitation incentive had mixed effects on accuracy and efficiency, with MCQS-S being more sensitive to ξ.

5 Related Work

Research on action model learning focuses on dynamic models of the environment rather than of the agent (Pasula, Zettlemoyer, and Kaelbling 2007; Juba and Stern 2022; Gösgens, Jansen, and Geffner 2025; Verma, Marpally, and Srivastava 2021; Benyamin et al. 2025). This line of work does not address the problem of learning capability models that capture an agent's decision making capabilities, which is the focus of this paper. Tantakoun, Muise, and Zhu (2025) present a survey of LLM-to-PDDL approaches for world model learning, and recent benchmarks (Hu et al. 2025; Zuo et al. 2025) evaluate model generation from LLMs. Recent work also addresses the problem of learning capability models (Verma, Marpally, and Srivastava 2022; Shah, Nagpal, and Srivastava 2025) in the limited setting of deterministic models with add and delete effects. In contrast, our approach learns rich probabilistic capability models conditioned on arbitrary logical formulas in stochastic settings.

Testing and verification approaches such as DeepXplore (Pei et al. 2017), Metamorphic Testing (Chen et al. 2018), safety verification (Tran et al. 2019; Dreossi et al. 2019; Araujo, Mousavi, and Varshosaz 2023), etc. aim to find failures or verify properties, rather than discover what agents can do. Behavior modeling and inverse planning (Baker, Saxe, and Tenenbaum 2009; Shvo and McIlraith 2020) focus on inferring agent goals and strategies from observations, whereas specification and invariant learning (Leucker and Schallhart 2009; Neider et al. 2018; Bao et al. 2024) extract system constraints. While these areas share the goal of understanding black-box systems, our approach differs by discovering and modeling the capabilities an agent can reliably execute under different conditions.

6 Conclusions and Future Work

We presented MCQS, an active query-synthesis approach for discovering and modeling BBAI capabilities in stochastic settings. Empirical and theoretical results show that MCQS efficiently learns useful, accurate capability models within a finite query budget. While effective, MCQS leaves open improvements to exploration efficiency and generalization. In particular, real-world BBAIs often exhibit implicit or context-dependent preferences among multiple valid plans. Distinguishing such preferences from structural constraints remains an open challenge for learning models that both generalize and accurately reflect an agent's behavior.

Acknowledgments

This work was supported in part by the following grants: NSF IIS 2419809, ONR N00014-23-1-2416, and AFOSR FA9550-25-1-0320. We thank Ruthvick Suresh for contributions to the empirical evaluation in an earlier version of this work. We also thank Sai Saketh Reddy Kambalapally for contributing the Crafter agents evaluated in this paper.

References

Ahmetoglu, A.; Seker, M. Y.; Piater, J.; Oztop, E.; and Ugur, E. 2022. Deepsym: Deep symbol generation and rule learning for planning from unsupervised robot interaction. *Journal of Artificial Intelligence Research*, 75: 709–745.

Ahn, M.; Brohan, A.; Brown, N.; and et al. 2022. Do As I Can, Not As I Say: Grounding Language in Robotic Affordances. *arXiv preprint arXiv:2204.01691*.

AI, M. 2024. The Llama 3 Herd of Models. https://www.llama.com/. Includes Llama 3.1 model family. Accessed: 2025-02-05.

Araujo, H.; Mousavi, M. R.; and Varshosaz, M. 2023. Testing, Validation, and Verification of Robotic and Autonomous Systems: A Systematic Review. *ACM Transactions on Software Engineering and Methodology*, 32(2).

Baker, C. L.; Saxe, R.; and Tenenbaum, J. B. 2009. Action Understanding as Inverse Planning. *Cognition*, 113(3): 329–349.

Bao, J.; Trivedi, N.; Pathak, D.; Hsu, J.; and Roy, S. 2024. Data-Driven Invariant Learning for Probabilistic Programs. *Formal Methods in System Design*, 66(2): 278–306.

Bell, J. S. 1964. On the einstein podolsky rosen paradox. *Physics Physique Fizika*, 1(3): 195.

Benyamin, Y.; Mordoch, A.; Shperberg, S. S.; and Stern, R. 2025. Integrating Reinforcement Learning, Action Model Learning, and Numeric Planning for Tackling Complex Tasks. *arXiv preprint arXiv:2502.13006*.

Black, K.; Brown, N.; Darpinian, J.; Dhabalia, K.; Driess, D.; Esmail, A.; Equi, M. R.; Finn, C.; Fusai, N.; Galliker, M. Y.; et al. 2025. $\pi_{0.5}$: A Vision-Language-Action Model with Open-World Generalization. In *Proc. CoRL*.

Carroll, M.; Shah, R.; Ho, M. K.; Griffiths, T.; Seshia, S.; Abbeel, P.; and Dragan, A. 2019. On the Utility of Learning about Humans for Human-AI Coordination. In *Proc. NeurIPS*.

Chen, T. Y.; Kuo, F.-C.; Liu, H.; Poon, P.-L.; Towey, D.; Tse, T. H.; and Zhou, Z. Q. 2018. Metamorphic Testing: A Review of Challenges and Opportunities. *ACM Computing Surveys (CSUR)*, 51(1).

Chevalier-Boisvert, M.; Dai, B.; Towers, M.; de Lazcano, R.; Willems, L.; Lahlou, S.; Pal, S.; Castro, P. S.; and Terry, J. 2023. Minigrid & Miniworld: Modular & Customizable Reinforcement Learning Environments for Goal-Oriented Tasks. *arXiv preprint arXiv:2306.13831*.

Dreossi, T.; Fremont, D. J.; Ghosh, S.; Kim, E.; Ravanbakhsh, H.; Vazquez-Chanlatte, M.; and Seshia, S. A. 2019. VerifAI: A A Toolkit for the Formal Design and Analysis of Artificial Intelligence-Based Systems. In *Proc. CAV*.

Gösgens, J.; Jansen, N.; and Geffner, H. 2025. Learning Lifted STRIPS Models from Action Traces Alone: A Simple, General, and Scalable Solution. In *Proc. ICAPS*.

Ha, T.; Lee, D.; Kwon, Y.; Park, M. S.; Lee, S.; Jang, J.; Choi, B.; Jeon, H.; Kim, J.; Choi, H.; et al. 2023. AI-driven Robotic Chemist for Autonomous Synthesis of Organic Molecules. *Science Advances*, 9(44): eadj0461.

Hafner, D. 2022. Benchmarking the Spectrum of Agent Capabilities. In *International Conference on Learning Representations*.

Hu, M.; Chen, T.; Zou, Y.; Lei, Y.; Chen, Q.; Li, M.; Mu, Y.; Zhang, H.; Shao, W.; and Luo, P. 2025. Text2World: Benchmarking Large Language Models for Symbolic World Model Generation. In *Proc. ACL (Findings)*.

James, S.; Rosman, B.; and Konidaris, G. 2020. Learning Portable Representations for High-level Planning. In *Proc. ICML*.

Juba, B.; and Stern, R. 2022. Learning Probably Approximately Complete and Safe Action Models for Stochastic Worlds. In *Proc. AAAI*.

Khan, Z.; Prasad, A.; Stengel-Eskin, E.; Cho, J.; and Bansal, M. 2025. One life to learn: Inferring symbolic world models for stochastic environments from unguided exploration. *arXiv preprint arXiv:2510.12088*.

Kocsis, L.; and Szepesvári, C. 2006. Bandit based Monte-Carlo Planning. In *Proc. ECML*.

Konidaris, G.; Kaelbling, L. P.; and Lozano-Perez, T. 2018. From skills to symbols: Learning symbolic representations for abstract high-level planning. *Journal of Artificial Intelligence Research*, 61: 215–289.

La, N.; Mon-Williams, R.; and Shah, J. A. 2025. HDDL-Gym: A Tool for Studying Multi-Agent Hierarchical Problems Defined in HDDL with OpenAI Gym. In *Proc. ICAPS*.

Leucker, M.; and Schallhart, C. 2009. A Brief Account of Runtime Verification. *The Journal of Logic and Algebraic Programming*, 78(5): 293–303.

Liu, H.; Li, C.; Wu, Q.; and Lee, Y. J. 2023. Visual Instruction Tuning. In *Proc. NeurIPS*.

Mordoch, A.; Scala, E.; Stern, R.; and Juba, B. 2024. Safe Learning of PDDL Domains with Conditional Effects. In *Proc. ICAPS*.

Neider, D.; Garg, P.; Madhusudan, P.; Saha, S.; and Park, D. 2018. Invariant Synthesis for Incomplete Verification Engines. In *Proc. TACAS*.

OpenAI. 2026. GPT-.1-mini. https://platform.openai.com/docs/models/gpt-5.4-nano. Accessed: 2025-05-01.

Park, J.; Cho, H.; and Ahn, S. 2025. CrafterDojo: A Suite of Foundation Models for Building Open-Ended Embodied Agents in Crafter. *arXiv preprint arXiv:2508.13530*.

Pasula, H.; Zettlemoyer, L. S.; and Kaelbling, L. P. 2004. Learning Probabilistic Relational Planning Rules. In *Proc. ICAPS*.

Pasula, H. M.; Zettlemoyer, L. S.; and Kaelbling, L. P. 2007. Learning Symbolic Models of Stochastic Domains. *Journal of Artificial Intelligence Research*, 29: 309–352.

Pei, K.; Cao, Y.; Yang, J.; and Jana, S. 2017. DeepXplore: Automated Whitebox Testing of Deep Learning Systems. In *Proc. SOSP*.

Peng, A.; Bobu, A.; Li, B. Z.; Sumers, T. R.; Sucholutsky, I.; Kumar, N.; Griffiths, T. L.; and Shah, J. A. 2024. Preference-conditioned Language-guided Abstraction. In *Proc. HRI*.

Shah, N.; Nagpal, J.; and Srivastava, S. 2025. From Real World to Logic and Back: Learning Generalizable Relational Concepts For Long Horizon Robot Planning. In *Proc. CoRL*.

Shvo, M.; and McIlraith, S. A. 2020. Active Goal Recognition. *Proc. AAAI*.

Silver, T.; and Chitnis, R. 2020. PDDLGym: Gym Environments from PDDL Problems. In *ICAPS 2020 PRL Workshop*.

Stern, R.; and Juba, B. 2017. Efficient, Safe, and Probably Approximately Complete Learning of Action Models. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, 4405–4411.

Sutton, R. S.; Precup, D.; and Singh, S. 1999. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2): 181–211.

Świechowski, M.; Godlewski, K.; Sawicki, B.; and Mańdziuk, J. 2023. Monte Carlo Tree Search: A Review of Recent Modifications and Applications. *Artificial Intelligence Review*, 56(3): 2497–2562.

Tantakoun, M.; Muise, C.; and Zhu, X. 2025. LLMs as Planning Formalizers: A Survey for Leveraging Large Language Models to Construct Automated Planning Models. In *Proc. ACL (Findings)*.

Team, Q. 2024. Qwen2.5: A Party of Foundation Models.

Tran, H.-D.; Cai, F.; Diego, M. L.; Musau, P.; Johnson, T. T.; and Koutsoukos, X. 2019. Safety Verification of Cyber-Physical Systems with Reinforcement Learning Control. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(5s).

Verma, P.; Karia, R.; and Srivastava, S. 2023. Autonomous Capability Assessment of Sequential Decision-Making Systems in Stochastic Settings. In *Proc. NeurIPS*.

Verma, P.; Marpally, S. R.; and Srivastava, S. 2021. Asking the Right Questions: Learning Interpretable Action Models Through Query Answering. In *Proc. AAAI*.

Verma, P.; Marpally, S. R.; and Srivastava, S. 2022. Discovering User-Interpretable Capabilities of Black-Box Planning Agents. In *Proc. KR*.

Zuo, M.; Velez, F. P.; Li, X.; Littman, M.; and Bach, S. 2025. Planetarium: A Rigorous Benchmark for Translating Text to Structured Planning Languages. In *Proc. NAACL*.

A Formal Proofs

This appendix provides proofs of Lemma 1 (Sec. 3.1) and theorems presented in Sec.4.1, along with additional theoretical results. As discussed in Sec. 4.1, the proof of Theorem 1 requires the assumption *A1: MCTS returns an optimal policy*. This assumption is used only in the proofs of Lemmas 3 and 5, which in turn are used exclusively to establish Theorem 1. All remaining lemmas and theorems are proved without relying on Assumption A1.

We begin by proving that, with respect to the dataset from which they are constructed, the pessimistic model is sound and complete, while the optimistic model is complete.

Theorem 3. *Let $\mathcal{D}$ be the observed transitions, and let $h^\dagger_\wedge$ and $h^\dagger_\vee$ be the pessimistic and optimistic models computed by MCQS, respectively. Then $h^\dagger_\wedge$ is sound and complete with respect to $\mathcal{D}$, and $h^\dagger_\vee$ is complete with respect to $\mathcal{D}$.*

Proof. Let $\langle s, c, s' \rangle \in \mathcal{D}$. By construction $h^\dagger_\wedge \models \langle s, c, s' \rangle$, so $h^\dagger_\wedge$ is complete w.r.t. $\mathcal{D}$ (Def. 4). Since every transition admitted by $h^\dagger_\wedge$ lies in $\mathcal{D}$, it is possibly sound, and as $h^\dagger_\vee$ is the join of all possibly sound hypotheses, $h^\dagger_\wedge \preceq h^\dagger_\vee$. Hence $h^\dagger_\vee \models \langle s, c, s' \rangle$ and $h^\dagger_\vee$ is complete w.r.t. $\mathcal{D}$ (Def. 4).

Conversely, if $h^\dagger_\wedge \models \langle s, c, s' \rangle$ then $\langle s, c, s' \rangle \in \mathcal{D}$ by construction, so $h^\dagger_\wedge$ is sound w.r.t. $\mathcal{D}$ (Def. 4). □

A.1 Bounding the True Model

Let $T_{h^{\dagger\star}}$ denote the set of transitions consistent with the true model. We first show that $h^{\dagger\star}$ is bounded by the pessimistic and optimistic hypotheses when $h^{\dagger\star}$ is expressible with the vocabulary and $\mathcal{D}$ is transition and effect complete (Sec. 3.1).

Lemma 1. *Let C be a set of agent capabilities with true model $h^{\dagger\star}$ expressible over vocabulary V, and let $h^\dagger_\wedge$ and $h^\dagger_\vee$ be the pessimistic and optimistic models constructed from transition data $\mathcal{D}$. If $\mathcal{D}$ is transition and effect complete, then $h^\dagger_\wedge \preceq h^{\dagger\star} \preceq h^\dagger_\vee$.*

Proof. By construction, $h^\dagger_\wedge$ admits exactly the observed transitions $T_\mathcal{D}$, so $T_{h^\dagger_\wedge} = T_\mathcal{D}$. Since observed transitions are consistent with the true model, $T_\mathcal{D} \subseteq T_{h^{\dagger\star}}$, giving $h^\dagger_\wedge \preceq h^{\dagger\star}$.

For the upper bound, let $\langle s, c, s' \rangle \in T_{h^{\dagger\star}}$. If $\langle s, c, s' \rangle \in T_\mathcal{D}$, then $\langle s, c, s' \rangle \in T_{h^\dagger_\vee}$ by Theorem 3. Otherwise (s, c) is untried: were it tried, the premise of $\mathcal{D}$ being transition complete means that effect $h^{\dagger\star}$ produces at (s, c) was observed, i.e. $\langle s, c, s' \rangle \in T_\mathcal{D}$. Let η be the effect realizing s' at s, so $\mathrm{apply}(s, \eta) = s'$. By the premise of $\mathcal{D}$ being effect complete, η has been witnessed in $\mathcal{D}$, so $h^\dagger_\vee$ contains a rule r with $\eta \in \mathrm{effects}(r)$. By construction, the optimistic condition ocond excludes a state only if that state is observed to be governed by a different rule; since (s, c) is untried, no observation assigns s to a rule for c, so $s \models \mathrm{ocond}(r)$ and $\langle s, c, s' \rangle \in T_{h^\dagger_\vee}$.

Hence $T_{h^{\dagger\star}} \subseteq T_{h^\dagger_\vee}$, so $h^{\dagger\star} \preceq h^\dagger_\vee$ and $h^\dagger_\wedge \preceq h^{\dagger\star} \preceq h^\dagger_\vee$. □

A.2 MCQS Convergence to the True Model

We now prove the main theoretical result from Sec. 4.1: the variational distance between a model $h^\dagger$ produced by MCQS and the true model $h^{\dagger\star}$ almost surely converges to zero on

reachable transitions as the number of iterations of MCQS approaches infinity.

We first characterize the reward when the distributions ρ_0 and ρ_1 represented by a meta-state $s^\dagger$ are equal. Specifically, letting $\rho_0 = \rho_1 = \rho$, if capability c has previously been executed in every state in $\mathrm{supp}(\rho)$, we show that the reward of the resulting meta-state $s_1^\dagger$ is exactly the expected revisitation bonus over its constituent states.

Lemma 2. *Let $c \in C$ be a capability and $s^\dagger$ be a meta-state of the distinguishing MDP $\mathcal{P}(s_0, h_\wedge^\dagger, h_\vee^\dagger)$ representing distributions ρ_0 and ρ_1. If $\rho_0 = \rho_1 = \rho$ and, for every $s \in \mathrm{supp}(\rho)$, the transition dataset $\mathcal{D}$ contains an observation of executing c from s, then the reward of the resulting meta-state of executing c in $s^\dagger$, $s_1^\dagger$, is $R(s_1^\dagger) = \sum_{s \in supp(\rho)} \frac{\rho(s)\xi}{\mathcal{D}_{SC}(s,c)+1}$.*

Proof. By the premises, for each state $s \in \mathrm{supp}(\rho)$, $\mathcal{D}$ contains an observation of performing c in s. By construction, the pessimistic and optimistic models therefore predict the same transitions from performing c in s and thus induce the same successor distribution. Since $\rho_0 = \rho_1 = \rho$, the two distributions ρ_0' and ρ_1' represented by $s_1^\dagger$ are equal, and hence $\hat{\delta}(\rho_0', \rho_1') = 0$.

Thus, the only remaining reward for each state s is its revisit value, $\frac{\xi}{\mathcal{D}_{SC}(s,c)+1}$. Weighting each revisit value by $\rho(s)$, the reward from performing c in $s^\dagger$ is $R(s_1^\dagger) = \sum_{s \in supp(\rho)} \frac{\rho(s)\xi}{\mathcal{D}_{SC}(s,c)+1}$. □

We define consistency in terms of variational distance. Two models, $h_1^\dagger$ and $h_2^\dagger$, are *consistent* w.r.t. state-capability pair (s, c) iff $\forall s', \Pr_{h_1^\dagger}(s'|s,c) = \Pr_{h_2^\dagger}(s'|s,c)$. Therefore, $h_1^\dagger$ and $h_2^\dagger$ are inconsistent w.r.t. (s, c) iff $\exists s', \Pr_{h_1^\dagger}(s'|s,c) \neq \Pr_{h_2^\dagger}(s'|s,c)$. The inconsistency with $h_1^\dagger$ is reduced between models $h_2^\dagger$ and $h_3^\dagger$ w.r.t. (s, c) iff $\sum_{s' \in S} |\Pr_{h_3^\dagger}(s'|s,c) - \Pr_{h_1^\dagger}(s'|s,c)| < \sum_{s' \in S} |\Pr_{h_2^\dagger}(s'|s,c) - \Pr_{h_1^\dagger}(s'|s,c)|$.

We next show that any reachable state-capability pair (s, c) on which either $h_\wedge^\dagger$ or $h_\vee^\dagger$ is inconsistent with $h^{\dagger\star}$ is eventually queried with non-zero probability.

Lemma 3. *Let $h^{\dagger\star}$ be the true model, expressible over vocabulary V, and let $h_\wedge^\dagger$ and $h_\vee^\dagger$ be the pessimistic and optimistic models computed by MCQS, respectively. If $\xi > 0$ and there exists a reachable state-capability pair (s, c) such that either $h_\wedge^\dagger$ or $h_\vee^\dagger$ is inconsistent with $h^{\dagger\star}$ w.r.t. (s, c), then, after finitely many iterations, MCQS produces a query with non-zero probability of executing c in s.*

Proof. Since meta-states exactly represent the induced distributions, any finite capability sequence deterministically induces a sequence of meta-states.

Since (s, c) is reachable, there exists a capability sequence $\overline{c}^\star = (c_1, \ldots, c_n, c)$ such that, after $c_1, \ldots, c_n$, the resulting meta-state $s^\dagger$ represents distributions ρ_0 and ρ_1 with $s \in \mathrm{supp}(\rho_0) \cup \mathrm{supp}(\rho_1)$. Thus, executing $\overline{c}^\star$ has non-zero probability of executing c in s.

Suppose, for contradiction, that after some iteration k, MCQS never again produces such a query. Then $\mathcal{D}_{SC}(s, c)$ remains finite. Let $\rho = (\rho_0 + \rho_1)/2$. Since $\rho(s) > 0$ and the disagreement reward is non-negative, the reward of $\overline{c}^\star$ is bounded below by its revisit reward, $R(\overline{c}^\star) \geq \frac{\xi\rho(s)}{\mathcal{D}_{SC}+1} =: \beta > 0$.

Now consider the queries selected after iteration k. By assumption, all avoid executing c in s. Repeated selection of any such query increases the visitation counts of its encountered state-capability pairs, driving its revisitation reward to zero. Once all the finite combinations of state-capability pairs have been tried, its disagreement reward will also be zero by Lemma 2. Hence, its total reward converges to zero under repeated selection.

Because P and O are finite and queries have finite horizon, there are finitely many candidate queries. Therefore, after finitely many iterations, every query avoiding (s, c) has reward less than β. However, $R(\overline{c}^\star) \geq \beta$, and, by Assumption A1, MCQS selects an optimal query. It must therefore select a query with non-zero probability of executing c in s, contradicting the assumption.

Thus, MCQS produces such a query after finitely many iterations. □

Now we show that any inconsistency at a state-capability pair (s, c) has a non-zero probability of being reduced after finitely many observations of performing c in s.

Lemma 4. *Let $h^{\dagger\star}$ be the true model, expressible over vocabulary V, and let $h^\dagger$ be the pessimistic or optimistic model constructed from $\mathcal{D}$, respectively. If $h^\dagger$ is inconsistent with $h^{\dagger\star}$ w.r.t. state-capability pair (s, c), then there exists a finite k such that k additional executions of c in s have non-zero probability of reducing the inconsistency.*

Proof. Observations from performing c in s are i.i.d. draws from the true effect distribution $p^\star$ at (s, c). There are four possible causes for $\exists s', \Pr_{h^\dagger}(s'|s,c) \neq \Pr_{h^{\dagger\star}}(s'|s,c)$.

Case 1 (additional effect): $\Pr_{h^\dagger}(s' \mid s, c) > 0$ *and* $\Pr_{h^{\dagger\star}}(s' \mid s, c) = 0$. Then $\langle s, c, s' \rangle \notin \mathcal{D}$, since every observed transition is realizable under $h^{\dagger\star}$. By construction, $h_\wedge^\dagger$ classifies only transitions in $\mathcal{D}$ as consistent, so the inconsistency must occur in $h_\vee^\dagger$; thus, there exists a rule r assigning non-zero probability to η with $s \models \mathrm{ocond}(r)$. By construction of the optimistic condition ocond, this is possible only if s has not been partitioned into another rule for c, and hence (s, c) has not been observed in $\mathcal{D}$. Therefore, a single execution of c in s assigns s to a partition, removing η from the effects predicted at (s, c) and hence reducing the inconsistency.

Case 2 (missing effect): $\Pr_{h^\dagger}(s' \mid s, c) = 0$ *and* $p_\eta^\star = \Pr_{h^{\dagger\star}}(s' \mid s, c) > 0$. A single additional execution of c in s observes η with probability $p_\eta^\star > 0$, after which $h^\dagger$ assigns η positive probability, reducing the inconsistency.

Case 3 (incorrect condition). If s is assigned to a partition whose effect set differs from that of (s, c), the resulting discrepancy is a set of effects predicted but not realizable (Case 1) or realizable but not predicted (Case 2) at (s, c). Therefore, the preceding cases apply.

Case 4 (distributional error): $\Pr_{h^\dagger}(s' \mid s, c)$ *and* $\Pr_{h^{\dagger\star}}(s' \mid s, c)$ *are both positive but unequal.* Suppose instead $h^\dagger$'s error at (s, c) is a distributional error over the n effects: $e_{(s,c)} = d_{\mathrm{VD}}(\hat{p}_{k_0}, p^\star)\frac{1}{n}\sum_\eta |\hat{p}_{k_0,\eta} - p^\star_\eta| > 0$, where $\hat{p}_{k_0}$ is the current MLE from the k_0 observations of (s, c) in $\mathcal{D}$. Since $h^\dagger$ is given, $\hat{p}_{k_0}$ and hence $e_{(s,c)}$ are fixed. Let $\hat{p}_{k_0+k}$ denote the empirical MLE after k additional i.i.d. executions of c in s. By the strong law of large numbers applied to each of the finite effects, $\hat{p}_{k_0+k} \xrightarrow[k\to\infty]{\text{a.s.}} p^\star$, and since $d_{\mathrm{VD}}(\cdot, p^\star)$ is continuous, $d_{\mathrm{VD}}(\hat{p}_{k_0+k}, p^\star) \to 0$ almost surely. Almost sure convergence implies convergence in probability, so $q_k := \Pr\big(d_{\mathrm{VD}}(\hat{p}_{k_0+k}, p^\star) < e_{(s,c)}\big) \to 1$ as $k \to \infty$. By the definition of convergence, taking $\delta = \frac{1}{2}$, there exists a finite K such that $q_k > \frac{1}{2}$ for all $k \geq K$. Hence, $k = K$ additional executions suffice for $d_{\mathrm{VD}}(\hat{p}_{k_0+k}, p^\star) < d_{\mathrm{VD}}(\hat{p}_{k_0}, p^\star)$ with strictly positive probability

In all cases, there is a finite k such that k additional executions of c in s have non-zero probability of reducing the inconsistency in $h^\dagger$ w.r.t (s, c) and $h^{\dagger\star}$. □

Combining Lemmas 3 and 4, we show that any inconsistency at a reachable state-capability pair has a non-zero probability of being reduced after finitely many iterations of MCQS.

Lemma 5. *Let $h^{\dagger\star}$ be the true model, expressible over vocabulary V, and let $h^\dagger_\wedge$ and $h^\dagger_\vee$ be the pessimistic and optimistic models computed by MCQS from $\mathcal{D}$, respectively. If $\xi > 0$ and either $h^\dagger_\wedge$ or $h^\dagger_\vee$ is inconsistent with $h^{\dagger\star}$ w.r.t. a reachable state-capability pair (s, c), then there exists a finite number of iterations of MCQS after which this inconsistency has a non-zero probability of being reduced.*

Proof. Suppose either $h^\dagger_\wedge$ or $h^\dagger_\vee$ remains inconsistent with $h^{\dagger\star}$ w.r.t. (s, c). By Lemma 4, some finite k additional observations of c in s has a non-zero probability of reducing this inconsistency.

By Lemma 3, each such observation occurs after a finite number of MCQS iterations with non-zero probability, and this continues to hold if the inconsistency persists. Applying this at most k times, MCQS collects all k observations within a finite number of iterations with non-zero probability.

Since the probability of collecting the k observations is non-zero and the probability that they reduce the inconsistency is non-zero, the inconsistency at (s, c) has a non-zero probability of being reduced after finitely many iterations. □

Having established that any error has a non-zero probability of being reduced after finitely many iterations of MCQS, we now show that, in the limit, the learned model almost surely has variational distance 0 from the true model over reachable transitions.

Theorem 1. *Let C be a set of agent capabilities whose true model $h^{\dagger\star}$ is expressible over vocabulary V, and let $h^\dagger_i$ denote the model learned by MCQS after iteration i. If $\xi > 0$, then $\mathrm{VD}(h^\dagger_i, h^{\dagger\star}) \xrightarrow[i\to\infty]{\text{a.s.}} 0$ over all reachable transitions induced by capabilities in C.*

Proof. Since P and O are finite, there are finitely many reachable state-capability pairs. Suppose $h^\dagger$ remains inconsistent with $h^{\dagger\star}$ at state-capability pair (s, c). The hypotheses of Lemma 3 depend only on $\xi > 0$ and the reachability of (s, c), not on $|\mathcal{D}_{SC}(s, c)|$, so it applies afresh at each stage: after finitely many further iterations MCQS almost surely selects a query with non-zero probability of executing c in s, and by Lemma 5, the resulting observations have non-zero probability of reducing the inconsistency. Applying this successively yields $|\mathcal{D}_{SC}(s, c)| \to \infty$ unless (s, c) becomes consistent.

For infinitely many observations of (s, c), the MLE effect distribution converges almost surely to $p^\star$ by the strong law of large numbers, and by Lemma 4 each structural error is corrected with fixed non-zero probability per observation, hence almost surely after finitely many. The error at every reachable state-capability pair therefore converges almost surely to zero, and as there are finitely many such pairs this holds simultaneously for all of them almost surely. Since $\mathrm{VD}(h^\dagger_i, h^{\dagger\star})$ is the average of the absolute differences over the finitely many reachable transitions, each summand converging almost surely to zero gives $\mathrm{VD}(h^\dagger_i, h^{\dagger\star}) \xrightarrow[i\to\infty]{\text{a.s.}} 0$. □

A.3 Finite Transitions to Structural Convergence

We now prove Theorem 2 introduced in Section 4.1, which states that there exists a finite number of transitions that need to be collected to make the pessimistic and optimistic models equivalent.

Theorem 2. *Let $\mathcal{D}$ be the transitions observed by MCQS. If the true agent model $h^{\dagger\star}$ is expressible over vocabulary V, then there exists a finite set of transitions T_δ such that the pessimistic and optimistic models computed from $\mathcal{D} \uplus T_\delta$, denoted by $h^\dagger_\wedge$ and $h^\dagger_\vee$, respectively, satisfy $h^\dagger_\wedge \equiv h^\dagger_\vee$.*

Proof. Since P and O are finite, the set of states and capabilities are finite, and hence so is the set of state-capability-state transitions. Therefore, $T_{h^{\dagger\star}}$ is also finite.

Let $T_\mathcal{D}$ denote the unique set of transitions observed in $\mathcal{D}$. Since every observed transition must be realizable by the true model, $T_\mathcal{D} \subseteq T_{h^{\dagger\star}}$. Because $T_{h^{\dagger\star}}$ is finite, the set of unobserved transitions $T_\delta = T_{h^{\dagger\star}} \setminus T_\mathcal{D}$ is also finite.

Let $\mathcal{D}' = \mathcal{D} \uplus T_\delta$ denote the updated dataset with all transitions in T_δ observed. This results in $\mathcal{D}'$ having the same set of unique transitions as the true model $T_{\mathcal{D}'} = T_\mathcal{D} \cup T_\delta = T_{h^{\dagger\star}}$. Let $h^\dagger_\wedge$ and $h^\dagger_\vee$ be the pessimistic and optimistic models constructed from $\mathcal{D}'$, respectively.

By construction, $h^\dagger_\wedge$ admits exactly the observed transitions so $T_{h^\dagger_\wedge} = T_{D'} = T_{h^{\dagger\star}}$. Since $h^\dagger_\wedge$ is possibly sound and $h^\dagger_\vee$ is the join of all possibly sound hypotheses, $T_{h^{\dagger\star}} = T_{h^\dagger_\wedge} \subseteq T_{h^\dagger_\vee}$.

Suppose, for contradiction, that there exists a transition $\langle s, c, s'\rangle \in T_{h^\dagger_\vee} \setminus T_{h^{\dagger\star}}$. Since $h^\dagger_\vee$ is possibly sound, this can

only occur when (s, c) is unobserved in $\mathcal{D}$. However, it is always valid to ask an agent to achieve an intent, so every state–capability pair admits at least one transition in $T_{h^{\dagger\star}}$. Since $T_{\mathcal{D}'} = T_{h^{\dagger\star}}$, that transition is observed in $\mathcal{D}'$, contradicting the assumption that (s, c) is unobserved.

Therefore, $T_{h'_\vee} = T_{h^{\dagger\star}} = T_{h'_\wedge}$, and thus $h'_\wedge \equiv h'_\vee$. □

A.4 Convergence of Consistent Models

We now prove that, when $\mathcal{D}$ is transition and effect complete, if $h^\dagger_\wedge \equiv h^\dagger_\vee$ then (1) both $h^\dagger_\wedge$ and $h^\dagger_\vee$ are equivalent to $h^{\dagger\star}$; (2) a model $h^\dagger$ that is complete and possibly sound w.r.t. $\mathcal{D}$ is also equivalent to $h^{\dagger\star}$. We first prove that the pessimistic and optimistic models are equivalent to the true model when these conditions hold.

Theorem 4. *Let $h^{\dagger\star}$ be the true agent capability model, and assume $h^{\dagger\star}$ is expressible over vocabulary V. Let $\mathcal{D}$ be transition and effect complete. Then, at any stage of MCQS, if $h^\dagger_\wedge \equiv h^\dagger_\vee$, it follows that $h_\wedge \equiv h^\dagger_\vee \equiv h^{\dagger\star}$.*

Proof. Since $h^{\dagger\star}$ is expressible in V, $T_{h^{\dagger\star}}$ is the consistent.

By the premises, $h^\dagger_\wedge \equiv h^\dagger_\vee$ meaning $T_{h^\dagger_\wedge} = T_{h^\dagger_\vee} = T_c$. Suppose, for contradiction, that $T_c \neq T_{h^{\dagger\star}}$. Then either there exists a transition $\langle s, c, s'\rangle \in T_{h^{\dagger\star}} \setminus T_c$, or there exists a transition $\langle s, c, s'\rangle \in T_c \setminus T_{h^{\dagger\star}}$.

First, consider $\langle s, c, s'\rangle \in T_{h^{\dagger\star}} \setminus T_c$. If (s, c) had previously been observed, then, by the premise that $\mathcal{D}$ is transition complete, all possible effects of executing c in s would have been sampled. Since $\langle s, c, s'\rangle \in T_{h^{\dagger\star}}$, the transition would therefore have been observed, contradicting $\langle s, c, s'\rangle \notin T_c$. Hence, (s, c) must be unobserved.

By the premises that $\mathcal{D}$ is effect complete, there exists a $\langle s_1, c, s_2\rangle \in \mathcal{D}$ s.t. $\eta(s, s') = \eta(s_1, s_2)$. By construction of the optimistic model, there must exist a rule r s.t. $\eta(s, s') \in \text{effects}(r)$. Since (s, c) is unobserved, it cannot be assigned to any rule in the optimistic model. Therefore, by the construction of the optimistic model, $s_1 \models \text{ocond}(r)$, where $\text{ocond}(r)$ is the optimistic condition of r. Therefore, $\langle s, c, s'\rangle$ is consistent under $h^\dagger_\vee$. In contrast, $h^\dagger_\wedge$ classifies only observed transitions as consistent and therefore does not classify $\langle s, c, s'\rangle$ as consistent. Thus, $h^\dagger_\wedge \not\equiv h^\dagger_\vee$, yielding a contradiction.

Now consider $\langle s, c, s'\rangle \in T_c \setminus T_{h^{\dagger\star}}$. By construction, $h^\dagger_\wedge$ classifies only observed transitions as consistent, so $\langle s, c, s'\rangle$ must have been observed. Every observed transition is realizable under the true model, implying $\langle s, c, s'\rangle \in T_{h^{\dagger\star}}$, again yielding a contradiction.

Therefore, $T_c = T_{h^{\dagger\star}}$, and hence $h^\dagger_\wedge \equiv h^\dagger_\vee \equiv h^{\dagger\star}$. □

Due to the pessimistic and optimistic models bounding the lattice of consistent hypotheses when $\mathcal{D}$ is transition and effect complete, we can now prove that any other consistent hypothesis must also be equivalent to the true model when the pessimistic and optimistic models are equivalent.

Theorem 5. *Let $h^\dagger$ be any model of the discovered capabilities C that is complete and possibly sound with respect to the dataset $\mathcal{D}$, collected during a run of MCQS, in which $\mathcal{D}$ is transition and effect complete. If the true model $h^{\dagger\star}$ is expressible over vocabulary V, and the pessimistic and optimistic models $h^\dagger_\wedge$ and $h^\dagger_\vee$ learned from $\mathcal{D}$ using MCQS satisfy $h^\dagger_\wedge \equiv h^\dagger_\vee$, then $h^\dagger_\wedge \equiv h^\dagger_\vee \equiv h^\dagger \equiv h^{\dagger\star}$.*

Proof. Since $h^\dagger$ is complete with respect to $\mathcal{D}$, and $h_\wedge$ is the meet of all complete hypotheses, we have $h_\wedge \preceq h^\dagger$. Additionally, since $h^\dagger$ is possibly sound with respect to $\mathcal{D}$, and $h_\vee$ is the join of all possibly sound hypotheses, we have $h^\dagger \preceq h_\vee$.

When $h_\wedge \equiv h_\vee$, these inequalities imply $h_\wedge \equiv h^\dagger \equiv h_\vee$.

By Theorem 4, convergence of the pessimistic and optimistic models when $\mathcal{D}$ is transition and effect complete implies that both coincide with the true model. Therefore, $h_\wedge \equiv h_\vee \equiv h^\dagger \equiv h^{\dagger\star}$. □

B Comparison With Action Model Learning

As discussed in the Introduction, action model learning and capability model learning differ in several key ways, summarized in Table 1. Action model learning assumes access to, or seeks to recover, a model of the agent's primitive actions. In contrast, capability model learning does not require knowing the agent's action set, nor does it require observing and modeling the agent's behavior in the environment's vocabulary. Instead, it characterizes the long-horizon outcomes the agent can bring about through its planning and reasoning. Thus, action model learning models the effects of actions available in an environment, whereas capability model learning models what a particular agent is capable of achieving.

C Implementation of Compact Distributions Over States

This section provides implementation details regarding our representation of a distribution over states. For illustration purposes, let h_c be a hypothesis about capability c, $r \in h_c$ be a conditional effect rule, $\text{cond}(r)$ be the well-formed formula corresponding to the condition for r, and $\text{effects}(r) = \{(p_i, \eta_i)\}$ be the set of probabilistic effects of r. p_i is the probability of η_i occurring, and η_i is a conjunction of problem literals.

States as Bit Vectors We represent a problem state x as a binary vector, where the j'th bit corresponds to the truth value of the j'th problem literal assignment of x according to some fixed ordering. We represent a distribution over possible states S using a hash map $S : x \to \log P(x)$: keys are states x in bit vector form, and values are the log-probability mass associated with the state $\log P(x)$. This representation is sparse; we only store states with probability mass greater than zero so as not to materialize the exponential state space when possible.

Capability Conditions as DNFs We represent a capability condition $\text{cond}(r)$ as either a Disjunctive Normal Form (DNF) formula or the negation of one. Each clause $\text{clause}_k \in \text{cond}(r)$ is a conjunction of literals stored as a bit vector, similarly to how states are stored. This enables efficient checking of whether a state satisfies a capability condition via efficient bitwise operations.

	Action Model Learning	Capability Learning
Works without input action headers	✗	✔
Discovers and predicts agent capabilities	✗	✔
Models capture long-horizon execution	✗	✔
Observation and modeling over different vocabularies	✗	✔
Evaluates planning and reasoning capabilities	✗	✔
Models feature arbitrary condition-effect rules	✗	✔
Models primitive actions of agents	✔	✗

Table 1: Key differences between traditional action model learning and learning intent-driven capabilities.

Effects as Bit Vectors Similar to how we represent states, we represent probabilistic effect outcomes η_i as bit vectors, with each bit corresponding to a literal truth value. Furthermore, we store a binary mask mask_i that is 1 in position k when η_i affects literal k and 0 otherwise.

State Distribution Updates Let $x \models \text{cond}(r)$ denote that x is a model of $\text{cond}(r)$, and $S_1 + S_2 = S$ denote the distribution obtained by taking the sum of distributions S_1 and S_2. Utilizing this efficient satisfiability check, we update a state distribution using the following algorithm:

Algorithm 1: Update State Distribution

Inputs: state distribution S, conditional effect r
Output: new state distribution S'
$S_{change} \leftarrow \{(x_k \rightarrow \log P(x_k)) | x_k \models \text{cond}(r)\}$
$S_{nochange} \leftarrow \{(x_k \rightarrow \log P(x_k)) | x_k \not\models \text{cond}(r)\}$
$S' \leftarrow S_{nochange}$
for $(p_i, \eta_i, \text{mask}_i) \in \text{effects}(r)$ **do**
    $S_i \leftarrow \{((x_k \wedge \text{mask}_i) \vee \eta_i \rightarrow \log[P(x_k) * p_i]) | (x_i \rightarrow \log P(x_i)) \in S_{change}\}$
    $S' \leftarrow S' + S_i$
end for
return S'

D Additional Agent and Environment Details

In this section, we go through each evaluated agent and environment, and provide additional details on how they were implemented.

D.1 MiniGrid

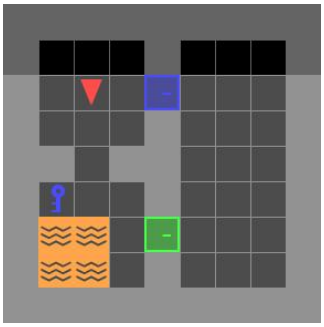


Figure 5: Minigrid environment evaluated

Environment We use a 9×9 MiniGrid (shown in Fig. 5) that contains walls, lava, a blue key, a locked green door, and a locked blue door. The grid is divided into 4 quadrants, with a single-tile path from each quadrant to each adjacent one. The blue door is between the northwest and northeast quadrants, and the green door is between the southwest and southeast quadrants. The blue key is located in the southwest quadrant near the lava. Finally, the agent starts in the northwest quadrant but cannot drop the key.

Therefore, there are some interesting capabilities in this domain. The agent can traverse near the lava to pickup the blue key, use the blue key to unlock the blue door, open/close the blue door, and traverse to any of the locations.

Symbolic Representation As mentioned, instead of using (x,y) locations in the grid, the locations of the key and agent are tracked based on their quadrant. For capabilities, a location in the quadrant is sampled to give as the intent to the agent.

Predicate	Description
`agent-at(q)`	The agent is in quadrant q
`blue-key-at(q)`	The blue key is in quadrant q
`carrying-blue-key()`	The agent is carrying the blue key
`is-dead()`	The agent is dead
`green-door-open()`	The green door is open
`green-door-locked()`	The green door is locked
`green-door-closed()`	The green door is closed
`blue-door-open()`	The blue door is open
`blue-door-locked()`	The blue door is locked
`blue-door-closed()`	The blue door is closed

Table 2: Predicate vocabulary used in MiniGrid.

ReAct Agent The ReAct agent is designed to describe the current state in text, prompt the agent to achieve the objective, and re-plan if necessary.

The base prompt given to the LLM uses the height, width, and action names, specifies the actions and rules, and gives a couple of examples.

```
"You are navigating a {height}x{width}
grid. Rows 1-{height} increase going
south; columns 1-{width} increase going
east. Coordinate format: (row, col).
```

GRID LEGEND
==========
 . = empty cell
 W = wall (impassable)
 L = lava (instant death -- NEVER step here)
bK = blue key (first letter = color)
bD = blue door (locked/closed)
b= = blue door (open)
gD = green door (locked/closed)
g= = green door (open)
A> = you, facing east (^ north, v south, < west, > east)

ACTIONS
=======
{action_names}

- face_north / face_south / face_east / face_west : rotate to face that compass direction (no movement).
- move_forward : move one cell in the direction you face. Blocked by walls, closed/locked doors, objects, and lava.
- pickup : pick up the object in the cell you face (you must be adjacent and facing it). You can carry only one object.
- toggle : open or close the door in the cell you face. Locked doors require holding the matching color key.
- done : signal that the goal is achieved or no further action is possible.

RULES
=====
Movement:
- move_forward moves you one cell in the direction you face.
- You CANNOT move into walls (W), lava (L), closed/locked doors, or cells with objects.
- Lava kills instantly -- NEVER step into a lava cell.
- You cannot move into a cell occupied by an object; pick it up from an adjacent cell instead.

Interaction:
- You must be adjacent to and facing a cell to interact with it (pickup or toggle).
- You can carry at most one object at a time.

Doors and keys:
- Doors block movement when closed or locked, just like walls.
- A locked door is opened by holding the matching COLOR key and using toggle. A blue key opens a blue door, a green key opens a green door, etc.
- Once opened, a door becomes a passable cell -- you can walk through it freely.
- IMPORTANT: doors are the ONLY way to cross through wall barriers. Before concluding a goal is unreachable, check every door on the map and whether you have (or can get) the matching key.

Planning:
- Before acting, scan the grid for barriers (walls, lava) and identify a all possible paths including through doors.
- For each move_forward in your plan, verify the target cell is empty or an open door.
- When the goal is on the other side of a wall, find a door that crosses the barrier and plan how to unlock it.

EX 1 -- Navigate and pick up an object
======================================
Observation:
```
     1  2  3  4  5
  1  .  .  .  .  .
  2  . Av  .  .  .
  3  .  .  . bK  .
  4  .  .  W  W  .
  5  .  .  .  .  .
```
Agent: (2, 2) facing south
Carrying: nothing

Goal: Pick up the blue key.

Thought:
I am at (2, 2) facing south. The blue key is at (3, 4).
Plan a path: go south one step to (3, 2), then east toward column 4.
- move_forward: I face south, so I move to (3, 2). Cell is empty -- safe.
- face_east: I now face east at (3, 2).
- move_forward: move to (3, 3). Cell is empty -- safe.
- move_forward: (3, 4) has the blue key -- BLOCKED, I cannot move into an occupied cell.
  Instead I stay at (3, 3) facing east, and the key is in the cell I face.
- pickup: pick up the blue key from (3, 4).

Actions:

```
move_forward
face_east
move_forward
pickup

EXAMPLE 2 -- Unlock a door
==========================
Observation:
     1  2  3  4  5
  1  .  .  .  .  .
  2  .  .  A>  .  .
  3  .  .  W bD  .
  4  .  .  .  .  .
  5  .  .  .  .  .
Agent: (2, 3) facing east
Carrying: a blue key

Goal: Open the blue door.

Thought:
I am at (2, 3) facing east, carrying
the blue key. The blue door is at
(3, 4).
I need to be adjacent to the door and
facing it. From (2, 4) I could face
south toward (3, 4).
- move_forward: move east to (2, 4).
Cell is empty -- safe.
- face_south: now I face south at
(2, 4). The cell ahead is (3, 4) which
is the blue door.
- toggle: I hold the blue key so the
locked blue door unlocks and opens.

Actions:
move_forward
face_south
toggle

OUTPUT FORMAT
=============
Always output exactly:
Thought: <your step-by-step reasoning
-- trace each action, verify the
target cell is safe>
Actions:
<one action per line from the action
list above, max {max_plan_length}
actions>
"
```

This prompt is used as the system prompt, followed by a description of the task and the state. It then parses out the actions to take. In the case of an invalid action, the following prompt is provided.

```
"I could not parse your previous
response. Please reply again using
exactly the required format:
Thought: <your reasoning>
Actions:
<one action per line>

Valid actions: {action_names}

Each line under Actions: must contain
exactly one action name from the
list above, nothing else. If the goal
is already achieved or impossible,
output a single action: done"
```

Then the policy is executed until either the maximum number of actions is reached or the LLM returns the "done" action.

D.2 Overcooked

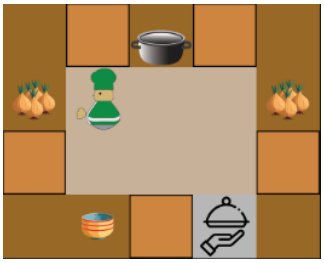


Figure 6: Overcooked environment evaluated.

Environment We use Overcooked, a cooperative kitchen gridworld environment. At each timestep, the agent selects an action from a discrete action space, thereby updating the environment state and producing a new observation.

The agent must navigate, collect ingredients, prepare dishes, and deliver them. The task involves a multi-step process: collect the onion, add it to the pot, wait for it to cook, retrieve the soup, and deliver. The environment includes stochastic elements, namely a 5-stage cooking process.

We use a single predefined initial state (shown in Fig. 6) that specifies the kitchen layout, the agent's position, the ingredient pile locations, the pot placement, and the delivery counter.

Attribute	Description
kitchen_layout	Fixed configuration of counters and tiles
agent_position	Initial position of the agent in the grid
ingredient_piles	Location of onion piles
pot_location	Placement of cooking pot
delivery_counter	Location of delivery area

Table 3: Attributes defining the predefined initial state.

Symbolic Representation We construct a symbolic state representation that maps low-level environment states to a set of logical predicates. The representation operates on the

structured state exposed by the Overcooked environment, including agent location, held items, pot state, and object positions.

Given a state, the symbolic representation provides a set of ground literals over a fixed predicate vocabulary. Objects are limited to the agent (chef1), items (onion, soup-dish), locations (onion-pile-1, onion-pile-2, pot, delivery), and pot cooking stages.

Each predicate is computed directly from the underlying state. Position predicates (e.g., `at`) are obtained by querying the agent's location, while state predicates (e.g., `pot-state`, `holding`) are determined from the environment's internal state representation.

Predicate	Description
`at(x, location)`	Entity x is at location
`holding(agent, x)`	Agent possesses item x
`in-pot(x)`	Item x is in the pot
`pot-state(state)`	Current cooking stage
`clean(dish, location)`	Clean dish at location
`at-pile(onion, pile)`	Onion at specified pile
`delivered(x)`	Item x has been delivered

Table 4: Predicate vocabulary used in the symbolic representation.

HDDLGym Agent We use an agent trained using hierarchical reinforcement learning from HDDLGym (La, Mon-Williams, and Shah 2025). The agent operates in single-agent mode in the Overcooked environment.

The agent receives high-level capability intents and executes learned low-level policies to achieve these intents. Our experiments show that spurious correlations in training data can induce unintended behaviors.

MCQS discovered 5 capabilities: get onion (c-0), add onion to pot (c-1), cooking progression (c-2), retrieve soup (c-3), and deliver soup (c-4). These capabilities form a linear dependency chain where each is the sole producer of its output state. The cooking capability (c-2) is the only temporal capability with length 5, modeling the stochastic multi-stage cooking process with probabilistic progression through cooking stages.

D.3 Crafter

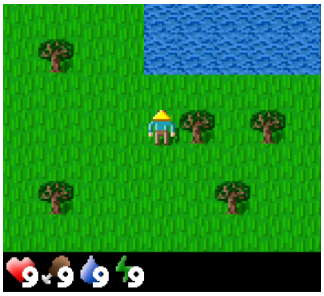


Figure 7: Crafter environment evaluated.

Environment We use Crafter, a grid-based survival and crafting environment. At each timestep, the agent selects an action from a discrete action space, thereby updating the environment state and producing a new observation.

The agent must collect resources, craft tools, and interact with terrain and objects. Reward is given based on the achievements unlocked by the agent; there are 22 such achievements in the Crafter world.

We use a fixed 9×9 grid initial state (shown in Fig. 7) containing grass, tree, and water locations with fixed stats for the player. This state was specified using the structured state representation provided by Crafter-OO (Hafner 2022; Khan et al. 2025).

Attribute	Description
`area`	Size of the environment grid
`player_position`	Initial position of the agent in the grid
`player_facing`	Initial orientation of the agent
`terrain_layout`	Fixed terrain configuration of the environment
`objects`	Initial placement of objects in the environment

Table 5: Attributes defining the predefined initial states.

Symbolic Representation We construct a symbolic state representation that maps low-level environment states to a set of logical predicates. The representation operates on the structured state exposed by Crafter-OO, including terrain, objects, player attributes, inventory, and achievements.

Given a state, the symbolic representation provides a set of ground literals over a fixed predicate vocabulary. Objects are limited to a predefined set of obstacles, tools, structures, and achievements. This symbolic state is not a perfect representation of the underlying environment state, as it retains only the information necessary for decision-making and removes irrelevant details.

Each predicate is computed directly from the underlying state. Some predicates are related to position (e.g., `next_to`, `close_to`), which are obtained by calculating the Manhattan distances between the agent and nearby

objects, while other predicates (e.g., `has`, `can_make`, `can_build`) are determined from inventory contents.

We specify the goal as a symbolic literal conjunction over this set of predicates.

Predicate	Description
`next_to(x)`	Agent is adjacent to object x
`close_to(x)`	Agent is within a Manhattan distance of 3 from x
`exists(x)`	Object x exists in the environment
`is_sleeping()`	Agent is sleeping
`has(x)`	Agent possesses tool x
`can_make(x)`	Agent has resources to craft x
`can_build(x)`	Agent has resources to build x
`achieved(x)`	Achievement x has been completed

Table 6: Predicate vocabulary used in the symbolic representation.

Qwen-Steve Agent We use an agent that combines the Qwen2.5-7B-Instruct (Team 2024) model with Crafter Steve1 from CrafterDojo to execute skills. Steve-1 executes higher-level skills as sequences of low-level Crafter actions.

The agent operates over a fixed subset of skills derived from the CrafterDojo skill set. CrafterDojo defines 61 skills; we use a subset of 16.

- `craft wood pickaxe`
- `craft wood sword`
- `dig a tunnel`
- `go explore`
- `go to sleep`
- `move to east`
- `move to north`
- `move to south`
- `move to west`
- `obtain coal`
- `obtain diamond`
- `obtain tree`
- `obtain water`
- `place crafting table on grass`
- `place crafting table to build shelter`
- `stay`

The agent takes a symbolic goal and converts it into task text. At each step, the current environment state is mapped to a symbolic state representation and then converted into state text. The task text and state text are passed to the Qwen model to select a skill.

Given the task text, state text, and candidate skills, the agent uses Qwen to select the next skill to execute. This selection is repeated after each skill attempt, allowing the agent to adapt based on the updated environment state.

The selected skill is executed by CSteve1 for multiple timesteps. We use greedy action selection during skill execution, making CSteve1's behavior deterministic. After each environment step, the agent updates the symbolic state and checks whether the original symbolic goal is satisfied. If the goal is achieved, the agent returns success immediately. If not, CSteve1 continues executing the current skill until the fixed step limit for that skill is reached.

After a skill attempt finishes, if the symbolic goal is still not achieved, the agent selects a skill again using the updated state and the same original goal. The number of skill selection attempts is fixed. If the goal is still not achieved after that limit, the agent terminates and returns failure.

Skill Selection We score a set of candidate skills using the LLM to select the skill to be executed. We convert the goal and current-state representations into text, then evaluate each skill independently.

For each skill, we prompt the Qwen model using the goal description and state text, along with the skill name, and ask whether the skill is the best to execute to achieve the goal from the current state. We calculate two log-probabilities: $\log P(\text{yes})$ — the probability of the model answering yes for the prompt, and $\log P(\text{no})$ — the probability of the model answering no for the prompt. We calculate the difference between $\log P(\text{yes})$ and $\log P(\text{no})$ to score each individual skill. A high score means the model thinks the skill is more suitable for execution than not. We then pick the skill with the highest score.

An example prompt used for scoring a candidate skill is:

```
" Goal text: [goal description]
  Current state: [state description]
  Candidate skill: [skill name]
  Is this candidate the best skill to
  execute next for the goal?
  Answer yes or no."
```

D.4 PDDLGym

PDDLGym (Silver and Chitnis 2020) is a benchmark suite of planning environments based on the Planning Domain Definition Language (PDDL). In PDDLGym, each environment is specified by a PDDL domain file and a PDDL problem file. The domain file defines the predicate vocabulary and object types, while the problem file defines the objects, initial state, and goal condition. We use five PDDLGym environments: Blocksworld, Depot, First Responders, Tireworld, and Probabilistic Elevators.

Blocksworld Blocksworld is a block-stacking environment in which blocks are rearranged on a table using a robot hand.

Predicate	**Meaning**
`on(x, y)`	Block `x` is on block `y`.
`ontable(x)`	Block `x` is on the table.
`clear(x)`	Block `x` has no block on top of it.
`handempty(r)`	Robot `r` is not holding a block.
`handfull(r)`	Robot `r` is holding a block.
`holding(x)`	Block `x` is being held.

Table 7: Predicates for Blocksworld.

Component	Specification
Objects	a, b, and c of type block; robot of type robot.
Initial state	clear(a), clear(b), clear(c), ontable(a), ontable(b), ontable(c), and handempty(robot).

Table 8: Problem instance used for Blocksworld.

First Responders First Responders is an emergency-response environment involving fire units, medical units, victims, fires, hospitals, and water sources.

Predicate	Meaning
fire(l)	Location l has a fire.
nfire(l)	Location l does not have a fire.
victim-at(v, l)	Victim v is at location l.
victim-healthy(v)	Victim v is healthy.
victim-hurt(v)	Victim v is hurt.
victim-dying(v)	Victim v is dying.
hospital(l)	Location l is a hospital.
water-at(l)	Water is available at location l.
adjacent(l1, l2)	Location l1 is adjacent to location l2.
fire-unit-at(u, l)	Fire unit u is at location l.
medical-unit-at(u, l)	Medical unit u is at location l.
have-water(u)	Fire unit u has water.
have-victim-in-unit(v, u)	Victim v is inside medical unit u.

Table 9: Predicates for First Responders.

Component	Specification
Objects	l1 and l2 of type location; f1 of type fire_unit; m1 of type medical_unit; z1 and z2 of type victim.
Initial state	hospital(l2), water-at(l1), fire(l1), fire(l2), victim-at(z1, l2), victim-hurt(z1), victim-at(z2, l2), victim-dying(z2), adjacent(l1, l2), adjacent(l2, l1), fire-unit-at(f1, l1), and medical-unit-at(m1, l2).

Table 10: Problem instance used for First Responders.

Tireworld Tireworld is a navigation environment in which a vehicle moves between locations and may need to change a flat tire using spare tires.

Predicate	Meaning
vehicle-at(loc)	The vehicle is at location loc.
spare-in(loc)	A spare tire is available at location loc.
road(from, to)	There is a road from location from to location to.
not-flattire()	The vehicle does not currently have a flat tire.

Table 11: Predicates for Tireworld.

Component	Specification
Objects	l-1-1, l-1-2, l-1-3, l-2-1, l-2-2, and l-3-1 of type location.
Initial state	vehicle-at(l-2-1), road(l-1-1, l-1-2), road(l-1-2, l-1-3), road(l-1-1, l-2-1), road(l-1-2, l-2-2), road(l-2-1, l-1-2), road(l-2-2, l-1-3), road(l-2-1, l-3-1), road(l-3-1, l-2-2), spare-in(l-2-1), spare-in(l-2-2), spare-in(l-3-1), and not-flattire().

Table 12: Problem instance used for Tireworld.

Probabilistic Elevators Probabilistic Elevators is an elevator-navigation environment in which the agent moves across floors and positions, enters and exits elevators, and collects coins.

Predicate	Meaning
dec_f(f, g)	Floor f is one step below floor g.
dec_p(p, q)	Position p is one step left of position q.
in(e, f)	Elevator e is at floor f.
at(f, p)	The agent is at floor f and position p.
shaft(e, p)	Elevator e is associated with shaft position p.
inside(e)	The agent is inside elevator e.
gate(f, p)	There is a gate at floor f and position p.
coin-at(c, f, p)	Coin c is at floor f and position p.
have(c)	The agent has collected coin c.
underground()	The agent is underground.
is-first-floor(f)	Floor f is the first floor.
is-first-position(p)	Position p is the first position.

Table 13: Predicates for Probabilistic Elevators.

Component	Specification
Objects	`f1`, `f2`, and `f3` of type `floor`; `p1`, `p2`, `p3`, and `p4` of type `pos`; `e1` and `e2` of type `elevator`; `c1`, `c2`, and `c3` of type `coin`.
Initial state	`is-first-floor(f1)`, `is-first-position(p1)`, `underground()`, `dec_f(f2, f1)`, `dec_f(f3, f2)`, `dec_p(p2, p1)`, `dec_p(p3, p2)`, `dec_p(p4, p3)`, `shaft(e1, p3)`, `in(e1, f1)`, `shaft(e2, p3)`, `in(e2, f1)`, `coin-at(c1, f2, p3)`, `coin-at(c2, f3, p3)`, `coin-at(c3, f1, p1)`, `gate(f2, p4)`, `gate(f3, p3)`, and `gate(f3, p4)`.

Table 14: Problem instance used for Probabilistic Elevators.

D.5 SayCan

Environment The SayCan agent (Ahn et al. 2022) is a mobile manipulator in a PyBullet simulation environment. It interacts with four blocks and one cup on a tabletop. The agent is equipped with low-level skills (e.g., pick-and-place a yellow block). It uses a controlled based on Llama-3.1-8B-Instruct (AI 2024) to select a skill to execute based on its abstract description. MCQS uses the same abstraction function for learning the capability model.

Predicate	Meaning
`on(x, y)`	Block `x` is on block `y`.
`on-table(x)`	Block `x` is on the table.
`clear(x)`	Block `x` has no block on top of it.

Table 15: Predicates for SayCan.

Symbolic Representation Table 15 shows the predicates used in Saycan. An important difference between Saycan and Blocksworld is that SayCan can push blocks off the table, causing them to disappear.

Existing Results In a previous version of MCQS, we evaluated on SayCan. The capabilities learned in those runs are the ones reported in Fig. 1b. These models were learned using the same capability-learning code as the current version of MCQS; however, both MCQS-E and MCQS-S have since been substantially improved. Therefore, we do not report variational distance results for the SayCan models. Instead, we include these learned capabilities only as examples of the types of capabilities that MCQS can learn.

E Implementation Details and Setup

In this section we discuss the implementation details and setup.

E.1 MCQS Algorithm

Algorithm 2: **Monte Carlo Query Search (MCQS)**

1: **Inputs:** BBAI $\mathcal{A}$, simulator $\mathcal{S}_\mathcal{E}$, representation $\alpha : \mathcal{X} \to S$
2: **Output:** pessimistic capability model $h^\dagger$
3: $\mathcal{T}_0 \leftarrow random\_walk(\mathcal{S}_\mathcal{E})$
4: $C \leftarrow discover\_capabilities(\mathcal{T}_0, \alpha)$
5: $\mathcal{D} \leftarrow initialize\_dataset(C)$
6: $h^\dagger_\wedge, h^\dagger_\vee \leftarrow construct\_models(C, \mathcal{D})$
7: $x_i \leftarrow reset(\mathcal{S}_\mathcal{E})$
8: **while** not $stop\_condition()$ **do**
9: $\quad \xi \leftarrow set\_revisitation\_incentive()$
10: $\quad \pi \leftarrow synthesize\_query(\alpha(x_i), h^\dagger_\wedge, h^\dagger_\vee, \xi)$
11: $\quad \overline{\overline{x}} \leftarrow execute\_query(\mathcal{A}, \mathcal{S}_\mathcal{E}, x_i, \pi)$
12: $\quad \mathcal{D} \leftarrow update\_dataset(C, \mathcal{D}, \overline{\overline{x}}, \alpha)$
13: $\quad C \leftarrow update\_capabilities(C, \overline{\overline{x}}, \alpha)$
14: $\quad h^\dagger_\wedge, h^\dagger_\vee \leftarrow update(h^\dagger_\wedge, h^\dagger_\vee, C, \mathcal{D})$
15: $\quad x_i \leftarrow sample\_initial\_state(\overline{\overline{x}}, \mathcal{S}_\mathcal{E})$
16: **end while**
17: **return** $h^\dagger_\wedge$

In this section, we provide the full MCQS algorithm, shown in Alg. 2. As mentioned in Section 2.1, MCQS takes as input only the BBAI $\mathcal{A}$, the simulator $\mathcal{S}_\mathcal{E}$ for environment $\mathcal{E}$, and the representation function $\alpha : \mathcal{X} \to S$ (line 1). It outputs a capability model $h^\dagger$ for the capabilities discovered and learned for $\mathcal{A}$ in $\mathcal{E}$ (line 2).

As discussed in Sec. 3, MCQS begins by performing a random walk in $\mathcal{S}_\mathcal{E}$ to collect an initial set of low-level transitions $\mathcal{T}_0$, which are used to discover an initial capability set C (lines 3–4). The process for constructing capabilities is described in Sec. 3. The algorithm then initializes the dataset of observed state-capability-state transitions $\mathcal{D}$ from C and constructs the initial pessimistic and optimistic capability models, $h^\dagger_\wedge$ and $h^\dagger_\vee$, respectively (lines 5–6). The process for constructing $h^\dagger_\wedge$ and $h^\dagger_\vee$ is described in Sec. 3.1.

Until the stop condition is triggered (line 8), MCQS repeatedly synthesizes a query, executes it, and updates both the dataset and the learned hypotheses. In the first iteration, the query starts from the initial state of $\mathcal{S}_\mathcal{E}$ (line 7). In subsequent iterations, the initial state x_i is chosen by sampling (line 15): if the previous query policy was predicted to be distinguishing and the maximum number of observed changes in the symbolic state space is less than 100, then a resulting state from the previous query is reused; otherwise, a non-sink state from $\mathcal{D}$ is sampled with probability inversely weighted by the number of times it has been observed. Details on initial-state sampling are provided in Appendix E.4.

The state revisitation value ξ is then dynamically set based on how recently new information has been collected (line 9); details are provided in Appendix E.5. Given the initial symbolic state $s_i = \alpha(x_i)$, and the current models $h^\dagger_\wedge$ and $h^\dagger_\vee$, MCQS constructs a distinguishing MDP $\mathcal{P}^\dagger(s_i, h^\dagger_\wedge, h^\dagger_\vee)$ (Def. 6). This MDP is solved with ξ by either MCQS-S or

MCQS-E to produce a query policy $\pi : S \to C$ (line 10). For the random-query ablation used to evaluate MCQS in Sec. 4.2, a sequence of 30 capabilities is sampled instead.

The query policy is then executed 25 times from the initial low-level state x_i by resetting $\mathcal{S}_\mathcal{E}$ to x_i and having $\mathcal{A}$ follow the policy. At each symbolic state s, the next capability is selected as $c = \pi(s)$ and executed by $\mathcal{A}$. A rollout terminates if π does not specify a capability for the current state s or if more than 20 capabilities have been executed in the current query iteration. Thus, each query produces a set of trajectories, denoted $\overline{\overline{x}}$, for the capabilities executed during the query (line 11). This set is used to update both $\mathcal{D}$ and C (lines 12–13). The process for converting trajectories into transitions is described in Sec. 3.3. After this update, $h^\dagger_\wedge$ and $h^\dagger_\vee$ are updated (line 14).

As mentioned in Sec. 4.2, we output the pessimistic model $h^\dagger_\wedge$, since it provides a safer representation of the learned capabilities (line 16).

E.2 Dynamic policy synthesis

Both MCQS-E and MCQS-S synthesize non-stationary policies rather than fixed capability sequences, allowing execution to adapt to observed outcomes.

In MCQS-E, the returned policy contains the MCTS tree and tracks the current node. After a capability is selected, the current node advances to the corresponding child. Once the resulting state is observed, the policy performs a forward update from that node along the maximum-UCB continuation. During this pass, predicted state distributions are updated and impossible capabilities are pruned. The affected MCTS node statistics are then updated in a backward pass. If the maximum-UCB capability changes during this update, the newly selected branch is forward-updated and included in the backward pass.

MCQS-S represents states and capabilities separately and therefore constructs a bipartite policy graph $\langle N, E\rangle$, where nodes correspond to capabilities and edges correspond to observed states. Let $n_0 \in N$ denote the initial node. Let $f : N \to C$ label each node with a capability, and let $g : E \to S \times \mathbb{Z}^+$ label each edge with a state and its depth in the MCTS tree.

For each state node in the MCTS tree produced by MCQS-S, let s be the represented state at depth d, and let n_0 be the graph node corresponding to the capability executed to reach this state. The UCB-maximizing capability c is selected and a new node n_1 is added to N together with edge $e = (n_0, n_1)$, where $f(n_1) = c$ and $g(e) = (s, d)$. The algorithm then recursively proceeds to the child corresponding to c, carrying forward node n_1.

For a capability node, let n_0 denote the carried-forward node. All child state nodes are considered. For a state node labeled (s, d), if there exists an edge (n'_0, n'_1) with $g((n'_0, n'_1)) = (s, d)$, then edge (n_0, n'_1) is added. Otherwise, the algorithm recursively proceeds to that child while carrying forward n_0.

Let n be the current node in the graph. The next capability selected by the policy is $f(n)$. After observing the next state s', if there exists an edge $e = (n, n')$ such that $g(e) = (s', \ldots)$, then the current node is updated to n'.

E.3 MCQS-E Implementation

To improve the efficiency of MCQS-E, we introduce two optimizations. First, to prevent the tree from containing redundant branches, we prune any newly generated node whose support set is identical to that of an existing node. This avoids exploring multiple nodes that are effectively equivalent. Second, to reduce the frequency of distribution expansions and total-variation distance computations, we limit expansion to three child nodes when a node is first expanded, and again each time it is revisited. For rollout, we do 3 random policy rollouts.

E.4 Initial State Sampling

To encourage exploration of under-explored regions of the observed state space $S_o \subseteq S$, we sample the initial state for query synthesis and execution according to observation counts in $\mathcal{D}$. For each state $s \in S_o$, let $N(s; \mathcal{D}) = |\{\langle s, c, s'\rangle \in \mathcal{D}\}|$ denote the number of observed transitions from s. Let $n_{\max} = \max_{s \in S} N(s; \mathcal{D})$. We assign each state s sampling weight $w_s = (n_{\max} + 1 - N(s; \mathcal{D}))/\sum_{\tilde{s} \in S_o}(n_{\max} + 1 - N(\tilde{s}; \mathcal{D}))$.

When we are sampling a new initial state from the outcome of the previous query, the process is in the same, but we use only the observed resulting states from each iteration of the query instead of S_o.

E.5 Empirical Evaluation Information

For running MCQS, we considered many hyperparameters. In Table 16 we list all the additional hyperparameters we used when designing MCQS.

For ξ, we scale it according to the number of queries since the last significant update. Let n be the number of queries since the last observed information and let ξ_m be the maximum exploration constant. Then ξ is set as $\xi = \frac{n}{25}\xi_m$. Note 25 is the number of queries before we early stop, meaning ξ can never exceed ξ_m.

Setting ξ is an exploration–exploitation trade-off: if ξ is too large, hard-to-reach capabilities may be starved because reducing the missing-effect probability below a threshold may require exponentially many samples. In this work, we set $\xi_m = 10^{-5}$.

Finally, First responders contains many irreversible actions requiring execution from the initial state to reach specific states; therefore, for this domain, instead of initial-state sampling, we reset the simulator every 100 steps.

F Additional Results

As discussed in Sec. 4, we evaluated MCQS further by running an ablation where $\xi = 0$ on 3 PDDLGym domains. We also evaluated models produced by MCQS-S on there accuracy on reachability.

F.1 No Revisitation Incentive

We evaluated three PDDLGym domains (Blocksworld, Tireworld, and First Responders) using MCQS-E and MCQS-S

Description	Value
Number of runs per query	25
Environment state horizon	100
Warm Start Random Capability Walks	0
Max Capability Sequence	20
MCTS Exploration Constant	$\sqrt{2}$
MCTS Iteration Count	1000
Random Policy if no distinguishing policy found	True
Early Stop Condition	25 Queries with no new information

Table 16: Hyperparameters used for setting MCQS-E and MCQS-S

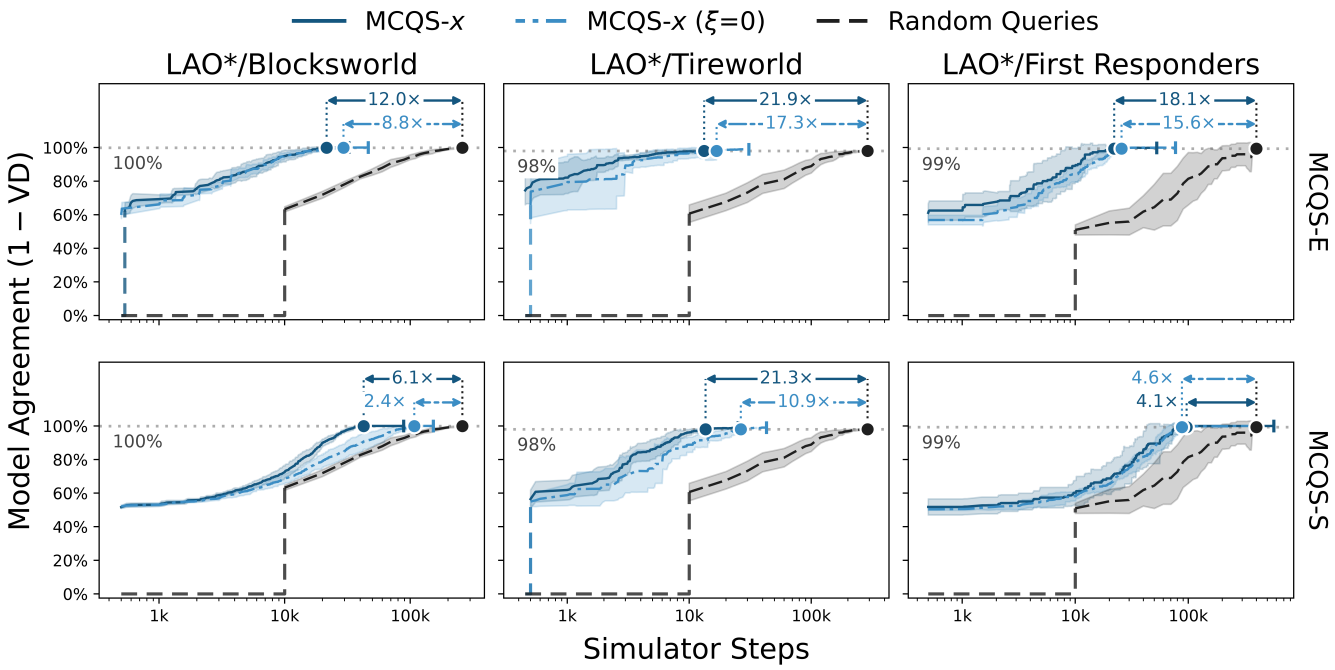


Figure 8: Model agreement (1 - weighted VD) of MCQS-E and MCQS-S on three evaluation problems. Shaded regions indicate one standard deviation over multiple runs. Annotated arrows indicate the simulator step-count ratio between each MCQS approach and the Random Query ablation baseline.

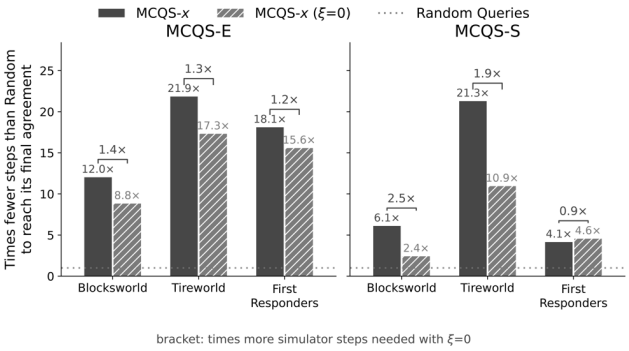


Figure 9: Times fewer simulator steps MCQS-S and MCQS-E had on three evaluation problems compared to random querying.

with the parameters from the main paper and with the revisitation incentive disabled ($\xi = 0$). We additionally evaluated Random Querying as a baseline. We report averages over five independent runs.

Fig. 8 shows that MCQS learns accurate capability models both with and without the revisitation incentive. In most settings, MCQS achieved at least 98% model agreement with a non-zero ξ with fewer simulator steps than with $\xi = 0$, suggesting that the revisitation incentive can improve the quality of the policies tried. However, the effect is not uniform: MCQS-S is more sensitive to ξ, and in some domains a non-zero ξ requires substantially more simulator steps before stopping. Thus, while ξ can improve sample quality, setting it too high may increase the total model learning cost.

This trade-off is highlighted by Fig. 9, which shows the

number of simulator steps until the stopping condition is triggered. For MCQS-E, using a non-zero ξ required 20% to 40% more simulator steps than $\xi = 0$. For MCQS-S, the effect ranged from 10% fewer steps to 150% more steps, depending on the domain. Overall, ξ can lead to better intermediate policies and faster agreement, but may also increase the total simulator steps needed before termination.

F.2 Reachability Methodology

To evaluate the reachability predictions of a model $h^\dagger$ learned by MCQS-S, we uniformly sample an initial state s_i from the observed states S_{observed}. We then uniformly sample a capability c that $h^\dagger$ predicts is achievable from s_i and sample a possible successor state from $h^\dagger$. From this predicted state, we repeat the process until constructing a sequence of five capabilities. We reject sequences for which no valid capability is available before reaching length five.

We execute each sequence 100 times on the BBAI $\mathcal{A}$, resetting the simulator to s_i before each execution. For each capability c in the sequence, $\mathcal{A}$ attempts to achieve the intent associated with c. Execution continues until the sequence is completed or $h^\dagger$ classifies an observed transition as inconsistent with its prediction.

For each sequence, we compute the fraction of the 100 executions for which $h^\dagger$ remains consistent throughout the entire sequence. We report the mean and standard deviation of this accuracy across sampled sequences.

In Crafter, due to CSteve1 being set to greedy and evaluating a temporal length of 1, most capabilities were invalid resulting in only a handfull of valid 5-capability sequence. Therefore, we ran MCQS on Crafter with a temporal length of 4 and set the agent to stochastic. While this creates more opportunities for the agent to achieve the intent, it also increases the difficulty of capability model learning due to a significant increase in stochasticity. However, as reported, MCQS-S was able to learn a model that had a reachability accuracy of $78\% \pm 16\%$.

G QACE Baseline

In this section we describe how we ran QACE (Verma, Karia, and Srivastava 2023), the closest existing approach to the problem addressed in this paper. We describe the inputs we supplied, the compilation required to express our learning targets in QACE's model class, and the reasons QACE can be run only on symbolic agents.

G.1 Inputs and setup

QACE does not discover capabilities. It requires a predicate vocabulary V and a list of capabilities, each identified with an intent, and learns for each supplied capability a model with conjunctive preconditions and probabilistic effects. We supplied the predicate vocabulary induced by the representation function α used by MCQS, so that both methods operate over the same symbolic state space. We supplied the capabilities discovered by MCQS (Sec. 3) as the capability list. QACE therefore begins each run with the output of the discovery step, and the comparison in §E1 measures model-learning efficiency alone.

G.2 Compiling disjunctive conditions

MCQS represents a capability as a set of conditional-effect rules whose conditions are Boolean formulas over V (Def. 2), and the pessimistic and optimistic conditions constructed are disjunctive. QACE supports only conjunctive preconditions, so such capabilities are not expressible in its model class.

We therefore compile disjunctions away before passing capabilities to QACE. For each capability c and each rule $r \in h_c$, we rewrite $\mathrm{cond}(r)$ in disjunctive normal form as $\bigvee_{k=1}^{m} \varphi_k$, where each φ_k is a conjunction of literals, and construct m capabilities $c^{(1)}, \ldots, c^{(m)}$. Each $c^{(k)}$ retains the intent and effect distribution of c and has precondition φ_k, and each is passed to QACE as a separate capability.

This compilation affects the comparison in three ways. First, the number of capabilities QACE receives is $\sum_c \sum_{r \in h_c} m_{c,r}$ rather than $|C|$, and $m_{c,r}$ grows with the number of condition blocks in the effect partition Φ_c. Second, QACE learns each $c^{(k)}$ independently and does not represent the fact that these branches describe a single capability, so the sample cost of learning c is incurred m times. Third, the disjunctive normal form is derived from the model learned by MCQS, so the compilation supplies QACE with information about the structure of the conditions. We use it because without it QACE's hypothesis class does not contain the true model.

G.3 Query execution on symbolic and simulator-based agents

QACE poses queries at a specified symbolic state: a query fixes an initial symbolic state s and asks the agent to execute a capability, or a short sequence of capabilities, from s. Executing such a query requires an inverse of the representation function, that is, a state $x \in X$ with $\alpha(x) = s$ to which the simulator can be reset.

For symbolic agents this requirement is satisfied directly. For the LAO* agent on PDDLGym domains, α is the identity, every symbolic state named in a query is instantiable, and the simulator can be reset to it. All QACE runs reported in this paper use symbolic agents.

For BBAIs operating in non-symbolic simulators, including MiniGrid, Overcooked, Crafter, and SayCan, α is many-to-one and has no inverse. A symbolic state such as *agent-at*(NW) $\wedge$ $\neg$*carrying-blue-key*() corresponds to a set of grid configurations, and the simulator functionality, namely resetting to an initial state, reverting to a previously encountered state $x \in X$, stepping with an action, and querying available actions, does not include constructing or sampling a state from that set. MCQS avoids this requirement, since a query is $\langle x_0, \pi, n \rangle$ where $x_0 \in X$ is an environment state that has already been visited (Def. 5) and the policy is executed forward from x_0.

Executing a QACE query on such a simulator is possible in principle by searching for a reachable x with $\alpha(x) = s$. This requires solving a reachability problem in X for every query, with no guarantee that the abstract state is reachable, and the cost of this search is charged to the same simulator-step budget used for evaluation. We therefore report QACE

on symbolic agents only.

G.4 Scaling

The scaling of QACE on symbolic agents is limited by the grounded capability representation. Capabilities are grounded (Sec. 3), so MCQS constructs an intent for every type-consistent grounding of each predicate observed in an effect, and $|C|$ grows as $O(|O|^k)$ for a predicate of arity k. The compilation in Appendix G.2 increases this count further. QACE generates queries over grounded state-capability pairs, so a grounded capability set combined with a grounded predicate vocabulary causes the per-query search space to grow rapidly with $|O|$.

QACE learned an accurate model for LAO* on Blocksworld, where the object set is small enough that the grounded capability set remains small. For all other domains, including the remaining PDDLGym domains whose agents are symbolic and whose queries are therefore executable, QACE produced no model within 50 hours. Table 17 reports the configuration and outcome for each domain.

Domain	Agent	Outcome
Blocksworld	LAO*	Model learned
First Responders	LAO*	No model
Tireworld	LAO*	No model
Prob. Elevators	LAO*	No model
Depot	LAO*	No model
MiniGrid	ReAct (GPT-5.4-nano)	Not runnable
Overcooked	HDDLGym	Not runnable
Crafter	CSteve1	Not runnable
Crafter	Qwen CSteve	Not runnable
SayCan	SayCan	Not runnable

Table 17: configuration and outcome of QACE runs for each domain. $|C|$ is the number of capabilities discovered by MCQS and supplied to QACE, and $|C_{\mathrm{DNF}}|$ is the number after compiling disjunctive conditions into conjunctive capabilities (Appendix G.2). Not runnable indicates that α is not invertible for the agent's simulator (Appendix G.3).

H Limitations

A key result of this work is that accurately modeling BBAI capabilities requires an expressive grounded representation; lifting and aggressive generalization can produce inaccurate capability models. The consequence is that MCQS must learn grounded capabilities directly from observed transitions, making model quality dependent on transition coverage. In practice, this requires tracking all observed transitions between symbolic states and capabilities. Although this space is substantially smaller than the underlying environment state space, for symbolic states S and capabilities C, the resulting worst-case space and time complexity is $O(S^2C)$. While this level of expressivity is currently necessary for accurate capability learning, future work is needed to develop more efficient representations that preserve modeling fidelity while improving scalability.

A related limitation is that BBAIs and their environments are often highly stochastic. This substantially increases sample complexity because repeated policy executions are required both to observe rare effects and to accurately estimate their probabilities.

I Computational resources

Experiments were conducted on two hardware configurations: (1) an Intel Core i9-9900 CPU @ 3.10GHz with an NVIDIA GeForce RTX 2080 and 64GB RAM, and (2) an AMD Ryzen Threadripper PRO 7975WX (32 cores) with an NVIDIA RTX 6000 Ada Generation GPU and 64GB RAM.

MiniGrid, ReAct, Overcooked, and the PDDLGym domains were run on the i9 system using 15 parallel runs for MCQS-E, MCQS-S, and the Random Query baseline. Smaller PDDLGym domains, such as blocksworld and tireworld, required less than one hour each. First responders required approximately 3 hours, while probabilistic elevators and ReAct each required approximately 24 hours. Constructing the evaluation datasets required roughly one additional day. In total, this portion of the experiments required approximately 60 GPU-hours on hardware comparable to the i9 setup.

Both Crafter variants required approximately 24 hours each on the Threadripper system, with an additional 8 hours each for dataset construction. Running the SayCan experiments required approximately 72 hours per run, with only two runs executable in parallel, resulting in roughly 563 total hours. Overall, the Crafter and SayCan experiments required approximately 625 hours on the Threadripper system.

J Broader impacts

This work studies methods for learning interpretable capability models of black-box AI agents. The primary goal is improving transparency, reliability, and safety assessment by identifying the conditions under which agents succeed, fail, or exhibit unintended side effects. Such capability models may help users deploy AI systems more safely and help developers identify behavioral limitations.

We do not foresee direct harmful applications of this work. However, like many evaluation and interpretability techniques, these methods could potentially be used to characterize weaknesses or behavioral patterns of deployed agents. Overall, we believe the primary impact of this work is enabling more reliable understanding and evaluation of increasingly complex AI systems.