CASSANDRA: Programmatic and Probabilistic Learning and Inference for Stochastic World Modeling

Anonymous Author(s)
Submission Id: 914

ABSTRACT

Building world models from limited data is crucial for planning in real-world environments such as businesses. Such environments have semantically rich relationships between variables and stochastic dynamics arising from unobservable factors. Consequently, these environments require world knowledge to efficiently model complex action-effects and the causal relationships between variables. In this work, we propose CASSANDRA, a neurosymbolic approach to world modeling that leverages a Large Language Model's (LLM) general knowledge to create lightweight state-transition models for model-based planning in stochastic domains. We achieve this through a novel, two-part process: 1) using an LLM to synthesize executable code that models deterministic environment features, and 2) leveraging the LLM as a structural prior to discover a probabilistic graphical model for the stochastic features that captures causal relationships between variables. By integrating these two components, we create a single, factored world model where the symbolic code computes the deterministic action effects and the graphical model captures the stochastic effects that propagate through the variables. We apply our method to two environments: i) a small-scale coffee-shop simulator, in which we perform a detailed examination of our method's operation, and ii) a complex theme park business simulator, where we demonstrate significant improvements in planning and modeling transition dynamics over baselines.

KEYWORDS

World Model, Stochastic, Deterministic, LLM, Neurosymbolic

ACM Reference Format:
Anonymous Author(s). 2026. CASSANDRA: Programmatic and Probabilistic Learning and Inference for Stochastic World Modeling. In *Proc. of the 25th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2026), Paphos, Cyprus, May 25 – 29, 2026*, IFAAMAS, 22 pages.

1 INTRODUCTION

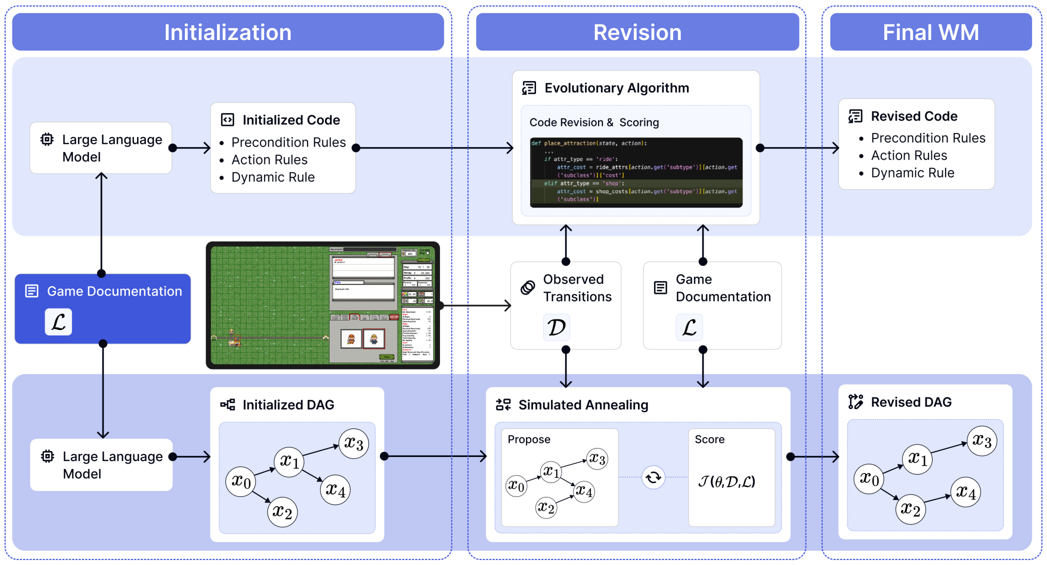
Effective planning over extended horizons requires agents to develop an internal model of their environment [20]. Recent advances in large language models (LLMs) have led to their application in world modeling from limited data, either by constructing code-based abstractions [26] or by directly predicting future textual states [31]. While promising in deterministic settings, these approaches often struggle with domain-specific uncertainty. Consider a restaurant introducing a larger pizza to its menu: the new menu and ingredients cost can be computed deterministically, but the change in customer demand and revenue are inherently stochastic. A competent world model must model both the deterministic and stochastic outcomes of such actions from limited data, a capability many current benchmarks overlook [5, 9, 12].

Existing approaches fall short in modeling these mixed deterministic and stochastic dynamics. Methods using pre-trained LLMs as zero-shot world models [11] both rely on general prior knowledge that fails to capture domain-specific data distributions, and require the LLM to perform symbolic computation such as arithmetic for the deterministic variables (a task for which the transformer architecture is not well-suited [2]). Code world models address the second issue by instead executing LLM-written code. Code revision works well for learning dynamics of deterministic environments, but lacks an efficient and principled mechanism for fitting models to real-world data, unlike well-established statistical methods.

To address these limitations, we introduce CASSANDRA, a dual-stream neurosymbolic framework that integrates a symbolic code component for deterministic variables with a probabilistic neural component for stochastic variables. As shown in Figure 1, our approach leverages a natural language description of the environment to initialize code for the deterministic model and a graphical structure for the stochastic model. Both of these components are then refined using observed trajectories. The programmatic component uses an evolutionary algorithm to propose, score, and select code refinements for transition prediction errors. Simultaneously, the graphical model's structure is refined via simulated annealing that uses observed trajectories and an LLM structural prior to surface semantically coherent causal relationships. The two components are tightly integrated, wherein the deterministic component models complex action effects and the stochastic component models their subsequent indirect stochastic consequences. We call the resulting world model CASSANDRA: Combining A Symbolic and Stochastic Architecture for Non-deterministic Domains.

We evaluate CASSANDRA against relevant baselines, including WALL-E [31] and WorldCoder [26], as well as multiple ablations of our framework. Our main contributions are: 1) A novel dual-stream, neurosymbolic framework, CASSANDRA, that decomposes world modeling of semantically rich tabular environments into separate deterministic (programmatic) and stochastic (probabilistic) components, initialized from natural language. 2) A methodology for refining this framework, where the programmatic model is improved via an evolutionary algorithm and the probabilistic model is learned using a guided search informed by an LLM structural prior and environment data. 3) Extensive evaluations showing that CASSANDRA models environment dynamics more accurately than baselines while maintaining fast inference time. As a result, CASSANDRA enables superior planning performance in environments such as the Mini Amusement Park (MAP) [1].

Proc. of the 25th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2026), C. Amato, L. Dennis, V. Mascardi, J. Thangarajah (eds.), May 25 – 29, 2026, Paphos, Cyprus. © 2026 International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org). This work is licenced under the Creative Commons Attribution 4.0 International (CC-BY 4.0) licence.



```
def place_attraction(state, action):
    ...
    if attr_type == 'ride':
        attr_cost = ride_attrs[action.get('subtype')][action.get
        ('subclass')]['cost']
    elif attr_type == 'shop':
        attr_cost = shop_costs[action.get('subtype')][action.get
        ('subclass')]
```

Figure 1: Overview of the CASSANDRA architecture. Deterministic dynamics are modeled by LLM-generated code, which is optimized by an evolutionary algorithm using observed trajectories (top). Stochastic dynamics are modeled by a probabilistic graphical model, designed through simulated annealing structure search with an LLM prior and observed transitions (bottom).

2 RELATED WORK

The planning and execution of actions in complex environments is a long-standing problem in machine learning. Given a perfect world model (i.e., simulator), model-based RL methods have achieved superhuman performance on tasks such as Go [23] and Chess [24]. When a world model (WM) is unavailable, algorithms such as MuZero [22] and DreamerV3 [10] can learn a neural model for use in planning. However, this process is data-intensive, often requiring millions of environment steps [10, 22].

To address this data burden, some works have aimed to leverage the knowledge and commonsense priors [30] captured by LLMs. One approach is to directly use an LLM as a world model to generate the next state. This can be used alongside traditional planning methods, e.g., using Monte Carlo Tree Search to balance exploration and exploitation [11]. However, these direct-prediction LLM world models struggle with domain-specific dynamics (particularly involving arithmetic) [27], and verifying action preconditions and effects [29]. WALL-E [31] partially addresses this by learning code rules for action preconditions, but is still reliant on the LLM for state prediction, lacking a mechanism for accurately learning distributions of stochastic variables. This is limiting as stochasticity remains a challenge, with untuned LLMs performing poorly at probabilistic forecasting [21], and fine-tuning remains a data-hungry process [28]. The high inference cost of LLM world models also makes repeated sampling for planning slow.

Instead of directly predicting states, another approach is code world models: using LLM-generated code to model transition dynamics. Such a world model is lightweight to run, and more sample efficient than finetuning an LLM. PDDL [8, 15] and Python [7, 26] are two common choices for the underlying language. For instance, WorldCoder [26] generates Python code that is refined using environment interactions. However, it cannot model stochastic environments, requires access to the ground-truth environment, and produces unstructured code that can only be refined as a monolithic block. In CASSANDRA's code-learning component, we learn offline from observations and structure the generated code, thus enabling targeted code revisions. We include an offline version of WorldCoder as a baseline, which learns from stored environment transitions. The Visual Predicator algorithm [14] overlaps with our work by generating functions that determine action effects and their preconditions. But unlike theirs, our work applies to complex text-based domains. AlphaEvolve [17] shares the general idea of evolving code using targeted refinement with our symbolic component. However, our method is also designed to handle stochasticity alongside the deterministic dynamics. While the Product of Programmatic Experts method [18] learns a stochastic code world model, it is limited to simple categorical distributions over code fragments and remains untested in stochastic environments. Probabilistic program induction has been attempted [6], but this approach cannot fit complex distributions to data and has only been tested in simple environments solvable with tabular learning. CASSANDRA integrates a code WM to compute deterministic action effects and a Bayesian Network to model stochastic environment changes, making it scalable to more complex domains such as MAP [1].

Fundamentally, LLMs benefit world modeling because they capture commonsense causal knowledge. Recent work on causal discovery with LLMs has demonstrated that large scale pre-training enables LLMs to reason about the causal structure of real-world systems [3, 13, 16] and act as common-sense priors [4, 19].However, to our knowledge, we are the first to explicitly exploit LLM's causal discovery capabilities in the context of world modeling. This enables robust modeling of stochastic environment dynamics by capturing causal variable relations from prior knowledge, and refining them using observed data.

3 METHODOLOGY

Our goal is to learn an expressive, hybrid world model of an environment from a collection of observed trajectories. At a high level, we do this by dividing the state into deterministic variables modeled by Python programs and stochastic variables modeled by a Bayesian Network and parameterized via neural networks. The code and network are then revised using LLMs and observed trajectories (see Figure 1).

3.1 Framework Overview

We model the environment as a Semantically-Rich Partially Observable Markov Decision Process (SR-POMDP) defined by the tuple $(\mathcal{O}, \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma, \mathcal{L})$, representing the observation space, state space, action space, transition function, reward function, discount factor, and semantic description, respectively. The semantic description $\mathcal{L}$ represents a corpus of text containing descriptions of environment dynamics and observation variables. We assume the observation space $\mathcal{O}$ is decomposed into two known components, such that an observation at time t is a composite of a deterministic observation vector $\mathbf{s}_t \in \mathcal{O}_{\text{det}}$ and a stochastic observation vector $\mathbf{x}_t \in \mathcal{O}_{\text{stoch}}$. We also observe rewards $r_t \in \mathbb{R}$. Our goal is to model the ground truth transition probability $\mathcal{P}$ and reward function $\mathcal{R}$ in terms of the observables $(\mathbf{s}_t, \mathbf{x}_t)$ and action $a_t \in \mathcal{A}$ from a dataset of observed transitions $\mathcal{D} = \{(\mathbf{s}_{t-1}, \mathbf{x}_{t-1}, a_{t-1}, \sigma_{t-1}, \mathbf{s}_t, \mathbf{x}_t)_i\}_{i=1}^{N}$, where σ_{t-1} denotes whether the action could be executed at the corresponding state. We assume σ_{t-1} can be computed deterministically and treat it as a deterministic observation. To simplify our notation, we include the reward r_t in the stochastic state vector x_t as another observable.

By separating deterministic from stochastic variables, we can factorize the transition probability to separate deterministic and stochastic dynamics given an action. This distinction is natural in many environments, such as business simulators or games. For example, in a business simulator, an agent may choose to invest in advertising. The world model needs to simulate the resulting deterministic changes (e.g., reduce funds by advertisement cost) and stochastic changes (e.g., customer interest $\rightarrow$ sales $\rightarrow$ revenue). Similarly, in a game of Monopoly, a player may choose to buy a property. Paying for and acquiring the property is a deterministic process, while the revenue collected from rent paid by other players is stochastic. Both these components need to be modeled to accurately represent the transition dynamics

We decompose the transition probability into a deterministic and stochastic component as follows:

$$p(\mathbf{x}_t|\mathbf{s}_{t-1}, \mathbf{x}_{t-1}, a_{t-1}) = p(\mathbf{x}_t|\mathbf{x}_{t-1}, a_{t-1}, \mathbf{s}_t, \mathbf{s}_{t-1})p(\mathbf{s}_t|\mathbf{s}_{t-1}, \mathbf{x}_{t-1}, a_{t-1}). \quad (1)$$

This separation enables the use of specialized algorithms that best model each component. The primary goal of our work is to learn the parameterized approximations $p_\theta(\mathbf{x}_t|\mathbf{x}_{t-1}, a_{t-1}, \mathbf{s}_t, \mathbf{s}_{t-1})$, and $p_\phi(\mathbf{s}_t|\mathbf{s}_{t-1}, \mathbf{x}_{t-1}, a_{t-1})$ where the latter is defined as the indicator function, $\mathbb{1}_{\{f_\phi(\mathbf{s}_{t-1}, \mathbf{x}_{t-1}, a_{t-1})\}}(\mathbf{s}_t)$ using a deterministic function f_ϕ. Both θ and ϕ are learned using the environment prior knowledge in $\mathcal{L}$ and the observed transitions $\mathcal{D}$.

Due to the decomposition in equation (1), we can treat these as two separate optimization problems:

$$\max_{\phi} \sum_{(\mathbf{s}_{t-1}, \mathbf{x}_{t-1}, a_{t-1}, \sigma_{t-1}, \mathbf{s}_t, \mathbf{x}_t) \in \mathcal{D}} \log p_\phi(\mathbf{s}_t|\mathbf{s}_{t-1}, \mathbf{x}_{t-1}, a_{t-1}), \quad (2)$$

$$\max_{\theta} \sum_{(\mathbf{s}_{t-1}, \mathbf{x}_{t-1}, a_{t-1}, \sigma_{t-1}, \mathbf{s}_t, \mathbf{x}_t) \in \mathcal{D}} \log p_\theta(\mathbf{x}_t|\mathbf{x}_{t-1}, a_{t-1}, \mathbf{s}_t, \mathbf{s}_{t-1}). \quad (3)$$

In the following subsection, we describe our strategy for each of the two optimization problems. Given that we operate in a semantically rich context, in both problems we leverage LLMs to facilitate learning using representations appropriate for each modality: f_ϕ is parameterized by code, which is iteratively refined in accordance with both prior environment knowledge $\mathcal{L}$ expressed in natural language and observed data. In turn, the distribution $p_\theta(\mathbf{x}_t|\mathbf{x}_{t-1}, a_{t-1}, \mathbf{s}_t, \mathbf{s}_{t-1})$ is represented by a Bayesian network parameterized by a collection of smaller neural models. The structure of the Bayesian network is informed by $\mathcal{L}$ and refined using observed data using a novel LLM-as-prior technique that focuses structure search on semantically coherent causal model structures.

3.2 Modeling Deterministic Dynamics using LLM-Generated Code

We model direct action effects and other environment-driven deterministic changes in the state using code. The model aims to generate code ϕ that maximizes $p_\phi(\mathbf{s}_t|\mathbf{s}_{t-1}, \mathbf{x}_{t-1}, a_{t-1})$ over the data. We approximate this objective by iteratively improving ϕ using an evolutionary algorithm. The code ϕ can be executed as a function denoted as f_ϕ to produce the following: 1) the deterministic observation vector, s_t, and 2) action validity boolean, $\sigma_{t-1} \in \{True, False\}$ that indicates whether the action a_{t-1} is valid for the given observation. This optimization process has two parts: **1)** Generating an initial code draft using the game description, $\mathcal{L}$, and **2)** Refining the code using $\mathcal{L}$, along with the dataset of trajectories, $\mathcal{D}$.

3.2.1 Code Components: The deterministic observation vector s_t can be derived from (s_{t-1}, x_{t-1}) by accounting for changes from two sources; a) Direct effects of the action a_{t-1} on (s_{t-1}, x_{t-1}) and b) Environment-driven deterministic changes that transform s_{t-1} independent of the action performed (e.g. advancing time-step, paying out fixed salaries to employees, etc.). Hence, we introduce **action functions** which are specific to each action, and a separate **dynamic function** that models other deterministic dynamics. Since LLMs used as world models are better at generating the difference in the state than the complete next state [27], we design action and dynamic functions to produce changes that combine to transform s_{t-1} into s_t. Let $\mathbb{OD}$ represent the space of possible observation differences for the domain. The change produced by the action function $c_t^a \in \mathbb{OD}$ and the change produced by the dynamic function $c_t^d \in \mathbb{OD}$ can be applied to the previous deterministic observation to produce the next one, i.e. $(s_t, x_{t-1}) = (s_{t-1}, x_{t-1}) \oplus c_t^a \oplus c_t^d$ (where $\oplus$ denotes the operation of applying a state change).

To model action validity σ_t, we need action-specific checks, all of which need to pass before an action can be executed. We introduce a third type of function called **precondition function** for this purpose. The three types of functions collectively become the code ϕ of the model:

- **Precondition function (pc)**: ($f_{pc_{a_t}}(s_t, x_t, a_t) \rightarrow \{True, False\}$) captures a precondition (of action a_t) that needs to be True for an action a_t to be valid for (s_t, x_t).
- **Action function (α)**: ($f_{\alpha_{a_t}}(s_t, x_t, a_t) \rightarrow \mathbb{OD}$) generates $c_t^{a_t}$, the change to the input observation vector (s_t, x_t) caused by a valid action a_t that is applied to it.
- **Dynamic function (d)**: ($f_d(s_t, x_t) \rightarrow \mathbb{OD}$) generates c_t^d, the change to the input observation vector (s_t, x_t) caused due to environment-driven factors.

Breaking down the code into multiple purpose-specific functions allows the LLM to focus on one task at a time and also enables targeted refinement using a smaller context size.

The function ϕ takes as input (s_{t-1}, x_{t-1}) and a_{t-1} to first compute σ_{t-1} as follows:

$$\sigma_{t-1} = \bigwedge_{i=1}^{n} f^i_{pc_{a_{t-1}}}(s_{t-1}, x_{t-1}, a_{t-1}). \tag{4}$$

Where $\bigwedge$ is the logical AND operation and f^i_{pc} for $i \in 1 \cdots n$ are the precondition functions of action a_{t-1}. i.e., *if any precondition fails, the action is determined to be invalid.* If σ_{t-1} is False, the observation is carried over, $s_t = s_{t-1}$. If σ_{t-1} is True, the deterministic observation vector is computed as:

$$s_t = s_{t-1} \oplus f_{\alpha_{a_{t-1}}}(s_{t-1}, x_{t-1}, a_{t-1}) \oplus f_d(s_{t-1}, x_{t-1}). \tag{5}$$

3.2.2 Code Optimization: We generate the three types of functions that comprise ϕ using LLMs (prompt templates are given in the Appendix F.1). This code is optimized in two steps: 1) Initializing ϕ using $\mathcal{L}$ and 2) Refining ϕ using $\mathcal{L}$ and $\mathcal{D}$. These two steps are explained below:

Initializing code: We use a prompt template T_{init} and fill in game documentation information $\mathcal{L}$ to create the prompt P_{init}. This prompt is fed to the LLM to generate an initial version of code ϕ^0:

$$\phi^0 \leftarrow LLM(T_{init}(\mathcal{L})). \tag{6}$$

Refine code: The initial code may be far from perfect, especially in complex domains. Revising the code using trajectory data ($\mathcal{D}$) helps correct errors in the code and fill in the missing gaps [7, 17, 26]. We follow an evolutionary algorithm guided by a heuristic scoring function to iteratively improve ϕ.

During refinement, the dataset is processed one transition at a time. This revision loop works as follows. Consider an arbitrary transition from observation $t-1$ to t. The transition would then look like $(s_{t-1}, x_{t-1}, a_{t-1}, \sigma_{t-1}, s_t, x_t)$. Given the current code version, ϕ', we can compute the predicted deterministic observation and action validity for the transition.

$$s'_t, \sigma'_{t-1} = f_{\phi'}(s_{t-1}, x_{t-1}, a_{t-1}) \tag{7}$$

Comparing (s'_t, σ'_{t-1}) with (s_t, σ_{t-1}), we identify four types of possible prediction errors, $E = \{E_{exec}, E_{pf}, E_{ps}, E_{od}\}$ which are described below:

- E_{exec}: Execution/syntax error.
- E_{pf}: Precondition too restrictive, i.e. $\sigma'_{t-1} = F$, but $\sigma_{t-1} = T$.
- E_{ps}: Precondition too permissive, $\sigma'_{t-1} = T$, but $\sigma_{t-1} = F$.
- E_{od}: Observation prediction error. $\sigma'_{t-1} = \sigma_{t-1}$, but $s'_t \neq s_t$.

Algorithm 1 Code Refinement using Observation Transitions

```
procedure RefineCode($\mathcal{D}$, $\phi'$)
    for each transition $(s_{t-1}, x_{t-1}, a_{t-1}, \sigma_{t-1}, s_t, x_t) \in \mathcal{D}$ do
        $s'_t, \sigma'_{t-1} \leftarrow f_{\phi'}(s_{t-1}, x_{t-1}, a_{t-1})$    ▷ Current code predictions
        if $s'_t = s_t \wedge \sigma'_{t-1} = \sigma_{t-1}$ then
            pass
        else
            $\epsilon \leftarrow ErrorType(s'_t, \sigma'_{t-1}, s_t, \sigma_{t-1}), \epsilon \in E$
            $P_{refine} \leftarrow T_{refine}(s'_t, \sigma'_{t-1}, s_t, \sigma_{t-1}, \epsilon, \mathcal{L})$
            $r_t^1, \ldots, r_t^K \leftarrow LLM(P_{refine})$    ▷ Generating refinements
            $r_t^\star \leftarrow \arg\max_{RS(s'_t, \sigma'_{t-1}, s_t, \sigma_{t-1}, \epsilon, r_t^j)} r_t^j : VS(r_t^j, \phi', \epsilon, V) > 0$
            $\phi'' \leftarrow ApplyRefinement(r_t^\star, \phi')$
            $\phi' \leftarrow \phi''$    ▷ Updating code with best refinement
        end if
    end for
    return $\phi'$
end procedure
```

Given the type of error ϵ, a prompt P_{refine} is constructed using template T_{refine} to generate K possible refinements, $r_t^1, \cdots r_t^K$. Each refinement is an operation that adds, removes, or replaces a single function in ϕ'. Given the candidate refinements, we compute a heuristic scoring function (called $RefinementScore$ (RS)) that computes the decrease in prediction error of each refinement on a given transition, compared to the original set of functions. The refinement with the highest score for the current transition is selected to update the code.

A refinement based on an incorrect idea may work well for the current transition, but will not generally apply to other transitions in the dataset. Accepting such a refinement does not improve ϕ' and may steer it in the wrong direction. We mitigate this by maintaining a randomly selected set of transitions, V, which are used to validate each refinement using $ValidationScore$ (VS). This mechanism is fundamental to our evolutionary algorithm as it strongly incentivizes code improvement with every iteration. The code refinement process is explained in more detail in Algorithm 1. Further details of the refinement scoring function are given in Appendix D.1.

3.3 Modeling Stochastic Dynamics with LLM priors

The core of our world model's ability to handle uncertainty lies in learning the conditional distribution $p(\mathbf{x}_t | \mathbf{x}_{t-1}, a_t, \mathbf{s}_t, \mathbf{s}_{t-1})$, where the reward r_t is included in $\mathbf{x}_t$ for conciseness. A key challenge is that individual variables within $\mathbf{x}_t$ are unlikely to be conditionally independent given the past. For example, in a business simulator, the number of customers and the daily profit are distinct stochastic variables, but they are clearly dependent on each other. Capturing causal variable dependencies is especially important for planning, as failing to do so can lead to unrealistic states during rollouts, which can negatively impact performance. We aim to leverage LLMs' knowledge of causal reasoning [16] to aid in structuring the dependencies in $\mathbf{x}_t$.

To model these complex dependencies, we utilize Bayesian Networks represented by a Directed Acyclic Graphs (DAG). This provides a principled trade-off, capturing conditional dependencies between variables while remaining computationally tractable for both training and inference and is a natural choice for modeling causal relations. Inference efficiency and causal modeling are critical in the planning context, where the world model must be queried repeatedly to simulate future trajectories that are consistent with the true environment dynamics. While more flexible, unstructured models like diffusion or energy-based models could be used, they are typically too data-hungry and computationally intensive for this setting. Therefore, in this section, we describe our method for searching for effective DAG structures using observed trajectories and LLMs as common-sense priors, focusing structure search in semantically coherent model structures with plausible causal relationships.

Our goal is to construct a graph $G = (V, E)$ where $V = V_x \cup V_s$ is a set of nodes such that $v \in V_x$ (corresponding to entries in $\mathbf{x}_t$), $v \in V_s$ (akin for $\mathbf{s}_t$), and edges $(v_i, v_j) \in E$ form a DAG, encoding variable dependencies in $\mathbf{x}_t$. Also, for a given node $v \in V$, $\mathcal{L}[v]$ corresponds to the name and description of the variable in natural language. Given a DAG $G = (V, E)$ and a topological ordering of the variables O, the probability in equation (3) factors as follows:

$$p_\theta(\mathbf{x}_t \mid \mathbf{x}_{t-1}, a_{t-1}, \mathbf{s}_t, \mathbf{s}_{t-1}) = \prod_{v \in O} p_{\theta_v}(\mathbf{x}_{t,v} \mid \mathbf{x}_{t,\mathrm{pa}(v)}, \mathbf{x}_{t-1}, a_{t-1}, \mathbf{s}_t, \mathbf{s}_{t-1}), \quad (8)$$

where $pa(v)$ indexes the parents of v in the DAG. Each model p_{θ_v} is represented by a parameterized distribution (e.g., a neural network) and can all be fit in parallel using observed transitions. In the rest of this section, we discuss how to leverage the environment description $\mathcal{L}$ to learn an effective structure E that accurately models variable dependencies.

3.3.1 *DAG structure search*.

We search for causal DAG structures for $p_\theta(\mathbf{x}_t|\mathbf{x}_{t-1}, a_t, \mathbf{s}_t, \mathbf{s}_{t-1})$ by a simulated annealing procedure that searches for structures that fit the data while being consistent with the dynamics described in $\mathcal{L}$. First, we sample initial DAGs using LLMs to inform the structure based on the environment description. This ensures we start from graphs that encode meaningful relationships between variables, such as the number of customers influencing the total revenue in a business. Then, we conduct a search guided by empirical data and $\mathcal{L}$ to refine it.

Sampling semantically coherent DAGs. To initialize our search, we sample DAGs for the variables in $\mathbf{x}_t$ using a two-stage algorithm. First, we prompt an LLM to create a topological ordering of V using $\mathcal{L}$. Then, we iteratively prompt to elicit causal variable dependencies consistent with the topological order.

Algorithm 2 shows the sampling procedure for DAGs encoding semantically coherent relationships between variables. First, a single pass over the variables is used to conduct a topological sort over the variables using the environment semantics. Given a partial topological sort, an LLM is instructed to select a variable that can be plausibly modeled using the variables sorted so far. In the second step, another single pass over the variables uses an LLM to elicit variable dependencies, respecting the topological order. This ensures that the final edge set E forms a DAG. While more complex procedures could be employed (e.g., by pairwise comparisons), we find that this approach works well in practice, especially when $\mathcal{L}$ is rich, and assuming it does not exceed context limits. Still, these initial structures need not be perfect since they will be refined during structure search. Overall, this procedure requires $O(d)$ LLM calls, where d is the dimension of x_t.

Algorithm 2 Sampling Semantically Coherent DAGs via LLM

```
1: procedure SemanticTopoSort($V_x$, $V_s$, $\mathcal{L}$)
2:     $O \leftarrow (v \text{ for } v \in V_s)$          ▹ Initialize with exogenous variables
3:     $V_{rem} \leftarrow V_x$
4:     while $V_{rem} \neq \emptyset$ do                  ▹ While not all sorted
5:         $P_1 \leftarrow$ ConstructPrompt($O, V_{rem}, \mathcal{L}$)
6:         $v^* \leftarrow$ LLM($P_1$)
7:         $O \leftarrow O \oplus v^*$                    ▹ Append $v^*$ to topological order
8:         $V_{rem} \leftarrow V_{rem} \setminus \{v^*\}$
9:     end while
10:    return $O$
11: end procedure
12: procedure ElicitDependencies($O, V_x, V, \mathcal{L}$)
13:    $E \leftarrow \emptyset$
14:    for $i \leftarrow 1$ to $|O|$ do
15:        $v_i \leftarrow O[i]$
16:        if $v_i \in V_x$ then                          ▹ Skip vars in $V_s$
17:            $C(v_i) \leftarrow \{v_j \in V \mid v_j \text{ precedes } v_i \text{ in } O\}$
18:            $P_2 \leftarrow$ ConstructPrompt($\mathcal{L}[v_i], C(v_i), \mathcal{L}$)
19:            $S \leftarrow$ LLM($P_2$)
20:            $E \leftarrow E \cup \{(v_p, v_i) \mid v_p \in S\}$   ▹ Directed edge
21:        end if
22:    end for
23:    return $E$
24: end procedure
25: Input: Set of nodes $V = V_x \cup V_s$, environment semantics $\mathcal{L}$
26: Output: A directed acyclic graph $G = (V, E)$
27: $O \leftarrow$ SemanticTopoSort($V_x, V_s, \mathcal{L}$)
28: $E \leftarrow$ ElicitDependencies($O, V_x, V, \mathcal{L}$)
29: return $G = (V, E)$
```

Searching for DAGs with data and LLM prior. While DAGs offer a flexible and expressive structure for modeling state variables, searching the super-exponential space of DAGs is computationally hard. We therefore concentrate the search to models that fit data well while also being semantically plausible given the natural language information in $\mathcal{L}$. First, we construct a prior $p(E|V, \mathcal{L})$ over the edge set E using an LLM, by posing the problem as a question-answering task. We construct a token sequence $w(\mathcal{L}, V, E)$ that, given $\mathcal{L}$ and V asks whether the relationships encoded in E are causally plausible, given the environment prior knowledge. We then use the probability of an affirmative answer as the prior probability of that structure in our search:

$$p(E|V, \mathcal{L}) = p_{LLM}(\text{"yes"}|w(E, V, \mathcal{L})). \quad (9)$$

In practice we use structured decoding and token log probabilities with a fixed temperature parameter to compute equation (9). While more complex approximations of the prior using multiple LLM calls are possible, we use this single-prompt approach to improve efficiency, as this term is evaluated on every step of the search. Additional details for the prior computation and prompts are available in Appendices E and F.2 respectively. Next, using the dataset $\mathcal{D}$ of trajectories, we can perform structure learning by the following optimization problem:

$$E_{MAP} = \arg\max_{E} p(D|E)p(E|V, \mathcal{L}), \tag{10}$$

where E_{MAP} is the maximum aposteriori estimate of the graph structure with our prior. In practice, we pose this optimization problem as a search and use the following objective function:

$$J(E; D) = \frac{1}{|\mathcal{D}|} \log p_\theta(D|E) - \lambda_1 \frac{|E|}{|V|^2} + \lambda_2 \log p(E|V, \mathcal{L}). \tag{11}$$

The first two terms of equation (11) are the model score, which encourages fitting the data with simpler models, similar to the Bayesian Information Criterion, as the number of edges is directly related to the number of parameters. The third term is the LLM prior. The constants λ_1, λ_2 are hyperparameters that balance the contributions of different terms. In turn, λ_1 can be used to control sparsity, which can increase robustness at the cost of model flexibility. In practice, we find that with common LLM temperature settings (e.g. $\tau = 1$), the search is robust to the value of λ_2, as LLMs tend to reject or accept candidate E with high probability. Then, the LLM acts as a constraint, effectively limiting the search space towards graphs consistent with the dynamics described in $\mathcal{L}$ by heavily penalizing structures that contradict it. In our experiments, we set λ_2 to a large value, effectively turning it into a hard constraint in most cases. We tune λ_1 to balance the first two terms in magnitude.

To search for structures using the objective in equation (11) we use simulated annealing with random perturbations that preserve acyclicity. We start the search with initial samples obtained using algorithm 2, and search in parallel. This approach automatically considers the LLM's uncertainty over the model structures: If the model is confident about a particular substructure, then it is more likely to appear in multiple samples, and as a result, more of the search effort is concentrated on graphs that include it. Additional details on our search procedure are available in Appendix E.

Fitting to data. Given a structure E, we fit the conditional distributions prescribed by the graph structure using neural networks for the approximation. It is important to fit the distribution of each variable, rather than attempt to predict the mean, as sampling is needed for planning. Therefore, in our experiments, we use quantile regression for continuous variables and train with the pinball loss to flexibly model the distribution of values without further assumptions. However, this can be modified if additional domain knowledge is available. We use cross-entropy loss for categorical variables. Additional implementation details are available in Appendix E.

4 EXPERIMENTS

In this section, we present experiments that evaluate world models on two business simulator environments: CoffeeShopSimulator[1] and Mini Amusement Park (MAP)[1]. First, we examine in detail the structure learning component of CASSANDRA. Then we compare the performance of agents using world models for planning. We also include evaluations on the prediction accuracy of world models for state variables and action validity.

We compare CASSANDRA against comparable baselines using two metrics: 1) transition prediction accuracy, and 2) task performance, when integrated with a planning agent. When evaluating the prediction accuracy, we consider the deterministic and stochastic components independently. For the deterministic component, we compare against an offline implementation of WorldCoder [26]. For the stochastic component, we include multiple ablations of CASSANDRA: (i) CASS-Ind predicts stochastic state variables independently (ii) CASS-L replaces the learned DAG with a randomly-ordered linear DAG, and (iii) CASS-R ablates our structure learning by optimizing a randomly initialized DAG without the LLM prior. Finally, we compare the full model against WALL-E-MAP, a version of WALL-E [31], adapted to the MAP environment and enhanced with a retrieval system that retrieves few-shot examples of similar state transitions. We use GPT-4.1-mini for all experiments. Additional implementation details are available in Appendix A.

Agent	Final Budget
MPC+CASS-L	987.5 ± 697.5
MPC+CASS-Ind	4853.3 ± 3689.0
MPC+CASS-R	1669.55 ± 1475
MPC+CASSANDRA	**9951.0 ± 820.0**

Table 1: Budget at the end of 50 days in CoffeeShopSim with different stochastic models. MPC+CASS-L and MPC+CASS-R perform the worst, likely because they rely on spurious correlations. MPC+CASS-Ind performs better on average, but exhibits high variance due to some runs going bankrupt. The CASSANDRA agent outperforms the baselines as it uses robust variable relations to make predictions for planning.

4.1 Experiment 1: CoffeeShopSimulator

In this experiment, we use the CoffeeShopSimulator to illustrate how our structure learning algorithm models stochastic dynamics and how the learned causal structure impacts planning. The simulator challenges an agent to manage inventory, set prices, and maximize profit over a 50-day horizon. Stochasticity arises primarily from customer behavior and sales, which depend on the store's daily state. This environment provides a controlled setting to evaluate our method's ability to discover a DAG to model these dynamics for effective planning.

Experiment Settings. We first generate a dataset of 100 trajectories using a simple heuristic agent, with 90 used for training. The simulator state contains 8 variables, 6 of which are dependent and modeled by CASSANDRA. We run 50 search steps from 5 initial samples to find the best-scoring DAG structure. To focus on the stochastic component, all planning evaluations use the ground truth simulator for deterministic effects.

Results. Figure 2 shows the initial graph structure sampled from the environment description versus the final, optimized structure. While the initial structure captures key high-level relations (e.g., price and quality attract customers), it fails to connect all variables

[1]Source code will be made available upon publication.

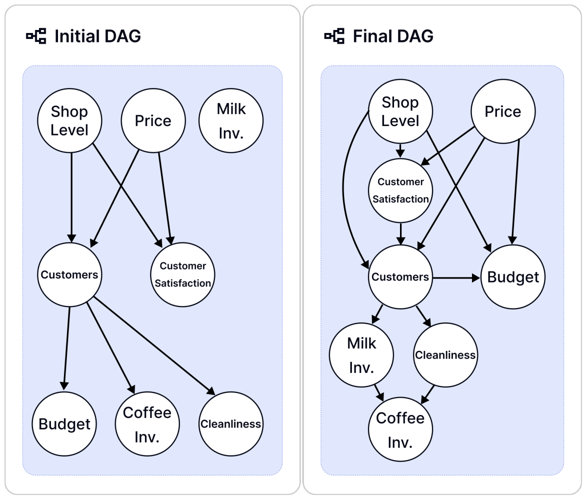


Figure 2: Initial and final DAGs for CoffeeShopSim, modeling dependencies of state variables. The initial DAG captures the basic relationships described in the environment description. The final one is refined, capturing key relations, while still having some noisy edges.

(e.g., Milk inventory is disconnected). After optimization, the search procedure uncovers a latent relation not explicitly mentioned in the description: customer satisfaction directly boosts the number of customers on the same day. While not perfect—the model adds a spurious edge between coffee and milk inventory—the final structure fits the data well and is consistent with $\mathcal{L}$.

Table 1 shows the performance of Model Predictive Control (MPC) agents using different stochastic models. Agents using ablated models with a linear or random DAG (CASS-L and CASS-R) perform poorly, likely by fitting brittle correlations that hinder planning. The agent predicting variables independently (CASS-Ind) fares better as predictions are likely more robust, but exhibits high variance due to occasional bankruptcy. This is because the independence assumption makes the model miss the crucial effect of customer satisfaction on guest count and cleanliness, leading to poor planning. An interesting finding is that all methods achieve similar next-state prediction errors (see Appendix B). This highlights that simply minimizing average errors is insufficient; a well-structured causal model, like that learned by CASSANDRA, is vital for robust planning even when raw prediction metrics are comparable.

4.2 Experiment 2: Mini Amusement Park (MAP)

We further evaluate our method on the Mini Amusement Park (MAP) simulator [1], a challenging testbed for world models. In this environment, an agent starts with a limited budget and must manage the park over a long time horizon by building rides, operating shops, and hiring staff. The primary challenge lies in modeling guest behavior, which is inherently stochastic and dependent on a complex interaction of hidden variables.

4.2.1 Experimental Setting. We collect 84 high-quality training trajectories from 7 distinct park layouts using an MCTS agent with ground-truth simulator access. All evaluations are performed on 3 held-out layouts. We construct planning agents by running MCTS using the CASSANDRA model and ablations. Similar to the original WALL-E paper, we embed WALL-E-MAP in a planning agent using MPC for planning with LLM-based action selection. We also include an MCTS version that uses the WALL-E-MAP. All agents use a random action proposal with simple heuristics. We use a similar search time budget for all agents ($\approx$ 1 hour per action) for comparison. We evaluate using 36 runs of each agent. Additional details are available in Appendix C.2.

The state observation of MAP consists of a complex JSON object of park information, including park statistics, ride, shop, and staff attributes. The state observation is a complex JSON object with 81 unique fields, creating an incredibly large modeling space. The deterministic component uses the entire JSON but only modifies deterministic variables. While many low-level variables in the JSON are stochastic, for planning purposes, it is not necessary to model all of them (e.g., the cleanliness of any one tile). Instead, we focus our stochastic model on 6 high-impact dependent variables related to planning and use the rest as independent variables. Therefore, the DAG is constructed over all variables (using summary statistics for repeated fields) but only 6 variables are considered as dependent. Given the super-exponential space of DAGs, the resulting search space is intractably large without strong heuristics.

4.2.2 Results - Survival Rates. Table 2 shows the performance of agents in MAP in terms of the survival rate over 50 days of park operation. The WALL-E-MAP agent eventually goes bankrupt in almost all runs, as it is unable to capture the stochastic dynamics sufficiently well. This is compounded by the computational cost of calling an LLM for every transition, limiting its search capability. CASS-Ind performs significantly better as the symbolic component accurately captures the deterministic dynamics, and the independent predictions can capture some of the stochastic dynamics. However, eventually, most runs go bankrupt. This is because without accurate modeling of the joint distribution of state variables, the agent struggles to navigate the tight margins of the simulation (e.g., hiring janitors at the right time to keep the park clean, making the park appealing, and thus increasing revenue). CASS-L performs a bit better as it is able to capture some variable dependencies. The CASSANDRA agent, which uses a more expressive model of the state distribution with causal dependencies, better models the environment dynamics and enables the agents to stay in business all 50 days.

4.2.3 Results - Modeling Transition Dynamics. To compare the world models based on their transition dynamics, we evaluate the observation vector (s_t, x_t) and the action validity (σ) predicted by each world model against the true values from 1800 observed transitions drawn from 36 test trajectories.

To measure observation prediction errors, we use metrics depending on the variable type and error. A prediction can be *invalid* when a variable has a value that is out of expected bounds. We measure an invalid value rate for each variable across all transitions in the test set. We report the average invalid value rate of a set of variables, Inv, by averaging the invalid value rates of variables in the set. An observation can have different types of variables (categorical and numerical). Hence, we include average accuracy measurements (Cat_{acc}) for categorical variables and average Root Mean Squared

Method	10 Days	20 Days	30 Days	40 Days	50 Days
MPC+WALL-E-MAP	26% ± 13%	20% ± 12%	17% ± 11%	13% ± 10%	8% ± 7%
MCTS+WALL-E-MAP	66% ± 5%	52% ± 15%	40% ± 15%	29% ± 14%	27% ± 14%
MCTS+CASS-Ind	90% ± 9%	54% ± 15%	29% ± 14%	29% ± 14%	29% ± 14%
MCTS+CASS-L	54% ± 15%	54% ± 15%	47% ± 16%	47% ± 16%	47% ± 16%
MCTS+CASSANDRA	95% ± 5%	**95%** ± 5%	**95%** ± 5%	**95%** ± 5%	**95%** ± 5%

Table 2: Comparison of 95% confidence interval of survival rates in MAP over 50 days. Higher values are better. The MPC+WALL-E-MAP agent goes bankrupt on almost every run by the 50 day mark, with most runs not making it past the first few days, but the MCTS variant performs better. The CASS-Ind and CASS-L agents performs better still. In our experiments, the CASSANDRA agent does not go bankrupt in any of the runs, significantly outperforming the baselines.

Model	$Cat^D_{acc}\uparrow$	$Num^D_{rmse}\downarrow$	$Inv^D\downarrow$	$Num^S_{rmse}\downarrow$	$Inv^S\downarrow$	$\sigma_{acc}\uparrow$	$\sigma_{prec}\uparrow$	$\sigma_{rec}\uparrow$	$\sigma_{F1}\uparrow$	$Time(s)\downarrow$
OfflineWorldCoder [26]	1.000	0.483	0.001	-	-	0.905	0.844	0.995	0.913	3.40
WALL-E-MAP [31]	0.944	0.084	0.042	0.515	0.157	0.878	0.804	1.000	0.891	222.30
CASS-Ind	1.000	0.176	0.012	0.254	0.122	0.945	0.902	0.998	0.948	8.44
CASS-L	1.000	0.197	0.017	0.249	0.119	0.945	0.902	0.998	0.948	9.59
CASSANDRA	1.000	0.176	0.012	0.248	0.117	0.945	0.902	0.998	0.948	10.67

Table 3: Comparison of world models' transition predictions. Cat_{acc} denotes prediction accuracy for categorical variables. Num_{rmse} denotes the average RMSE of numerical variables. Inv is the average invalid value rate. σ_{acc}, σ_{prec}, σ_{rec} and σ_{F1} are the accuracy, precision, recall and F1 scores of action validity prediction. $Time$ is the average prediction time for one transition. Superscript 'D' or 'S' indicates deterministic or stochastic component of the observation, respectively. $\uparrow$ indicates higher number is better and $\downarrow$ indicates lower number is better. (Since OfflineWorldCoder is purely deterministic, we omit stochastic results and enter '-' instead)

Error (RMSE) values (Num_{rmse}) for numerical variables (which are first normalized using Min-Max Scaling). Table 3 reports the results for both observation and action validity predictions.

The results show that CASSANDRA can model both deterministic and stochastic observation variables effectively in a fraction of the time required by WALL-E-MAP. Also, its action validity prediction is significantly better than all the baselines. Similar to experiment 1, CASSANDRA variants have similar prediction accuracies in stochastic variables, indicating that achieving low average errors does not guarantee accurate modeling for planning, as indicated by the large task performance differences in Table 2. We attribute this improvement to our model's targeted code refinement process that enables learning complex preconditions for each action.

5 CONCLUSION

In this work, we introduced CASSANDRA, a novel neuro-symbolic framework for building world models in semantically rich stochastic environments. Our primary contribution is a hybrid approach that leverages LLMs to use prior knowledge expressed in text for both programmatic and probabilistic modeling. Evaluations of our deterministic component show that targeted code refinement helps CASSANDRA model deterministic dynamics such as action validity more accurately than baseline methods. In addition, our experiments show that our structure learning with LLM priors is able to learn causal structures that improve planning capability in both a simple coffee shop environment and a complex theme park business simulation.

Despite these promising results, our approach has some limitations. First, CASSANDRA assumes a known decomposition of stochastic and deterministic state observations. While this decomposition may not apply to all domains, it applies to many realistic environments as discussed in Section 3.1. Furthermore, in practice, if the status of an observed variable is unknown, then it can be safely assumed to be stochastic and learned by statistical methods. Second, although our model doesn't depend on LLM calls during the planning phase, it relies on LLMs for code generation, code refinement, and structural priors. This learning process can require significant computational resources and time. The quality of the initial model is highly dependent on the LLM's pre-existing knowledge and environment description provided, which may not always align perfectly with the true dynamics. Also, while our method effectively refines the initial models using observed data, its performance is ultimately tied to the quality of those observed trajectories.

Our work points towards multiple avenues for future work. First, future work can forego the assumption that a deterministic-stochastic decomposition is known a priori, and can automatically classify variables based on priors and observed data. Second, improving the sample efficiency of world models through active learning can further improve learning efficiency and performance in domains where data is scarce and expensive to acquire. Third, online learning and refinement of world models can also enable continual deployment in evolving domains. This is especially important for applications such as businesses, where dynamics can shift over time. Finally, future work can refine learned causal relationships via active experimentation in the environment, thus further improving planning performance.

REFERENCES

[1] Anonymous. 2025. Mini Amusement Park: A Road Map to Modeling Businesses. OpenReview preprint. https://openreview.net/forum?id=ZHNbhztmbB Under review.

[2] Xiaoyan Bai, Itamar Pres, Yuntian Deng, Chenhao Tan, Stuart Shieber, Fernanda Viégas, Martin Wattenberg, and Andrew Lee. 2025. Why Can't Transformers Learn Multiplication? Reverse-Engineering Reveals Long-Range Dependency Pitfalls. arXiv:2510.00184 [cs.LG] https://arxiv.org/abs/2510.00184

[3] Taiyu Ban, Lyuzhou Chen, Derui Lyu, Xiangyu Wang, Qinrui Zhu, and Huanhuan Chen. 2025. Llm-driven causal discovery via harmonized prior. *IEEE Transactions on Knowledge and Data Engineering* (2025).

[4] Kristy Choi, Chris Cundy, Sanjari Srivastava, and Stefano Ermon. 2022. Lmpriors: Pre-trained language models as task-specific priors. *arXiv preprint arXiv:2210.12530* (2022).

[5] François Chollet. 2019. On the measure of intelligence. *arXiv preprint arXiv:1911.01547* (2019).

[6] Aidan Curtis, Hao Tang, Thiago Veloso, Kevin Ellis, Joshua B. Tenenbaum, Tomás Lozano-Peréz, and Leslie Pack Kaelbling. 2025. LLM-Guided Probabilistic Program Induction for POMDP Model Estimation. *CoRR* abs/2505.02216 (2025). https://doi.org/10.48550/ARXIV.2505.02216 arXiv:2505.02216

[7] Nicola Dainese, Matteo Merler, Minttu Alakuijala, and Pekka Marttinen. 2024. Generating Code World Models with Large Language Models Guided by Monte Carlo Tree Search. arXiv:2405.15383 [cs.AI] https://arxiv.org/abs/2405.15383

[8] Lin Guan, Karthik Valmeekam, Sarath Sreedharan, and Subbarao Kambhampati. 2023. Leveraging Pre-trained Large Language Models to Construct and Utilize World Models for Model-based Task Planning. arXiv:2305.14909 [cs.AI] https://arxiv.org/abs/2305.14909

[9] Danijar Hafner. 2021. Benchmarking the spectrum of agent capabilities. *arXiv preprint arXiv:2109.06780* (2021).

[10] Danijar Hafner, Jurgis Pasukonis, Jimmy Ba, and Timothy Lillicrap. 2024. Mastering Diverse Domains through World Models. arXiv:2301.04104 [cs.AI] https://arxiv.org/abs/2301.04104

[11] Shibo Hao, Yi Gu, Haodi Ma, Joshua Hong, Zhen Wang, Daisy Wang, and Zhiting Hu. 2023. Reasoning with Language Model is Planning with World Model. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, Singapore, 8154–8173. https://doi.org/10.18653/v1/2023.emnlp-main.507

[12] Peter Jansen, Marc-Alexandre Côté, Tushar Khot, Erin Bransom, Bhavana Dalvi Mishra, Bodhisattwa Prasad Majumder, Oyvind Tafjord, and Peter Clark. 2024. Discoveryworld: A virtual environment for developing and evaluating automated scientific discovery agents. *Advances in Neural Information Processing Systems* 37 (2024), 10088–10116.

[13] Thomas Jiralerspong, Xiaoyin Chen, Yash More, Vedant Shah, and Yoshua Bengio. 2024. Efficient causal graph discovery using large language models. *arXiv preprint arXiv:2402.01207* (2024).

[14] Yichao Liang, Nishanth Kumar, Hao Tang, Adrian Weller, Joshua B. Tenenbaum, Tom Silver, João F. Henriques, and Kevin Ellis. 2025. VisualPredicator: Learning Abstract World Models with Neuro-Symbolic Predicates for Robot Planning. (2025). arXiv:2410.23156 [cs.AI] https://arxiv.org/abs/2410.23156

[15] B. Liu, Yuqian Jiang, Xiaohan Zhang, Qian Liu, Shiqi Zhang, Joydeep Biswas, and Peter Stone. 2023. LLM+P: Empowering Large Language Models with Optimal Planning Proficiency. *ArXiv* abs/2304.11477 (2023). https://api.semanticscholar.org/CorpusID:258298051

[16] Jing Ma. 2024. Causal inference with large language model: A survey. *arXiv preprint arXiv:2409.09822* (2024).

[17] Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. 2025. AlphaEvolve: A coding agent for scientific and algorithmic discovery. arXiv:2506.13131 [cs.AI] https://arxiv.org/abs/2506.13131

[18] Wasu Top Piriyakulkij, Yichao Liang, Hao Tang, Adrian Weller, Marta Kryven, and Kevin Ellis. 2025. PoE-World: Compositional World Modeling with Products of Programmatic Experts. arXiv:2505.10819 [cs.AI] https://arxiv.org/abs/2505.10819

[19] James Requeima, John Bronskill, Dami Choi, Richard Turner, and David K Duvenaud. 2024. Llm processes: Numerical predictive distributions conditioned on natural language. *Advances in Neural Information Processing Systems* 37 (2024), 109609–109671.

[20] Jonathan Richens, Tom Everitt, and David Abel. 2025. General agents need world models. In *Forty-second International Conference on Machine Learning*. https://openreview.net/forum?id=dlIoumNiXt

[21] Philipp Schoenegger and Peter S. Park. 2023. Large Language Model Prediction Capabilities: Evidence from a Real-World Forecasting Tournament. arXiv:2310.13014 [cs.CY] https://arxiv.org/abs/2310.13014

[22] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, L. Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy P. Lillicrap, and David Silver. 2019. Mastering Atari, Go, chess and shogi by planning with a learned model. *Nature* 588 (2019), 604 – 609. https://api.semanticscholar.org/CorpusID:208158225

[23] David Silver, Aja Huang, Christopher Maddison, Arthur Guez, Laurent Sifre, George Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* 529 (01 2016), 484–489. https://doi.org/10.1038/nature16961

[24] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan, and Demis Hassabis. 2017. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. *CoRR* abs/1712.01815 (2017). arXiv:1712.01815 http://arxiv.org/abs/1712.01815

[25] Hao Tang, Keya Hu, Jin Zhou, Sicheng Zhong, Wei-Long Zheng, Xujie Si, and Kevin Ellis. 2024. Code Repair with LLMs gives an Exploration-Exploitation Tradeoff. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (Eds.). http://papers.nips.cc/paper_files/paper/2024/hash/d5c56ec4f69c9a473089b16000d3f8cd-Abstract-Conference.html

[26] Hao Tang, Darren Key, and Kevin Ellis. 2025. WorldCoder, a model-based LLM agent: building world models by writing code and interacting with the environment. In *Proceedings of the 38th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS '24)*. Curran Associates Inc., Red Hook, NY, USA, Article 2243, 65 pages.

[27] Ruoyao Wang, Graham Todd, Ziang Xiao, Xingdi Yuan, Marc-Alexandre Côté, Peter Clark, and Peter Jansen. 2024. Can Language Models Serve as Text-Based World Simulators? 1–17. https://doi.org/10.18653/v1/2024.acl-short.1

[28] Jiannan Xiang, Tianhua Tao, Yi Gu, Tianmin Shu, Zirui Wang, Zichao Yang, and Zhiting Hu. 2023. Language models meet world models: embodied experiences enhance language models. In *Proceedings of the 37th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '23)*. Curran Associates Inc., Red Hook, NY, USA, Article 3295, 21 pages.

[29] Kaige Xie, Ian Yang, John Gunerli, and Mark Riedl. 2024. Making Large Language Models into World Models with Precondition and Effect Knowledge. arXiv:2409.12278 [cs.CL] https://arxiv.org/abs/2409.12278

[30] Zirui Zhao, Wee Sun Lee, and David Hsu. 2023. Large Language Models as Commonsense Knowledge for Large-Scale Task Planning. In *RSS 2023 Workshop on Learning for Task and Motion Planning*. https://openreview.net/forum?id=tED747HURfX

[31] Siyu Zhou, Tianyi Zhou, Yijun Yang, Guodong Long, Deheng Ye, Jing Jiang, and Chengqi Zhang. 2024. WALL-E: World Alignment by Rule Learning Improves World Model-based LLM Agents. arXiv:2410.07484 [cs.AI] https://arxiv.org/abs/2410.07484

A BASELINE IMPLEMENTATION

A.1 WALL-E

To adapt WALL-E [31] for the MAP environment we modify all the original prompts to incorporate our new state and action specifications. Both the state and action specifications are represented using JSON objects. We further modify the prompt for predicting the next state given the action to incorporate and provide examples of state transitions. For few-shot examples, we associate transitions from the training data with an embedding of the action string, then at inference time we retrieve the top K (5) transitions based on the current action. For rule generation, we follow [31]; however, unlike in WALL-E, we do not use different outputs for rules predicting valid and invalid actions. All rules will predict True if the action is valid and False if the action is invalid. For learning the rules, we use randomly sampled trajectories with an invalid action error rate of 0.2, however since random trajectories often result in short trajectories due to bankruptcy, we follow a three step process. First, we sample a set of trajectories by searching the space for high rewards via Monte-Carlo Tree Search (MCTS) equipped with the ground-truth MAP environment for rollouts. Second we sample a trajectory and select a random timestep. Finally we rollout using an agent that selects random actions, producing an invalid action with probability 0.2. We believe this would result in something akin to having a small set of suboptimal trajectories. For MPC game playing experiments we combine the WALL-E world model with the same rule-based heuristic agent used with CASSANDRA. For each step, we first sample multiple distinct initial actions, then we rollout for k (4) steps. We compute the rewards at the end of each rollout and take the action with the highest-scoring rollout.

A.2 WorldCoder

WorldCoder [26] is an online approach to learning a deterministic code world model. As our work presents an offline approach to learning a stochastic world model, it was necessary to adapt WorldCoder into a variant we refer to as OfflineWorldCoder.

The original WorldCoder assumes access to the ground-truth environment, as well as a set of objectives within said environment that it must learn to solve. WorldCoder learns in an online manner, maintaining a replay buffer of transitions observed in the ground-truth environment. It infinitely loops over objectives in the same environment; for each objective it executes a rollout in the environment. In this rollout, actions are selected randomly with probability ϵ. Otherwise, the learned world model and reward functions are used to create a plan satisfying the goal, and the next step of this plan is executed. For most of their environments they use BFS as their planner, for AlfWorld they use MCTS with a manually engineered heuristic for guiding search. If at any time the planner fails to find a plan satisfying the goal, or if the world model becomes inconsistent with a transition in the replay buffer, then code revision is triggered. Revision is via REx code generation [25], conditioned the internal LLM with the existing code. In the case of inconsistency with the replay buffer, a single failing transition is sampled and the learned transition function and reward function are revised wholly independently. In the case of the planner being unabled to find a solution, they are revised together.

Initially the system collects purely random data until the replay buffer is sufficiently large, at which point REx is used twice – once to create the transition function, and again to create the reward function.

First and foremost: as WorldCoder is designed for deterministic environments we only learned a code world model for the deterministic variables of the environment. I.e., we used WorldCoder to optimize the program ϕ for the same objective used by our code learning system for deterministic components:

$$\max_{\phi} \sum_{(\mathbf{s}_{t-1},\mathbf{x}_{t-1},a_{t-1},\mathbf{s}_t,\mathbf{x}_t)\in\mathcal{D}} \log p_{\phi}(\mathbf{s}_t|\mathbf{s}_{t-1},\mathbf{x}_{t-1},a_{t-1}), \qquad (12)$$

As the reward function in the MAP environment (i.e., profit) is non-deterministic, we did not use WorldCoder for reward learning. This lack of a reward function, combined with the stochastic MAP environment complicating search, the inability of the WorldCoder model to produce rollouts as it cannot predict stochastic variables, and the nature of our task being optimization rather than satisfaction (thus requiring a conversion to a planning task), means it would require significant engineering and scientific effort to adapt the online training loop. Given that our system is offline and unlike WorldCoder does not assume access to the ground truth environment, we decided that the best course of action was to remove the planning loop and make WorldCoder offline.

In OfflineWorldCoder, the replay buffer is initialized with the full training set, and REx is run. In WorldCoder, REx normally runs until code is generated that fully explains the replay buffer. However, we found that gpt-4.1-mini was unable to learn the function predicting aggregate ride excitement. Given that aggregate ride excitment penalizes duplicate attractions via a geometric decay, and the exact formula is not in the MAP documentation, it is unsurprising that abducing the rule from a single failed example is virtually impossible. For this reason we instead run REx until convergence, considering REx to be converged if performance does not improve after 10 iterations. Running our experient, REx terminated after the 33rd iteration, failing to improve after iteration 22 (which explained 86.7% of the training transitions).

B EXPERIMENT 1 ADDITIONAL RESULTS.

Table 4 shows prediction RMSE of CASSANDRA and ablations in CoffeeShopSimulator. Prediction accuracy is statistically the same across methods. This is because the environment changes relatively smoothly and changes are correlated across timesteps. However, as indicated in the main paper, average prediction accuracy metrics are not sufficient for planning performance, as a model can achieve high accuracy without capturing causal relations between variables. This leads to poor task performance when the world model is deployed for planning.

C EXPERIMENTAL SETUP

C.1 System Configuration

Since the evaluation can be computationally expensive, especially for MCTS rollouts involving an LLM-world model (Wall-E-MAP), we use AWS Graviton2 instance with 32 vCPUs (16 cores).

C.2 Planning Agents

To convert our world models into agents we use Monte Carlo Tree Search (MCTS). To cope with the stochasticity in the environment, we use a MPC

Method	Money	Coffee	Milk	Customers	Satisfaction	Cleanliness
CASSANDRA	128.8 ± 2.5	2.5 ± 0.0	12.6 ± 0.0	32.6 ± 0.3	3.1 ± 0.1	43.5 ± 0.0
CASSANDRA-Ind	126.1 ± 3.5	2.5 ± 0.0	12.5 ± 0.0	37.0 ± 0.5	3.2 ± 0.0	44.3 ± 0.1
CASSANDRA-Lin	133.2 ± 1.3	2.8 ± 0.0	13.1 ± 0.0	30.3 ± 0.1	3.0 ± 0.0	44.0 ± 0.1
CASSANDRA-R	127.3 ± 3.1	2.6 ± 0.0	13.2 ± 0.0	31.3 ± 0.2	2.9 ± 0.0	44.1 ± 0.1

Table 4: RMSE of predictions in CoffeeShopSimulator

approach: using MCTS to produce an optimized plan up to a finite horizon H, and re-planning after each action. We also modified our states to instead represent sequences of actions (i.e., a node in the search tree represents the distribution of possible states after a given sequence of actions, rather than a specific state – to accurately estimate this we perform all rollouts by sampling the world model from the root node to the horizon). We optimize for predicted money and do not use a value function – i.e., we solely optimize over the timesteps of the horizon. Experiments with MAP suggest that a horizon of 2 actions is adequate under an oracle world model, but a horizon of a single action fails to capture the benefits of staff hiring.

Due to the high branching factor in MAP we sample a subset of the possible actions at each timestep and apply basic heuristics to bias the sampled actions towards useful choices. Specifically:

- We enforce that the wait command is always present exactly once in the set of proposed actions
- We sample at most one price change action
- We do not sample staff movement or guest survey actions
- The number of attraction placement and attraction sale actions is not limited
- We do not sample invalid actions (e.g., invalid price, positions away from paths, etc...)
- We force all ride and shop construction to set prices to their maximum
- We bias the sampling of build commands to move the composition of the park towards a drink shop:food shop:specialty shop ratio of 4:3:2 – this ratio was chosen after playing several games in the MAP environment
- We constrain (x, y) coordinates to only be sampled from the 5 valid locations nearest the entrance
- We de-duplicate actions that are identical up to choice of (x, y) coordinates. Consequently there are at most 2 place staff actions and at most 2 fire staff actions, each for a janitor or mechanic respectively.

MCTS hyperparameters for PRISM variants were 90 iterations (i.e., 90 select, expand, simulate, backpropagate rounds), 4 rollouts per frontier node (i.e., simulation was performed with 4 rollouts executed in parallel), 100 sampled actions per node sampled from the heuristic before deduplication, and a horizon of 3 actions.

For WALL-E there were difficulties due to the slow speed and high-cost of the world model, particularly as inference time increased throughout the trajectory as the state became larger. For this reason is was necesary to increase the time budget per trajectory from 1 hours to 2 hours. Despite this, for it to be possible to run the system under this time budget, it was still necessary to reduce the horizon to 2 actions and the number of iterations to 2. Under these settings the total cost of our MCTS+WALL-E experiments still exceeded $100USD, with a time to complete the trajectory of 117 minutes. Note that the number of rollouts (4) was kept unchanged, and did not impact runtime as rollouts were performed in parallel.

As the original WALL-E paper uses MPC over independent rollouts instead of MCTS (possibly due to the slow inference times and high costs we encountered), we also implemented a non-MCTS MPC for WALL-E. The details are outlined in Appendix A.1.

D CODE OPTIMIZATION

We use gpt 4.1 mini model for both code initialization and refinement to match the baselines (Wall-E-MAP and OfflineWorldCoder). The prompts used for code initialization and refinement are given in F.1.

Code Initialization. : No observation data is used for code initialization; we only provide game documentation and information needed for generating action, precondition, and dynamic functions. The average token usage for code initialization (for 5 attempts) is *248853* input tokens and *21790* output tokens.

Code Revision. : We select 300 training transitions from 84 training trajectories such that they have a ratio of 1:1 successful ($\sigma = True$) and failing ($\sigma = False$) transitions. We notice that exposing the refinement process to enough failing transitions is effective in learning strong preconditions. These transitions are refined as mentioned in Algorithm 1. Each refinement loop takes an average of 10.9 seconds and has a token usage of *5118* input tokens and *540* output tokens.

Ablation Study. : To understand the importance of the code refinement algorithm, we present an ablation of CASSANDRA where it uses code that was initialized by an LLM, but not refined. We call it CASS-init. The results in Table 5 show the significant improvement the refinement process brings to transition prediction.

D.1 Refinement Score Implementation

$RefinementScore$ is of two types, depending on the type of function r_t^j is trying to improve. They are 1) Precondition Error Reduction Score ($PERS$) (how much the precondition function error reduces with the refinement) and 2) Observation Difference Reduction Score ($ODRS$) (how much the action or dynamic function can reduce the observation difference after the refinement). We generate $K = 3$ refinements for each error (using experiments with 50 train transitions, we notice refinements beyond 3 tend to be irrelevant, adding little value).

We use a validation set V (of 300 transitions randomly selected from the training data) to assess the quality of a refinement r_t^j to the code ϕ' generated for a transition t in the training set. We select the highest scoring refinement only if it shows an average improvement over the validation trajectories as well (we denote the function that averages $RefinementScore$ applied to validation trajectories as $ValidationScore$ (VS)). As mentioned in Algorithm 1, we only choose refinements that have a $VS > 0$ since any value less than 0 indicates that the refinement worsens the predictions of ϕ' for the validation set.

Let an arbitrary transition in the validation set be $(s_{v-1}, x_{v-1}, \sigma_{v-1}, s_v, x_v)$. Let s'_v, σ'_{v-1} be the observation vector and action validity computed by $f_{\phi'}$ and $\hat{s_v}$ and $\hat{\sigma_{v-1}}$ be the observation vector and action validity computed after refinement r_t^j is applied to ϕ' to get $\phi'_{r_t^j}$, i.e. $\hat{s_v}, \hat{\sigma_{v-1}} = f_{\phi'_{r_t^j}}(s_{v-1}, x_{v-1}, \sigma_{v-1})$. Algorithm 3 explains the function's implementation in detail.

Model	$Cat^{D}_{acc}\uparrow$	$Num^{D}_{rmse}\downarrow$	$Inv^{D}\downarrow$	$\sigma_{acc}\uparrow$	$\sigma_{prec}\uparrow$	$\sigma_{rec}\uparrow$	$\sigma_{F1}\uparrow$
OfflineWorldCoder [26]	1.000	0.483	0.001	0.905	0.844	0.995	0.913
CASS-init	0.978	0.187	0.019	0.859	0.794	0.968	0.872
CASSANDRA	1.000	0.176	0.012	0.945	0.902	0.998	0.948

Table 5: Evaluating ablation CASS-init - without code refinement. Cat_{acc} **denotes prediction accuracy for categorical variables.** Num_{rmse} **denotes the average RMSE of numerical variables.** Inv **is the average invalid value rate.** σ_{acc}, σ_{prec}, σ_{rec} **and** σ_{F1} **are the accuracy, precision, recall and F1 scores of action validity prediction. Superscript 'D' indicates the deterministic component of the observation. ↑ indicates higher number is better and ↓ indicates lower number is better. CASSANDRAshows significant improvement after code refinement in both observation and action validity prediction.**

Algorithm 3 Code Refinement Heuristic Scoring Function

```
procedure PERS($\sigma_{v-1}, \sigma'_{v-1}, \hat{\sigma_{v-1}}$)
    return $1_{\hat{\sigma_{v-1}}=\sigma_{v-1}} - 1_{\sigma'_{v-1}=\sigma_{v-1}}$
end procedure

procedure ODRS($s_{v-1}, s_v, s'_v$)
    $d_1, \cdots, d_L \leftarrow DeepDiff(s_{v-1}, s_v)$
            ▷ Assuming values that are lists or dictionaries are treated as on-numerical entries
    $val^0_1, \cdots, val^0_L \leftarrow GetValues([d_1, \cdots d_L], s_{v-1})$
    $val_1, \cdots, val_L \leftarrow GetValues([d_1, \cdots d_L], s_v)$
    $val'_1, \cdots, val'_L \leftarrow GetValues([d_1, \cdots d_L], s'_v)$
    $score \leftarrow 0$
    $total \leftarrow 0$
    for $i \leftarrow 0 to L - 1$ do
        if $DiffType(d_i) \neq values\_changed$ then
            $score \leftarrow score - 1$
            $total \leftarrow total + 1$
        else
            if $IsNumerical(val_i)$ then
                if $abs(val_i - val'_i) <= abs(val_i - val^0_i)$ then
                    $score \leftarrow score + 1$
                    $total \leftarrow total + 1$
                else
                    $score \leftarrow score - 1$
                    $total \leftarrow total + 1$
                end if
            else
                if $val_i = val'_i$ then
                    $score \leftarrow score + 1$
                    $total \leftarrow total + 1$
                else
                    $score \leftarrow score - 1$
                    $total \leftarrow total + 1$
                end if
            end if
        end if
    end for
    return $score/total$
end procedure

procedure RefinementScore($r^j_t, \phi'$, V)
    $funcType \leftarrow FunctionType(r^j_t)$
    for $i \leftarrow$ 0 to N-1 do
        $s_{v-1}, x_{v-1}, \sigma_{v-1}, s_v, x_v \leftarrow V[i]$
        $s'_v, \sigma'_{v-1} \leftarrow f_{\phi'}(s_{v-1}, x_{v-1}, \sigma_{v-1})$
        $\hat{s_v}, \hat{\sigma_{v-1}} \leftarrow f_{\phi'_{r^j_t}}(s_{v-1}, x_{v-1}, \sigma_{v-1})$
        $score \leftarrow 0$
        if $funcType = PreconditionFunction$ then
            $score = score + PERS(\sigma_{v-1}, \sigma'_{v-1}, \hat{\sigma_{v-1}})$
        else if $funcType = ActionFunction \lor funcType = DynamicFunction$ then
            $score = score + (ODRS(s_v, x_v, \hat{s_v}) - ODRS(s_v, x_v, s'_v))$
        end if
    end for
    return $score/N$
end procedure
```

E STOCHASTIC COMPONENT IMPLEMENTATION

In this section we provide some additional implementation details for our stochastic component.

Simulated Annealing. We use a standard simulated annealing search to improve DAG structures. We generate candidate solutions by randomly considering graph modifications and only keeping ones that maintain acyclicity. We consider flipping a directed edge, removing one and adding one. Alternatively, we could use other strategies to suggest actions, such as via word embedding similarity, which could improve the search. We use a geometric cooling schedule with $\alpha = 0.99$. We run 100 optimization steps per initial DAG for the coffeshop environment and 200 for MAP.

DAG Sampling. We use structured outputs to ensure consistent outputs when sampling DAGs from the LLM. Exact prompts are available in appendix F. For topological sorting, our structured format includes a field for thinking traces and a single string representing the next variable in the topological sort. For eliciting dependencies, our structured format contains the thinking field and a list of variable names chosen. Empirically we notice that this tends to sample sparse graphs, as LLMs will tend to not answer with too many dependencies (likely due to context dilution when the number of variables is large). Still, we find that is acceptable as the search procedure in practice tends to add edges.

LLM prior. We use structured outputs to ensure consistent outputs from the LLMs. Exact prompts are available in appendix F. Our structured output schema includes a field for thinking traces and a final answer as a boolean variable. We extract log probabilities for the corresponding answer token and return it as the prior value. In practice, these values tend to be large (either for the token 'true' or 'false', depending on the answer provided). As such, the LLM provides a nearly hard constraint for the search, rejecting DAGs with semantically implausible dynamics. We use $\lambda_1 = 10$ to bring the sparsity factor in a similar scale to the pinball loss. We use $\lambda_2 = 100$. This turns the LLM prior into a steep constraint. The optimization first finds structures that match the environment dynamics ($\log p(E|w(V, \mathcal{L}) \approx 10^{-2}$). Then the search is concentrated among such graphs.

F PROMPTS

MAP Environment Gameplay Rules:

```
## Game Mechanics

The purpose of the game is to maximize a theme
    park's profit given a starting budget and
    timeframe. As the CEO of the park, you perform
     one action at the start of the day. The park
    then opens and guests interact with the park
    for a full day, which consists of 500 ticks.
```

```
There are three difficulty modes to the game ("easy", "medium", and "hard").
  - Easy: All attractions are available from the beginning. Water tiles are disabled. Short time horizon (50 days by default)
  - Medium: Only yellow attractions are available from the beginning, other attractions must be researched. Layouts can include water tiles. Medium time horizon (100 by default)
  - Hard: Only yellow attractions are available from the beginning, other attractions must be researched. Layouts can include water tiles and terraforming actions (i.e. adding/removing paths/water) are enabled. Long time horizon (250 by default)

There are 7 primary components to the game. We provide a high level overview here, and a detailed description of each follows.
- The Park. This is defined by a square grid (defaults to 20x20) and contains all other components. The theme park has an entrance and an exit, which are connected by a path.
- Terrain. There are three kinds of terrain: Paths, Water, and Empty.
- Rides. Rides are one of two types of attractions you can place in your theme park. They are the core of the theme park, and are what draw guests in and keep them happy. There are three subtypes of rides: Carousels, Ferris Wheels, and Roller Coasters. Each has four subclasses: yellow, blue, green, and red.
- Shops. Shops are the second type of attractions you can place in your theme park. They allow you to cater your guests' needs. There are three subtypes of shops: Drink shops, Food shops, and Specialty shops. Drink shops alleviate the thirst of guests. Food shops alleviate the hunger of guests. Specialty shops provide a range of utilities. There similarly four subclasses for each subtype: yellow, blue, green, and red.
- Staff. Staff can be hired to maintain your park. There are two types of staff: janitors and mechanics. Janitors keep your park clean, while mechanics can repair rides that need maintenance.
- Guests. This is who you build the park for! Guests enter your park to enjoy the attractions you've built.
- Research. Early on, you only know how to build certain rides. Investing in research will allow you to build more subclasses of rides and shops as time goes on.

A core feature of your park is your park rating. This rating depends on several factors in your park and is a driving force in how many guests will come visit your park.

### Terrain
- Empty tiles: A blank tile on which something can be built.
- Path tiles: The tiles used by guests to move around your park. All attractions must be placed on an empty tile that is adjacent to a path tile.
- Water tiles: The excitement of any ride adjacent to a water tile is increased by 1, however it nothing can be built on it.

In hard mode, terrain can be terraformed. Paths can be built ($1000) and removed ($2500). Water tiles can similarly be built ($5000) and removed ($10000)

### Rides
Rides are the core of your theme park. They drive guest attraction and guest happiness. Rides have several key attributes:
- Capacity. How many guests can fit on your ride at a single time. The cummulative capacity of your park is also a key factor in how many guests can visit your park.
- Excitement. How exciting a ride is. Higher excitement scores lead to greater guest happiness. A high excitement score is also crucial to your park rating.
- Intensity. How intense the ride is. Keeping your average intensity balanced (i.e., as close to 5 as possible) will create a park that caters to a wide range of guests, and will help increase your park rating.
- Ticket price. How much money guests have to spend to ride the ride. You are here to make a profit after all. Note that if a guest does not have enough money to pay the ticket price, they will be rejected by the ride, which will decrease their happiness.
- Cost per operation. How much it costs each time you operate the ride.

Rides have no fixed costs, their operating costs depends solely on the number of times it is operated each day.
Rides only operate if guests have boarded, and wait for a few turns after the first guest boards to see if more guests will join. This wait is longer for rides with a larger capacity, but decreases as more guests board the ride. A full ride will always operate.
Rides will sometimes breakdown after operating. While broken down, it is out of service and will not accept guests. Having broken rides will negatively impact your park rating, so it is wise to hire mechanics to fix broken rides.

There are three subtypes of rides: Carousels, Ferris Wheels, and Roller Coasters. Each subtype has distinct characteristics:
```

```
- Carousels a cheap to build, operate, and they rarely breakdown. However, they provide limited other benefits.
- Ferris Wheels are an intermediate ride option. They are the ride subtype with the highest capacities.
- Roller Coasters are an expensive, but high-value ride. They boast the highest excitement and intensity scores, but breakdown more than the other types of rides.

Each ride also has four possible subclasses. In medium and hard mode, you only start with the yellow version of each ride, and have to perform research to unlock other subclasses of rides. Here is a general outline of subclasses
- Yellow rides are starter rides. Cheap to build and operate, but otherwise unremarkable.
- Blue rides provide an immediate step up. Often with higher excitement, intensity, capacity, and maximum allowable ticket prices.
- Green rides are the subsclass that provides the highest capacity, but have lower excitement and worse intensity values.
- Red rides are thrill rides. High excitement, high intensity, and have the highest possible ticket prices, but they prone to breaking down.

For the exact parameters of each ride, see [All Rides](#all-rides)

### Shops
Shops are necessary to adequately cater to guest needs. Shops have several key attributes:
- Operating costs: how much it cost to stock the shop each day. Unlike rides, shops cost a fixed amount per day.
- Item price: the price guests pay for the item being sold.

Similar to rides, there are three subtypes of shops:
- Drink. Drink shops sell drinks that quench the thirst of guests.
- Food. Food shops sell food that satiate the hunger of guests.
- Specialty. Specialty shops provide a range of services based on their subclass. See [All Shops](#all-shops) for a description of each. Notably, guests will never seek out specialty shops, they will only visit specialty shops if they walk by them.

Again similar to rides, shops have subclasses:
- Yellow food and drink shops are starter shops. They provide a basic version of their product.
- Blue food and drink shops sell an improved (more hunger satiating / thirst quenching) version of their respective product.
- Green food and drink shops provide multiple benefits instead of a single benefit.
- Red drink shops sell coffee which caffeinate guests. Red food shops are luxury food items, providing highly satiating food that also boost happiness.

For exact parameters of each shops, see [All Shops](#all-shops)

### Staff
There are two types of staff: a janitor and a mechanic. Each staff has a salary: $25 for mechanics and $100 for janitors.

Janitors will roam the park, generally moving toward dirtier areas. When on dirty tiles, janitors will clean them.

Mechanics will move toward rides that are broken down, with a preference towards nearer broken rides. When they reach a tile containing a ride that needs maintenance, mechanics will perform repairs until the ride is operational again. Rides will require 0.05% of their total building costs to fully repair once broken, and the speed of repair is proportional to this cost.

Multiple staff can occupy the same tile. Multiple of the same kind of staff on a tile will multiplicatively increase the speed at which the tile is cleaned / repaired.

### Guests
Guests are what the park is made for. The amount of guests that visit your theme park is based on the park's rating and the overall capacity of the park.
*Capacity* determines both how many guests can be in the park at any given time, as well as how many potential guests consider visiting the park.
Capacity is entirely determined by the cummulative capacity of the current rides in the park.
**NOTE:** Since only rides increase capacity, a park with no rides will receive no visitors. We aren't making a food hall.

*Park Rating* determines the likelihood that a potential guest decides to enter the park and become a real guest. New guests cannot enter the park if the park is currently at capacity.
Park rating can be increased by increasing the total excitement of the park, and by having guests leave your park happy. Park rating will drop if the park is dirty, rides have low uptimes (i.e., are out of service for portions of the day), or if the average intensity of rides is too high or too low.
```

```
Each guests that visits your park will bring with
    them some amount of money. If they run out of
    money, they will leave your park.
Each guests also has a finite amount of energy.
    Every step they take in the park will decrease
     this energy. If they run out of energy, they
    will leave your park.

Guests also have hunger, thirst, and happiness
    levels. Hungry guests will seek out food shops
    . Thirsty guests will seek out drink shops.
    Unhappy guests will seek out rides. If a guest
    's hunger or thirst levels get too high, it
    will negatively impact their happiness. If
    guests become too unhappy, too thirsty, or too
     hungry, they will leave your park.

If a guest has all their needs met, they will
    target any attraction (except specialty shops,
     see below). If guests pass by another
    attraction on their way to their target, they
    may visit that attraction. Guests like novelty
     and will favor attractions they have visited
    less frequently. Guests will also favor
    attractions that are nearby, but they will
    never visit the same attraction twice in a row
    . If a guest visits an attraction that they
    cannot afford or that is currently broken down
    , their happiness will be affected.
**NOTE:** Guests will never seek out specialty
    shops. They will only visit specialty shops if
     they pass by them on their way to another
    attraction.

Guests sometimes litter. This decreases the
    cleanliness of the path or attraction they are
     currently on. Unhappy guests are more likely
    to litter. If guests visit dirty tiles, it
    negatively impacts their happiness. If a guest
     visits a ride that is too dirty, they may
    choose to go elsewhere which will negatively
    impact their happiness.

In hard mode, guests may have preferences, meaning
     they will only interact with a subset of
    attractions. If a guests visits a ride that
    does not match their preference, it will
    negatively impact their happiness. Providing
    guests with information through an information
     booth (blue specialty shop) will allow them
    to know ahead of time which attractions match
    their preference.

**NOTE:** You can learn more about guests by
    surveying them through the SurveyGuest action.
     This provides information about why the guest
     left the park, what the guest preferences are
    , and more. You can choose how many guests to
    survey (up to a maximum of 25) for a price of
    $1000 per guest.

### Research

In easy mode, all attractions are avaiable from
    the start and research is disabled. This
    section is only relevant to medium and hard
    mode.

At the start of the game, you only the yellow
    subclass of each attraction is available. To
    learn how to build more subclasses, you have
    to invest in research. To do this, you have to
     set your research, which includes setting how
     fast you want to perform research and the
    topic(s) of research (where the topics are
    attraction subtypes). Once set, research will
    be performed each day according to your
    settings. This persists until you change the
    research settings, until you run out of funds,
     or you successfully research all possible
    subclasses for the specified research topics.
    In the latter two cases, the research speed
    will be set back to "none". Research will
    always unlock new subclasses in the following
    order: blue, green, red. Once a new attraction
     has been unlocked, research will continue on
    the next topic in your list of topics.

**NOTE:** if, for example, you want to unlock the
    red roller coaster as fast as possible, you
    will want to set the research speed to "fast"
    and set your research topics to ONLY ["roller
    coaster"].

Performing research faster requires more money.
    Below are the research speeds and their rates:
- "none": 0$/day; research is halted at this speed
    .
- "slow": 2000$/day.
- "medium": 10000$/day.
- "fast": 50000$/day

The default setting is a research speed of "none"
    and all attraction subtypes selected as topics
    .

## All Rides

### Carousels

**Yellow**
- Building Cost: 250
- Cost per Operation: 1
- Capacity: 7
- Max Ticket Price: 3
- Excitement: 1
- Intensity: 1
- Breakdown Rate: 0.001

**Blue**
- Building Cost: 1250
- Cost per Operation: 2
- Capacity: 14
- Max Ticket Price: 6
- Excitement: 4
- Intensity: 5
- Breakdown Rate: 0.002

**Green**
- Building Cost: 7500
- Cost per Operation: 16
- Capacity: 26
- Max Ticket Price: 7
- Excitement: 2
- Intensity: 4
- Breakdown Rate: 0.003

**Red**
- Building Cost: 15000
- Cost per Operation: 12
- Capacity: 18
- Max Ticket Price: 11
- Excitement: 7
- Intensity: 5
- Breakdown Rate: 0.005

### Ferris Wheels

**Yellow**
- Building Cost: 500
- Cost per Operation: 6
- Capacity: 10
- Max Ticket Price: 4
- Excitement: 3
- Intensity: 2
- Breakdown Rate: 0.006

**Blue**
- Building Cost: 3750
- Cost per Operation: 8
- Capacity: 22
- Max Ticket Price: 5
- Excitement: 6
- Intensity: 3
- Breakdown Rate: 0.009

**Green**
- Building Cost: 37500
- Cost per Operation: 40
- Capacity: 42
- Max Ticket Price: 9
- Excitement: 5
- Intensity: 7
- Breakdown Rate: 0.023

**Red**
- Building Cost: 55000
- Cost per Operation: 28
- Capacity: 24
- Max Ticket Price: 12
- Excitement: 9
- Intensity: 8
- Breakdown Rate: 0.018

### Roller Coasters

**Yellow**
```

```
- Building Cost: 1000
- Cost per Operation: 10
- Capacity: 5
- Max Ticket Price: 15
- Excitement: 4
- Intensity: 6
- Breakdown Rate: 0.01

**Blue**
- Building Cost: 10000
- Cost per Operation: 25
- Capacity: 8
- Max Ticket Price: 20
- Excitement: 8
- Intensity: 8
- Breakdown Rate: 0.02

**Green**
- Building Cost: 30000
- Cost per Operation: 40
- Capacity: 16
- Max Ticket Price: 18
- Excitement: 6
- Intensity: 9
- Breakdown Rate: 0.03

**Red**
- Building Cost: 75000
- Cost per Operation: 50
- Capacity: 10
- Max Ticket Price: 50
- Excitement: 10
- Intensity: 10
- Breakdown Rate: 0.033

## All Shops

### Drink Shops

**Yellow**
- Building Cost: 100
- Operating Cost: 16
- Max Item Price: 2
- Thirst Reduction: 0.5

**Blue**
- Building Cost: 1750
- Operating Cost: 75
- Max Item Price: 6
- Thirst Reduction: 0.9

**Green**
- Building Cost: 17500
- Operating Cost: 280
- Max Item Price: 10
- Thirst Reduction: 0.8
- Happiness Boost: 0.4

*In addition to quenching thirst, green drink
    shops additionally provide a boost to guest
    happiness.*
```

```
**Red**
- Building Cost: 48000
- Operating Cost: 600
- Max Item Price: 15
- Thirst Reduction: 0.4
- Energy Boost: 50
- Caffeinated Steps: 50

*Red drinks caffeinate guests, which boosts a
    guest's energy and allows them to move twice
    as fast*

### Food Shops

**Yellow**
- Building Cost: 150
- Operating Cost: 28
- Max Item Price: 4
- Hunger Reduction: 0.4

**Blue**
- Building Cost: 3000
- Operating Cost: 150
- Max Item Price: 10
- Hunger Reduction: 0.8

**Green**
- Building Cost: 32000
- Operating Cost: 500
- Max Item Price: 18
- Hunger Reduction: 0.6
- Thirst Reduction: 0.6

*Green food shops both satiate hunger and quench
    thirst.*

**Red**
- Building Cost: 60000
- Operating Cost: 1000
- Max Item Price: 25
- Hunger Reduction: 0.9
- Happiness Boost: 0.5

*Red food shops sell luxury food. This greatly
    satiates hunger and increases happiness*

### Specialty Shops

NOTE: Guests will not target specialty shops, and
    will only visit specialty shops if the walk
    adjacent to one.

**Yellow (Souvenir Shop)**
- Building Cost: 250
- Operating Cost: 80
- Max Item Price: 12
- Happiness Boost: 0.3

*These provide a happiness boost to guests that
    buy them the first time, but this happiness
    boost diminishes with each subsequent souvenir
     purchased.*
```

```

**Blue (Information Booth)**
- Building Cost: 10000
- Operating Cost: 200
- Max Item Price: 5

*These provide information to guests about the
    rides in the park and ensure that guests only
    visit rides that fall within their budget and
    preferences. These do not provide guests about
     information related to the cleanliness or
    operation (i.e., if an attraction is out of
    service) of an attraction.*

**Green (ATM)**
- Building Cost: 50000
- Operating Cost: 500
- Max Item Price: 2
- Money Withdrawal: 50

*ATMs allow guests to withdraw more money. The
    amount of money withdrawn decreases
    exponentially with every subsequent withdrawal
    , to a minimum of 5*

**Red (Billboard)**
- Building Cost: 10000
- Operating Cost: 0
- Max Item Price: 0
- Thirst Boost: 0.5
- Hunger Boost: 0.5

*Billboards make guests more hungry and thirsty,
    will reset the visit count of attractions (
    meaning that guests are more likely to revisit
     attractions), and, if the guest has less than
     $20, will set the guest's target to an ATM.*
```

CoffeeShopSimulator Environment Gameplay Rules:

```
    """
Welcome to the manager's seat! Running a
    successful coffee shop is an art. Here's what
    you need to know to become a local legend.

The Heart of the Business: Customer Happiness

Your central goal is keeping customers happy. A
    shop's **satisfaction** level is the key to
    everything. This mood is shaped by the **
    quality of your shop** upgrades make a big
    differenceand the **price** you set.
    Finding the right balance is crucial, as
    customers have strong opinions about what a
    coffee should cost. Remember, your reputation
    lingers, so yesterday's vibe will influence
    today's crowd.

Drawing a Crowd

```

```
The number of customers you attract is a direct
    result of their happiness. A buzzing,
    satisfied shop creates a **word-of-mouth**
    effect that tends to increase your future
    customers! Your shop's **upgrade level** also
    determines its basic appeal and how many
    potential customers are in the area. However,
    be prepared! You can only serve as many people
     as you have **beans and milk** for, so a
    sudden rush can leave you empty-handed if you
    don't plan ahead.

The Daily Ledger

At the end of each day, you'll see the results of
    your hard work. The number of customers you
    served and the price you charged will
    determine your **revenue**, but don't forget
    you have daily costs to cover. Every cup sold
    uses up your precious **inventory**, and every
     visitor makes the shop a little less **clean
    **. Mastering this daily cycle of cause and
    effect is your path to building a coffee
    empire.

"""
```

F.1 Code Optimization

Environment Background Info - TPT

```
*** Environment Background ***
You are working with a theme park world model. The
     infomation about the state of the world is
    represented in JSON format below.
You are also given a set of actions that can be
    taken to change the state of the world.
Finally, you are given a description of the Theme
    Park environment that is modeled.

***Park State Specification***

The theme park is represented by {PARK_SIZE} x {
    PARK_SIZE} cartesian grid.
The state of theme park is represented in the
    following json structure.

State Specification:
{STATE_SPEC}

Action Specification:
{ACTION_SPEC}

***Gameplay Rules***
{GAMEPLAY_RULES}

Below is an example of how a state of the world is
     represented:
{EXAMPLE_STATE}

Below are examples of how some actions may
    initialize objects in the state.
{INITIALIZERS}
```

JSON Patch Instructions

~~~
### How to Use JSON Patch to Define Changes to
    JSON Files

**JSON Patch** (based on [RFC 6902] is a standard
    format for describing updates to a JSON
    document. A **JSON Patch** document is an
    array of operations. Each operation tells you
    what to change and where.

---

## 1. Structure of a JSON Patch

Each patch operation is a JSON object with:

| Field   | Description
| `op`    | Operation to perform: `add`, `remove`,
     `replace`, `move`, `copy`, or `test` |
| `path`  | JSON Pointer (`/path/to/key`) to the
    location in the document. Array indices are
    specified as `/path/to/key/0`. If the array is
     empty, use index 0.                 |
| `value` | (Optional) New value for `add`, `
    replace`, `test`                         |
| `from`  | (Required for `move` and `copy`) The
    source location                        |

---

## 2. Supported Operations (with Examples)

Let's say you have a state that has a list of cars
     in the path /car_info/cars. We can work with
    this example to understand the operations.
Note that the path is a JSON Pointer, so you need
    to use the correct syntax for the path.

### `add`

Adds a value to an object or array.

When a variable that should be an array is empty,
    create it first.

Example: Adding a new car to the list of cars.
```json
{
  "op": "add",
  "path": "/car_info/cars",
  "value": [
    {
      "id": "car-1",
      "make": "Toyota",
      "model": "Camry",
      "year": 2020
    }
  ]
}
```
~~~

~~~
To add a new member to the cars variable, below is
     an example:
```json
{
  "op": "add",
  "path": "/car_info/cars/2",
  "value": {
    "id": "car-2",
    "make": "Toyota",
    "model": "Camry",
    "year": 2020
  }
}
~~~
Never simply add an empty dictionary to the list.
    Always add a fully populated dictionary.
If the path exists, it replaces the current value
    (like `replace`). If it doesn't, it adds the
    new value.
**Note that when adding to an array, the index
    must be specified. Do not overwrite the entire
     array.**
**Empty arrays don't have any index. If there are
    no cars, the path is `/car_info/cars`.**

### `remove`

Deletes a value from the document.

```json
{
  "op": "remove",
  "path": "/car_info/cars/0"
}
```

Removes the first car (index 0).

### `replace`

Replaces the value at the given path. Only replace
     INDIVIDUAL VALUES. Do not replace the entire
    dict object!

```json
{
  "op": "replace",
  "path": "/car_info/cars/0/make",
  "value": "Ford"
}
```

Changes the `make` field to "Ford".

## 3. Path Syntax (JSON Pointer)

* `/foo`   the key `"foo"` at the root level
* `/foo/0`   the first element in the array
    under key `"foo"`
* `/a~1b`   refers to key `a/b` (because `/` is
    escaped as `~1`)
* `/m~0n`   refers to key `m~n` (because `~` is
    escaped as `~0`)

Make sure you produce a valid JSON Patch. This
    includes following the JSON Patch
    specification, and expressing it with a valid
    json. Do not include any raw computations in
    your prediction. Do not include comments or
    other text in your prediction as it will be
    parsed automatically.
Return all the operations in a single JSON Patch
    array.
```

Action Function Format:

```
*** Action Function Format ***
The action function is a Python function that
    takes the current state and action as input,
    and returns a JSON patch that captures the
    changes that the action makes to the state.
The format for a JSON patch is described in the
    instructions below:
{patch_instructions}

*Guidelines:*
  - The function should be implemented to strictly
       follow the description of the function
      provided.
  - Make sure the function sticks to the state and
       action specifications provided.
  - Don't use any libraries other than the
      standard Python libraries.

**Output Format**
First plan the function implementation step by
    step within <thought> and </thought> tags
    accounting for all the edge cases.
Then generate the code for the function in <code>
    and </code> tags.

<code>
```python
def action_name(state, action):
    # Your code here
    return <JSON patch dict>
```
</code>
```

Precondition Function Format:

```
*** Precondition Function Format ***
The precondition function is a Python function
    that takes the current state and action as
    input, and returns a boolean value indicating
    whether the action can be executed in the
    current state.
```
```

~~~
It also returns a feedback string that explains
    why the action cannot be executed if it
    returns False. Feedback simply says "
    Precondition does not fail" if the action can
    be executed.

*Guidelines:*
  - The function should be implemented to strictly
       follow the description of the function
      provided.
  - Make sure the function sticks to the state and
       action specifications provided.
  - Don't use any libraries other than the
      standard Python libraries.

**Output Format**
First plan the function implementation step by
    step within <thought> and </thought> tags
    accounting for all the edge cases.
Then generate the code for the function in <code>
    and </code> tags.

<code>
```python
def precondition_function_name(state, action):
    # Your code here
    return <boolean value>, <feedback string>
~~~
</code>
```

Dynamic Function Format:

```
*** Dynamic Action Format ***
The dynamic action is a Python function that takes
     the current state as an input and returns a
    JSON patch that captures the change to the
    state due to some continuous event (
    independent of any action).
The format for a JSON patch is described in the
    instructions below:
{patch_instructions}
*Guidelines:*
  - The function should be implemented to strictly
       follow the description of the function
      provided.
  - Make sure the function sticks to the state and
       action specifications provided.
  - Don't use any libraries other than the
      standard Python libraries.

**Output Format**
First plan the function implementation step by
    step within <thought> and </thought> tags
    accounting for all the edge cases.
Then generate the code for the function in <code>
    and </code> tags.

<code>
```python
def dynamic_function_name(state):
    # Your code here
    return <JSON patch dict>
```
</code>
```

**Code Initialization Prompt Template ($T_{init}$): Generate Action and Precondition Functions**
*System Prompt Template*

```
You are a world model agent for a given
    environment. The background of the environment
     is provided below.
{background_info}.
You need to generate a complete action function
    implementation including both the description
    and the Python code. An action function is a
    Python function that takes the current state
    and action as input, and returns a JSON patch
    that captures the changes that the action
    makes to the state.
You also need to generate complete precondition
    function implementations for each action. A
    precondition function is a Python function
    that takes the current state and action as
    input, and returns a boolean value along with
    a string feedback indicating whether the
    action can be executed in the current state.
The functions should only change the state
    varibles in the deterministic state
    specification provided below:
{DETERMINISTIC_STATE_SPEC}
A function description is a JSON with the
    following keys:
name: <function_name>
purpose: <function_purpose>
implementation_details: <
    function_implementation_details>

The format for action function is described below:
{action_function_format}
The format for precondition function is described
    below:
{precondition_function_format}

You need to generate a complete action function
    implementation for the action specification
    provided.
Note: The precondition functions will check for
    the preconditions, action function can assume
    that the preconditions are SATISFIED.
Remember to keep the action description as general
     as possible, so that it can be used in any
    state of the world model.

Think carefully and write all preconditions that
    need to be satisfied. Remember to keep the
    preconditions as general as possible, so that
    they can be used in any state of the world
    model.
The preconditions should be granular, each
    checking for a single aspect of the state
    change.
```
```

```
Note: Changes to the state independent of any
    action are captured by a dynamic function that
     is executed after the action function. So,
    don't include any logic in the action function
     that is not an affect of the action.

**Output Format**
Return a JSON with the following keys and values:
"action_name": <action_name>,
"action_description": <action_function_description
    >,
"code": <action_function_code>,
"preconditions": [<precondition_1_output>, <
    precondition_2_output>, ...]

Where each precondition output contains:
"precondition_description": <
    precondition_function_description>,
"code": <precondition_function_code>
```

User Prompt Template

```
Now generate the complete action function
    implementation and precondition function
    implementations for the following action:
{action_spec}
```

Generate Dynamic Function *System Prompt Template*

```
You are a world model agent for a given
    environment. The background of the environment
     is provided below.
{background_info}.
You need to generate a complete dynamic function
    implementation including both the description
    and the Python code.
The world model agent needs to be able to predict
    the next state given the current state. This
    can only be done by capturing default changes
    that happen in the environment dict which is
    not influenced by any action being taken.

A function description is a JSON with the
    following keys:
name: <function_name>
purpose: <function_purpose>
implementation_details: <
    function_implementation_details>
The format for dynamic function is described below
    :
{dynamic_function_format}
When generating the dynamic function, only
    consider computing changes to the elements of
    the deterministic state specification provided
     below:
{DETERMINISTIC_STATE_SPEC}
Note: An action function may be applied to the
    state first, then the patch generated by the
    dynamic function is applied to the modified
    state to compute the next state's
    deterministic values correctly.

Only include logic in the dynamic function if it
    is needed to compute the correct next state.
```

```
Note: The dynamic function should capture all the
    continuous events that happen in the world
    model. If multiple continuous events are
    happening at the same time, combine them into
    a single dynamic function.

**Output Format**
Return a JSON with the following keys and values:
"dynamic_description": <
    dynamic_function_description>,
"code": <dynamic_function_code>
```

User Prompt Template

```
Now look at the deterministic state specification
    and generate a complete dynamic function
    implementation.
```

Code Refinement Prompt Template (T_{refine})
System Prompt Template:

```
You are a World Model Agent for a given
    environment. The background of the environment
     is provided below.
{background_info}.
You are using Python functions to predict the
    changes to the world state based on an action
    taken, which is called a transition.
You are given one of more functions that are
    currently being used and need to be refined.
    Refinement is done to address shortcomings in
    the functions which will be described.
A refinement can have the following forms: Adding
    a new function, removing an existing function,
     or replacing an existing function.
You are also given context. Take your time to
    understand the context and identify the
    shortcomings with the function set before
    proposing the refinements.
You are also given the number of refinements you
    need to propose. Think carefully and return
    the best refinements you can come up with.

Refinements can have the following forms: Adding a
     new function, removing an existing function,
    or replacing an existing function.

Assume that ALL the important environment details
    required to predict the changes to the world
    state are provided in the background.

Functions might need to produce JSON patches to
    update the state. The format for a JSON patch
    is described in the instructions below:
{patch_instructions}

While the functions can use all the variables in
    the state spec, they only need to correctly
    compute the changes to the variables in
    deterministic values (in state spec given
    below).
{DETERMINISTIC_STATE_SPEC}
```

```
The formats of the three types of Python functions
     are provided below (stick to this format
    depending on the type of function you are
    adding/replacing):
{action_function_format}
{precondition_function_format}
{dynamic_function_format}

When generating code, make sure to include both
    the function description and the complete
    Python code implementation (that handles
    necessary edge cases).

Note: Here is how the functions are applied to the
     state:
- First, preconditions are checked. If they are
    satisfied, action function is executed and the
     resulting patch is applied to the state.
- Then, dynamic function is executed and the
    resulting patch is applied to the modified
    state. (Even if the action function patch is
    not applied, the dynamic function patch is
    still applied.)

Precondition functions can be granular, focusing
    on a narrow part of the state change. But the
    action function should cover all changes an
    action makes to the state, and the dynamic
    function should cover all continuous changes
    that happen in the world model.

Make sure to use the correct 'function_id' for the
     function to be removed or replaced. Stick to
    the format expected for the output.
**Output Format**
Return a JSON of refinements.
```

User Prompt Template

```
The required information is provided below:
The type of error is: {error_type}
The context:
{context}
Context functions:
{functions}

Now generate the most useful refinements for the
    function(s) given:
```

F.2 Graphical learning

Topological Sort Prompt

System Prompt Template:

```
You are an expert at the {name} game. Here is the
    game manual:

{game_manual}

Your task is to determine, based on the game
    manual and your knowledge of the game, how
    game variables relate to each other in the
    current state.
```

User Prompt Template:

```
In the game {game name}, which variables at time t
     would you use to compute the value of {
    variable_name}: {variable_description} at the
    same time t, given values of {temporal_parents
    } at time t-1 and the selected action.
    Candidate variables at time t: {
    candidate_variables}
```

Causally Plausibe Relationship Prompt Template ($w(\mathcal{L}, V, E)$)

System Prompt Template:

```
You are an expert machine learning engineer and
    theme-park game player. Here is the game
    manual:

{gameManual}.

Base your responses on the game manual as well as
    your general knowledge of the world.
```

User Prompt Template:

```
I need help forecasting the variable {
    targetVariable_name}: {
    targetVariable_description} in game {name}
    using a graphical model. I need your help
    deciding whether the design of the graphical
    model dependencies are causally valid based on
     the game manual and your knowledge of the
    game. It is okay if some of the dependencies
    are not mentioned in the manual, as long as
    they are sensible. To predict {
    targetVariable_name} at step t, i use the
    values of {temporal_parents_str} at t-1. This
    is the list of candidate variables: {
    all_variables_str}. Answer True if the
    dependencies are plausible and False otherwise
    . Note that all dependencies need to be at
    least plausible. If any dependencies are not
    plausible, then say False. Always think step
    by step, considering each variable one at a
    time.
```