# Uncovering Systemic and Environment Errors in Autonomous Systems Using Differential Testing

**Yashwanthi Anand[*], Rahil P. Mehta[*], Manish Motwani, Sandhya Saisubramanian**

Oregon State University
{anandy, mehtara, motwanim, sandhya.sai}@oregonstate.edu

## Abstract

Deploying autonomous agents in complex environments requires distinguishing between undesirable behaviors caused by the impreciseness of the agent's reasoning model or its policy (i.e. *systemic agent error*) and those due to inherently unsolvable tasks (*environment error*). We introduce *AIProbe*, a novel black-box differential testing framework to validate autonomous agents under varied and challenging environment configurations. We first describe how AIProbe generates diverse environmental configurations and tasks for testing the agent, by modifying configurable parameters using Latin Hypercube sampling. It then solves each generated task using a search-based planner, independent of the agent. By comparing the agent's performance to the planner's solution, AIProbe identifies whether failures are due to errors in the agent's model or policy, or due to unsolvable task conditions. We then demonstrate its broad applicability to both model-free and model-based agents operating in discrete and continuous domains. Our evaluation across multiple domains shows that AIProbe significantly outperforms state-of-the-art techniques in detecting unique errors, thereby contributing to a reliable deployment of autonomous agents.

## Introduction

Autonomous agents performing complex tasks, such as autonomous driving (Yurtsever et al. 2020) and precision agriculture (Solow, Saisubramanian, and Fern 2025), often exhibit *execution anomalies*: undesirable deviations from the expected behavior, including task failure. Execution anomalies typically arise from two sources: (1) *agent errors*: systemic errors in agent modeling or training, which results in an incorrect policy; and (2) *environment errors*: unfavorable environment configuration that makes task success inherently infeasible, even for an ideal agent.

Agent errors may arise from model defects such as misaligned rewards or incomplete state representation (Saisubramanian, Kamar, and Zilberstein 2020), or training flaws in model-free settings, such as suboptimal choices of learning algorithms or sim-to-real gaps (Ramakrishnan et al. 2020). Environment errors stem from unfavorable layouts, such as poorly placed air vents in warehouses that reduce the efficiency of robot navigation (Simon 2019), or logically infeasible tasks such as painting the wall in blue color and red color at the same time. Diagnosing the root cause of execution anomalies is essential for designing and deploying trustworthy autonomous systems.

Consider a simple example: a mobile robot in a warehouse repeatedly fails to deliver packages from the counter to a storage area. Without a principled investigation, it is unclear whether the failure is due to (1) *agent error*: the robot's policy is flawed because its model did not include information about avoiding slippery tiles, or its path planning algorithm may be suboptimal; or (2) *environment error*: a newly placed pallet may have blocked all feasible paths to the goal, making the task inherently infeasible even for an optimal agent.

As both environments and agents grow more complex, identifying the source of execution anomalies becomes increasingly difficult. In practice, such anomalies are often incorrectly and reflexively attributed solely to agent errors. However, if the root cause lies in the environment configuration, no amount of training or verification will resolve the issue unless the environment is modified. Without systematically testing alternative paths using a model-independent planner or simulating task variants, it is difficult to determine whether the issue lies in the agent or the environment. While prior works have focused on testing for model errors (He et al. 2024; Nayyar, Verma, and Srivastava 2022; Pang, Yuan, and Wang 2022) or using formal verification methods to provide guarantees on the occurrence of anomalies (Corsi, Marchesini, and Farinelli 2021; Shea-Blymyer and Abbas 2024), they do not determine whether an anomaly is due to the agent or the environment, *without* requiring detailed internal access to the agent.

We present *AIProbe*, a black-box technique that applies differential testing to determine whether execution anomalies are due to agent deficiencies or environment-induced infeasibility. Differential testing is a software testing methodology in which the same inputs are run through two or more independent systems (or solvers) and their outputs are compared (McKeeman 1998). If the outputs differ, it indicates error in one of those systems. AIProbe does not require access to the agent's internal model or training data, treating the agent as a black-box system. To uncover configurations that are unfavorable for agent deployment, AIProbe systematically generates diverse environment configurations, by mod-

[*] These authors contributed equally.

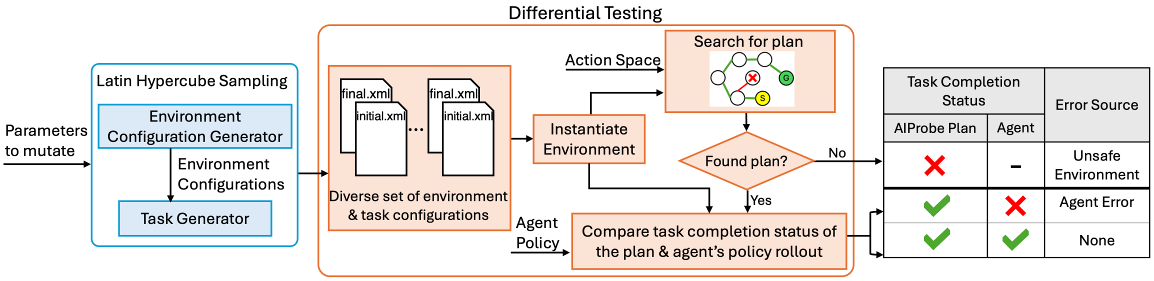


Figure 1: Overview of AIProbe

ifying the configurable parameters of a base environment (e.g., repositioning objects), using Latin Hypercube sampling (LHS) (Loh 1996). For each generated configuration, AIProbe generates a range of tasks and tests whether the agent can complete them.

To determine which tasks are feasible in an environment, AIProbe uses an independent search-based oracle planner that shares the agent's action space but not its model. The planner aims to find a satisficing sequence of actions to solve these tasks. By comparing the agent's behavior with that of the oracle planner, AIProbe determines whether the anomaly stems from the agent's decision-making or from the infeasibility of the task itself. If the agent fails a task that AIProbe can complete safely, this suggests an agent error. If no safe plan exists, the environment itself is unsuitable for task completion. Figure 1 illustrates our approach. AIProbe *does not localize* the modeling or training step that causes the agent error, nor does it reason why the task is unsolvable, and this is by design. We take the pragmatic stance that having a principled approach to perform such a high-level diagnosis is a prerequisite for a more fine-grained error localization.

We empirically evaluate AIProbe in model-free and model-based agents in discrete and continuous domains. Our evaluation on five domains shows that our black-box differential testing method outperforms the state-of-the-art methods in error detection[1].

## Problem Formulation

**Setting** We target agents that optimize reaching a goal in environments modeled as fully observable Markov decision processes (MDPs) (Puterman 1990). An MDP is formally defined by the tuple $M = \langle S, A, T, R, s_o, s_G \rangle$, where $S$ is a set of states; $A$ is the set of all actions that an agent can take; $T : S \times A \rightarrow S$ is the transition function determining the successor state when taking an action $a \in A$ in state $s \in S$; $R : S \times A \rightarrow \mathbb{R}$ specifies the reward associated with taking an action $a$ in the state $s$; $s_0 \in S$ and $s_G \in S$ denote the agent's start and goal states. We focus on both model-free and model-based decision-making settings. In the model-free setting, a reinforcement learning (RL) agent *learns* a policy by exploring the environment. In the model-based setting, the agent *computes* a policy, using its model of the environment, either learned by exploring the environment or prescribed by an expert.

**Problem Statement** Given a set of possible environment configurations $\mathcal{E}$, an agent with policy $\pi$ obtained either through training in a simulator or by solving its MDP $M$, and a baseline oracle planner that computes $\pi_b$ to solve a task $Z$ in an environment $E \in \mathcal{E}$, our goal is to distinguish between anomalies arising from infeasible tasks, where no policy can succeed under the given environment configuration, and those resulting from incorrect $\pi$ due to defects in the agent's model or training practices.

**Assumption 1** (Black-box agent access)**.** *We treat the agent as a **black box**: we can provide it with a task and observe its behavior, but we cannot access its policy, model, or learning process.*

**Assumption 2** (Simulator access)**.** *We assume the agent's behavior and the oracle planner's output can be determined using a simulator.*

The availability of such simulators is a common assumption as most AI systems already use simulators for training (Nayyar, Verma, and Srivastava 2022). In cases where a simulator is not available, a predictive model of the environment's dynamics can be learned from trajectories collected through interaction (Hafner et al. 2020).

**Execution Anomalies** An execution anomaly is any undesirable behavior such as the agent going around in cycles without reaching the goal state, entering a failure terminal state such as a crash, or stepping into unsafe undesirable states such as breaking a vase. Such behaviors may be due to agent errors, or unfavorable environment and task configurations that are fundamentally impossible to achieve. We consider three sources of agent errors: (1) *inaccurate state representation*: missing key features required for decision making; (2) *inaccurate reward function*: does not fully capture the desired and undesired behaviors of the agent; or (3) *both*: inaccurate state representation and reward function. Such defects lead to incorrect policy both in model-based decision-making and in model-free settings since the agent learns a policy directly often by training in a simulator that is prone to these defects. We do not consider anomalies due to external influences such as adversarial attacks.

### Task and Environment Representation

**Task** Each task $Z$ is characterized by an initial state and a final state. A task in an environment is considered to be *solvable* if there exists at least one sequence of actions that can make the agent go from initial state to final state.

[1] https://github.com/ANSWER-OSU/AIProbe

```
<Environment id="" type="">
  <Attribute> ... </Attribute>
  ...
  <Objects>
    <Object id="" type="">
          <Attribute> ... </Attribute>
          ...
    </Object>
    ...
  </Objects>
  <Agents>
    <Agent id=""  type="">
          <Attribute> ... </Attribute>
          ...
    </Agent>
    ...
  </Agents>
</Environment>
```

Figure 2: The XML template for environment configuration using the `<Attribute>` structure shown in Figure 3.

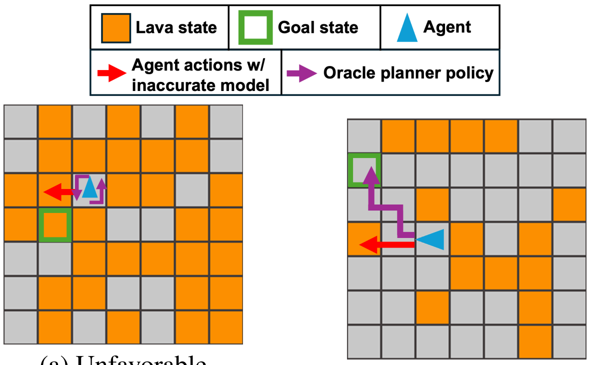
```
1  <Attribute>
2    <Name value=""/>
3    <Description value=""/>
4    <DataType value=""/>
5    <CurrentValue value=""/>
6    <Mutable value=""/>
7    <Constraint Range="" Categories=""
        NumValues="" />
8  </Attribute>
```

Figure 3: XML template for specifying attributes of the environment, objects, and agents.

**Environment** An environment configuration is an instantiation of tunable parameters that define a particular task instance (e.g., obstacle layout, friction coefficients). Diverse configurations can be generated by tuning the attributes of the environment, the attributes of the objects that can exist in that environment, and the attributes of one or more agents that may interact with each other and the objects in that environment. Figure 2 shows the formal representation of an environment configuration in the form of an XML template used by AIProbe. This representation enables generating diverse configurations in a *principled* manner.

Attributes are specified using a generic Attribute template (Figure 3) that specifies the attribute name, a natural language description, data type, and the current value. The `<Mutable>` tag of an attribute can be set to `false` or `true`, indicating whether the attribute's value should remain constant or not, respectively (Line 6). The `<Constraint>` tag (Line 7) describes the allowed values the attribute can take, either a numerical `range` or categorical (`categories`) that can be specified using constants or formula referencing other attributes (e.g. `<Constraint Range=[1,` *grid_size*`]>` for agent coordinates). The `NumValues` tag describes the number of values that the attribute takes, which is useful to represent attributes of array or list data types.

**Definition 1.** *An environment-task configuration is unsafe or unfavorable if the task is unsolvable by any sequence of agent actions in the given environment.*



(a) Unfavorable configuration

(b) Favorable configuration

Figure 4: Lava domain illustration. (a) Impossible to reach the goal while avoiding the lava states (*environment error*). (b) Oracle planner reaches the goal but agent with inaccurate model fails (*agent error*).

**Example** Figure 4 illustrates the lava domain in which an agent must navigate to the goal location (green cell) while avoiding the lava states (Chevalier-Boisvert et al. 2023). This popular environment is modular and configurable, facilitating the generation of multiple configurations. Figure 4a shows an *environment error* as no path to the goal exists without stepping into lava. Figure 4b shows an *agent error* where despite the existence of a safe path, the agent steps into lava due to its model lacking information about undesirability of stepping into lava. When an agent is unable to reach the goal, we want to automatically distinguish between the cases represented in Figures 4a and 4b . As agent architectures grow more complex, especially with learned models, they end up being considered as a black-box model by evaluators. Error detection in a black-box system deployed in large, complex environments is challenging, as we cannot directly inspect the agent's model fidelity or manually analyze all possible environment-task configurations.

## The AIProbe Approach

Our approach operates in three phases to identify the anomaly source (Figure 1): (1) generate diverse environment and task configurations; (2) identify task feasibility using a search-based baseline oracle planner that is independent of the agent; and (3) simulate agent behavior in feasible settings and conduct differential analysis between the observed agent behavior and expected behavior. We describe each phase in more detail below.

### Diverse Environment and Task Configurations

Assessing agent performance across varied environment configurations is critical to identify settings that are safe for agent operation. Exhaustively testing all possible configurations is practically infeasible due to the large number of possible configurations, each characterized by a large state space. To ensure broad and unbiased coverage of the space of possible tasks and environments in which agents can be deployed, AIProbe uses Latin Hypercube

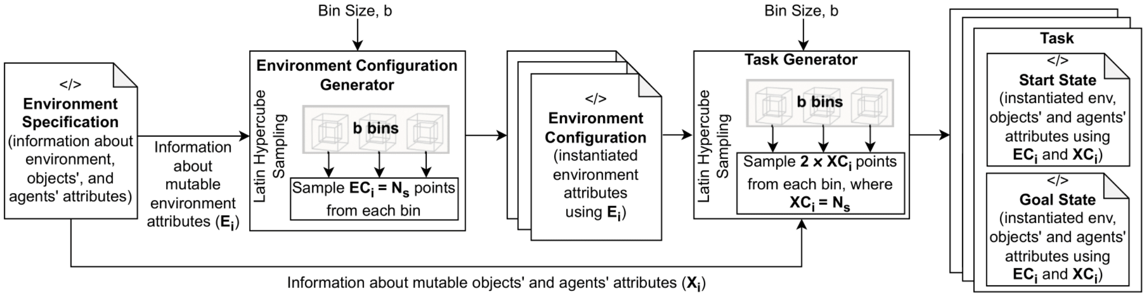

Bin Size, b
Bin Size, b
</>
Environment Specification
(information about environment, objects', and agents' attributes)
Information about mutable environment attributes ($\mathbf{E_i}$)
Environment Configuration Generator
Latin Hypercube Sampling
b bins
Sample $\mathbf{EC_i = N_s}$ points from each bin
</>
Environment Configuration
(instantiated environment attributes using $\mathbf{E_i}$)
Task Generator
Latin Hypercube Sampling
b bins
Sample $\mathbf{2 \times XC_i}$ points from each bin, where $\mathbf{XC_i = N_s}$
Task
</>
Start State
(instantiated env, objects' and agents' attributes using $\mathbf{EC_i}$ and $\mathbf{XC_i}$)
</>
Goal State
(instantiated env, objects' and agents' attributes using $\mathbf{EC_i}$ and $\mathbf{XC_i}$)
Information about mutable objects' and agents' attributes ($\mathbf{X_i}$)

Figure 5: Generating diverse environment and task configurations uniformly at random using Latin Hypercube sampling.

Sampling (LHS) (Loh 1996) which divides each attribute's range into equal intervals and samples within those intervals without overlap. For a D-dimensional environment, where a dimension (or attribute) can be continuous or discrete, AIProbe samples $N_s = b$ points in the space using LHS, where $b$ denotes the bin size. Thus, LHS can efficiently explore high-dimensional input spaces by uniformly sampling each dimension across its range. This is particularly useful for stress-testing agents on edge cases and rare scenarios that may be infrequent in real-world data but still pose safety risks. Figure 5 shows an overview of the process.

**Generating Environment Configurations** Given a manually-created XML file describing the environment, as in Figure 2, AIProbe first extracts all the mutable environment attributes. These form a D-dimensional space, where each attribute represents a dimension. The input to LHS is the number of bins ($b$), the number of dimensions ($D$), the value constraints on each dimension of the environment ($E_i$), and type (continuous or categorical). The output is a set of $EC_i$ samples from each bin, where $|EC_i| = N_s$ and each sample is a point in the $D$-dimensional space.

For continuous attributes, the range of $E_i$ is divided into $b$ equal bins or strata, and $EC_i$ points are sampled uniformly from each stratum. For categorical dimensions with $k$ categories, the algorithm maps the categories to the range $[0, 1]$ by partitioning the interval into $k$ equal segments, stratifies it into $b$ equal strata, samples $EC_i$ values uniformly from each, and maps the samples back to the original categories using inverse mapping. For example, in the lava domain, environment has two attributes: grid size $n$ and number of lava tiles in it denoted by $l$. The values of $n$ and $l$ are in the ranges $[3, 50]$ and $[0, n^2]$, respectively. AIProbe generates $N_s = 100$ diverse configurations by uniformly sampling across both dimensions. Each sampled configuration is saved as an XML file with the initialized attribute values.

**Generating Tasks** Given a generated environment configuration, AIProbe generates tasks defined by start and goal states by varying agent and object attributes ($X_i$), while keeping environment attributes ($E_i$) fixed. LHS is used to sample values across mutable agent and object attributes. For each mutable attribute, AIProbe generates two samples per bin: one for the initial state and one for the goal. The samples are denoted by denoted by $XC_i$. For example, in the lava domain with grid size 5 and 10 lava tiles, mutable dimensions include each lava tile's coordinates and the agent's start and goal positions, ensuring task diversity. Since tasks can be favorable or unfavorable, AIProbe first verifies task feasibility using an agent-agnostic algorithm.

## Search-based Planning as a Baseline Oracle

A natural way to determine if a task is solvable is to formulate it as a search problem to find a satisficing solution, independent of the agent's transition function or the reward function. To achieve this, *any* search algorithm can be used as a Baseline Oracle in practice. However, *accurate* diagnosis with differential testing requires the search to be *complete* even if not optimal. While the search is model-agnostic, it still requires knowledge of action outcomes. Instead of assuming access to environment dynamics, we assume the planner can execute an action sequence in the simulator and observe the resulting state.

**Assumption 3.** *The baseline oracle planner does not have knowledge of environment dynamics. It only observes the final state reached by applying a sequence of actions from a current state.*

To efficiently search for a satisficing plan in problems with large search spaces, we use a depth-limited heuristic search (Korf 1985) with backtracking that operates over a directed graph of states (nodes) and actions (edges), constructed on-the-fly. Depth-limited search bounds the exploration depth and is a well-established approach for managing large search spaces, which makes it suitable for stress-testing agents when exhaustive search is infeasible.

**Depth-limited heuristic search** The input to our planner is a task ($Z = (s_0, s_G)$), with an initial state ($s_0$) and a goal state ($s_G$), agent's action space ($A$), and the parameters used to measure the search heuristic: number of bins ($b$), number of plans per iteration ($N$), and maximum search depth ($D$). The output is a satisficing policy $\pi_b$, if one is found. Algorithm 1 presents the pseudocode.

*Heuristic Estimation:* To estimate the number of actions required to reach the goal state from a given state (i.e. plan length), we use the L1-norm distance between the bin indices of the normalized attributes of the current and goal states. For example, in the Lava domain on a 32 × 32 grid,

**Algorithm 1** Heuristic-Guided Search

```
    Input: Number of bins $b$; Task $Z = (s_0, s_G)$; Action space $A$; Number of paths to explore in one iteration $N$; Maximum search depth $L$
 1: $Visited \leftarrow \emptyset$                              ▷ set of visited states
 2: $\pi_b \leftarrow []$                                       ▷ Baseline Plan
 3: $h \leftarrow$ HEURISTIC($s_0$, $s_G$, $b$)
 4: return SEARCH($s_0, s_G, A, h, N, L, 0, Visited, \pi_b$)
 5: function HEURISTIC($S_{curr}, S_G, b$)
 6:     $\phi_{curr} \leftarrow$ BIN_NORMALIZED_ATTRIBUTE($S_{curr}, b$)
 7:     $\phi_{goal} \leftarrow$ BIN_NORMALIZED_ATTRIBUTE($S_G, b$)
 8:     return $\|\phi_{curr} - \phi_{goal}\|_1$
 9: function SEARCH($S_{curr}$, $S_G$, $A$, $h$, $N$, $L$, $depth$, $Visited$, $\pi_b$)
10:     if $depth > L$ then                                     ▷ reached max depth
11:         return (False, $\pi_b$)
12:     if TERMINAL($S_{curr}$) = True then
13:         if $\pi_b = []$ then
14:             return (False, $\pi_b$)                         ▷ terminal start state
15:         else
16:             $(S_{prev}, \pi_b^{prev}) \leftarrow$ BACKTRACK($S_{curr}, \pi_b$)
17:             $h \leftarrow$ HEURISTIC($S_{prev}$, $S_G$, $b$)
18:             return SEARCH($S_{prev}$, $S_G$, $A$, $h$, $N$, $L$, $depth + 1$, $Visited$, $\pi_b^{prev}$)
19:     if $(S_{curr}, \pi_b) \in Visited$ then
20:         return (False, $\pi_b$)      ▷ Avoid revisiting states to prevent infinite loops
21:     $Visited \leftarrow Visited \cup \{(S_{curr}, \pi_b)\}$
22:     if $S_{curr} = S_G$ then
23:         return (True, $\pi_b$)                              ▷ Valid plan found
24:     for $i = 1$ to $N$ do
25:         $\pi_b$ = Sample $h$ actions from $A$
26:         $S_{next} \leftarrow$ TRANSITION($S_{curr}, \pi_b$)
27:         $h \leftarrow$ HEURISTIC($S_{next}$, $S_G$, $b$)
28:         $result \leftarrow$ SEARCH($S_{next}$, $S_G$, $A$, $h$, $N$, $L$, $depth + 1$, $Visited$, $\pi_b$)
29:         if $result[0]$ = True then                          ▷ Valid plan found
30:             return $result$
31:     return (False, $\pi_b$)
```

the agent's position $(x, y)$ is a mutable attribute. Suppose a task specifies a start state as $(x = 32, y = 23)$ and goal state as $(x = 13, y = 2)$. With 100 bins, the heuristic is $|bin(x{=}32) - bin(x{=}13)| + |bin(y{=}23) - bin(y{=}2)| = |100 - 41| + |72 - 7| = 124$. Using this heuristic value, the algorithm generates $N$ plans, each with 124 actions.

*Depth-limited recursive search:* The planner uses a recursive search (Lines 9–31) that terminates when it (1) finds a valid plan (Lines 22–23 and 29–30), (2) hits the maximum depth (Line 10-11), or (3) reaches a failure terminal state determined by a domain-specific simulator. If an unfavorable state is reached during the search, with the current (partial) plan, then the algorithm backtracks to the previous state with unexplored paths and explores different paths (Lines 12–18). Revisited states are skipped (Lines 19–21). In each iteration, the algorithm samples and simulates $N$ plans, and continues recursively (Lines 24-28), until a valid solution is found or all search paths are exhausted. While some generated tasks may be infeasible, the majority tend to be solvable, particularly in less constrained environments where the agent has sufficient freedom to move.

The planner is *complete* up to depth $L$. That is, if a solution exists within this horizon, backtracking ensures it is found. If the plan reaches an unfavorable state, the algorithm backtracks to the most recent branching point and explores a different action sequence. For tasks where the algorithm fails to find a solution, AIProbe performs breadth first search (BFS) from the initial state to verify that the task is indeed impossible.

**Error Attribution** Once feasible tasks are identified, the agent is evaluated on them and its policies are compared to the planner's for task completion. If the planner succeeds to solve the task but the agent fails, the error is attributed to the agent. If the planner fails to solve the task, the environment-task setting is flagged as unsafe for deployment. In cases where an approximate planner is unable to solve the task, we take a conservative approach and label the environment-task setting as unfavorable for the agent. These cases can be further validated through human oversight.

## Evaluation

We evaluate AIProbe on open-source discrete and continuous RL domains, including single and multi-agent settings. All agents are trained using PPO (Schulman et al. 2017).

**Baselines** We compare AIProbe, with 10 and 20 seeds, against two state-of-the-art fuzz testing approaches: MDPFuzz (Pang, Yuan, and Wang 2022) and CureFuzz (He et al. 2024). We also perform ablation studies by comparing: (1) LHS-based generation with large language model (LLM)-based generated configurations, and (2) heuristic planner against Breadth First Search (BFS).

**Evaluation Metrics** We use the following metrics in our experiments: (1) number of *execution anomalies* identified, (2) number of *environment errors* from infeasible tasks, (3) number of *agent errors* from agent's model or training flaws, and (4) *state coverage*, reflecting the fraction of environment-task configurations tested, inspired by the *code coverage* metric used to demonstrate the effectiveness of automated testing techniques (Motwani and Brun 2019).

### Domains

We evaluate AIProbe across five domains representing a mix of discrete/continuous domains and model-free/model-based agents: (1) ACAS Xu (Julian et al. 2016), (2) Cooperative Navigation (Lowe et al. 2017), (3) Bipedal Walker (Brockman et al. 2016), (4) Flappy Bird (Tasfi 2016), and (5) Lava (Chevalier-Boisvert et al. 2023).

In each domain, we test the *base model*, which is the publicly available pre-trained model (He et al. 2024), along with three model variants reflecting common agent modeling errors: (1) incomplete state representation, where the agent is reasoning at an abstract level that does not fully capture the details for successful task completion; (2) incorrect reward function, where the reward function is under-specified and does not capture the full range of desirable and undesirable behaviors; (3) a combination of both (1) and (2).

**ACAS Xu** This continuous-state, discrete-action domain simulates an aircraft collision avoidance system with two aircrafts: ownship (agent) and intruder. Each state is denoted by $\langle \rho, \theta, \psi, v_{own}, v_{int} \rangle$, where $\rho$ is the relative distance, $\theta$ the relative angle, $\psi$ the relative heading angle, $v_{own}$ (m/s) is the ownship speed, and $v_{int}$ (m/s) is the intruder speed. Actions include Clear-of-Conflict, weak/strong left and weak/strong right. The agent receives a step reward of $(\gamma + \rho/60261.0)$, with a $-100$ penalty for close proximity. We introduce model errors by (1) omitting $\rho$ from the state, which impairs the agent's ability to reason about the relative distance; (2) under-specifying reward $(\gamma + \rho/1e6.0)$ that disproportionately emphasizes the distance component; and (3) a combination of (1) and (2).

**Co-operative Navigation (Coop-Navi)** This multi-agent continuous domain from Gymnasium's PettingZoo involves three agents coordinating to reach three distinct landmarks while avoiding collisions (Lowe et al. 2017). Each agent chooses from five discrete actions: move up/down/left/right or no-op. The state includes each agent's velocity $v_s$, position $p_s$, other agents' positions $p_o$, and a 2-bit communication vector $c$. Agents receive a shared global reward based on the sum of distances to the nearest landmark and a local penalty of $-1$ per collision. Episodes end after five collision or successful task completion. We introduce model errors by (1) omitting $p_o$ from the state, making it more difficult to coordinate; (2) removing collision penalties, which may lead to collisions and (3) a combination of (1) and (2).

**Bipedal Walker** In this continuous, non-deterministic control domain, modeled using the Box2D engine, a four-joint bipedal robot must walk across complex terrains with grass, pits, stairs and stumps, without falling (Brockman et al. 2016). We evaluate using the *hardcore* setting, with 2000 timesteps as the episode horizon and a maximum reward of 300. The state includes body and joint dynamics, leg contact, and 10 LiDAR readings. The agent incurs $-100$ penalty for falling and $-5$ for poor posture. Model errors are introduced by (1) omitting LiDAR readings from the state, affecting the agent's ability to reason about upcoming terrain; (2) removing posture penalty; and (3) both (1) and (2).

**Flappy Bird** This RL domain involves navigating a bird (agent) through pipe gaps to maximize survival time (Tasfi 2016). Each state is denoted by $\langle y_b, v_b, d_{p_1}, y_{p_1t}, y_{p_1b}, d_{p_2}, y_{p_2t}, y_{p_2b} \rangle$, denoting the bird's position and velocity, distances to pipes, and pipe edge positions. The agent earns $+0.5$ per step, $+1$ per pipe passed, and $-1$ on crash. Model errors are introduced by (1) omitting pipe edge positions from the state, (2) removing reward to pass a pipe due to which the agent may fail to learn to time its flaps; and (3) doing both (1) and (2).

**Lava** We modify the Lava domain from MiniGrid, where the agent must reach a goal while avoiding lava, aiming for minimal steps to goal (Figure 4). The agent uses a model of the environment for planning, allowing us to test AIProbe in model-based settings. Each state is denoted by $\langle x, y, d, l \rangle$, denoting agent position, orientation, and lava presence. Actions include turning left/right and moving forward. Rewards are $+100$ for reaching the goal, $-10$ for lava (terminal), and 0 otherwise. Model errors are introduced by (1) omitting lava information from the state; (2) removing the penalty for entering a lava tile; and (3) both (1) and (2).

## Results and Discussion

**Discovering execution anomalies** Table 1 compares the number of execution anomalies discovered by various techniques. AIProbe consistently finds more execution anomalies than baselines in all domains except Lava, where CureFuzz outperforms due to its 12 hour *total runtime* while AIProbe has a time limit of 1.5 hours *per configuration*. As a result, CureFuzz generates more than $10,000$ configurations. AIProbe also achieves higher state coverage in all domains but Bipedal, where CureFuzz benefits from its extended runtime to produce more configurations. We use a 1.5 hour per-configuration timeout to balance effectiveness and practicality. While longer timeouts (e.g. 12 hrs) improve coverage, they are costly and less feasible for large-scale or iterative testing. AIProbe's performance remains stable with more seeds (10 vs. 20), highlighting its reliability.

**Agent vs. Environment Errors** We use AIProbe to generate $10,000$ environment-task configurations per seed, and execute heuristic search planner with a timeout of 1.5-hour per configuration to detect if the task is feasible. Infeasible configurations are counted as environment errors. Agents are evaluated on feasible configurations and failures are counted as agent errors. Table 2 reports average errors across multiple seeds for different model variants. Since AIProbe aims for *diversity* rather than a fixed ratio of feasible/infeasible tasks when generating configurations, most configurations, especially in ACAS Xu and Coop Navi, are feasible. Defective models yield more agent errors, demonstrating AIProbe's ability to identify model-specific failures. In ACAS Xu, defective models sometimes produce fewer errors (crashes) by flying faster. Flappy Bird's large state space causes timeouts in $\sim$7700 cases. Bipedal Walker is excluded from this analysis, as its non-determinism prevents reliable plan generation (Kuter et al. 2008), though execution anomalies can still be measured.

**Generating configurations using LLMs vs. using LHS** We evaluate GPT-4o (OpenAI 2024) for generating $10,000$ environment-task configurations using prompts that describe the domain and the agent. GPT-4o was prompted to create configurations that are *likely* to induce failures, aiming to probe the limits of agent behavior. Feasibility was verified using AIProbe's heuristic search planner. As shown in Table 3, LLM-generated configurations complement those generated by AIProbe, uncovering additional anomalies in some domain-model combinations. While LLM can effectively target specific vulnerabilities when agent information is available, AIProbe offers a better trade-off between broad state coverage and anomaly detection. Integrating both approaches is a promising direction for future work.

| Domain | Technique | Execution Anomalies Detected | State Coverage |
|---|---|---|---|
| ACAS Xu | MDPFuzz | $9.0 \pm 0.9$ | $2.0e^{-6} \pm 4.0e^{-8}$ |
| | CureFuzz | $17.0 \pm 1.5$ | $6.0e^{-6} \pm 1.0e^{-6}$ |
| | **AIProbe (10 seeds)** | $\mathbf{53.7 \pm 5.8}$ | $\mathbf{8.1e^{-3} \pm 4.8e^{-4}}$ |
| | **AIProbe (20 seeds)** | $\mathbf{54.8 \pm 5.1}$ | $\mathbf{8.2e^{-3} \pm 6.8e^{-4}}$ |
| Coop Navi | MDPFuzz | $52.4 \pm 11.7$ | $5.0e^{-19} \pm 6.0e^{-20}$ |
| | CureFuzz | $85.3 \pm 7.3$ | $1.0e^{-18} \pm 5.0e^{-19}$ |
| | **AIProbe (10 seeds)** | $\mathbf{139.0 \pm 12.0}$ | $\mathbf{9.9e^{-9} \pm 1.1e^{-10}}$ |
| | **AIProbe (20 seeds)** | $\mathbf{138.4 \pm 10.8}$ | $\mathbf{9.9e^{-9} \pm 1.4e^{-10}}$ |
| BipedalWalker | MDPFuzz | $126.0 \pm 31.8$ | $6.5e^{-2} \pm 2.0e^{-3}$ |
| | CureFuzz | $658.0 \pm 98.3$ | $\mathbf{4.2e^{-1} \pm 2.0e^{-2}}$ |
| | **AIProbe (10 seeds)** | $\mathbf{7880.0 \pm 211.4}$ | $3.0e^{-3} \pm 1.0e^{-4}$ |
| | **AIProbe (20 seeds)** | $\mathbf{7890.0 \pm 166.8}$ | $3.0e^{-3} \pm 1.0e^{-4}$ |
| Flappy Bird | MDPFuzz | $3125.0 \pm 1334.9$ | $45.9 \pm 17.9$ |
| | CureFuzz | $1376.8 \pm 41.1$ | $22.3 \pm 0.5$ |
| | **AIProbe (10 seeds)** | $\mathbf{7277.0 \pm 135.6}$ | $\mathbf{99.9 \pm 0.3}$ |
| | **AIProbe (20 seeds)** | $\mathbf{7188.1 \pm 240.9}$ | $\mathbf{99.9 \pm 0.2}$ |
| Lava | MDPFuzz | $2160.2 \pm 139.7$ | $3.2e^{-9} \pm 1.5e^{-10}$ |
| | CureFuzz | $\mathbf{213585.4 \pm 106793.2}$ | $9.2e^{-7} \pm 1.1e^{-7}$ |
| | **AIProbe (10 seeds)** | $6775.5 \pm 319.7$ | $\mathbf{8.9e^{-5} \pm 2.4e^{-5}}$ |
| | **AIProbe (20 seeds)** | $6704.8 \pm 303.9$ | $\mathbf{8.9e^{-5} \pm 2.1e^{-5}}$ |

Table 1: Execution anomalies and state coverage across domains, averaged over base models. Best values are in **bold**.

| | | **Env-Task Configs** | | **#Agent Errors** | | | |
|---|---|---|---|---|---|---|---|
| **Domain** | **#Seeds** | **# Feasible** | **# Infeasible (Env. error)** | **Base Model** | **Inacc. State** | **Inacc. Reward** | **Both** |
| ACAS Xu | 10 | $9974.8 \pm 2.3$ | $25.2 \pm 2.3$ | $262.7 \pm 33.1$ | $119.9 \pm 18.8$ | $124.8 \pm 17.2$ | $95.9 \pm 11.4$ |
| | 20 | $9975.1 \pm 4.3$ | $24.9 \pm 4.3$ | $268.5 \pm 34.1$ | $122.5 \pm 25.1$ | $117.5 \pm 25.3$ | $90.6 \pm 15.0$ |
| Coop Navi | 10 | $9938.7 \pm 1.3$ | $47.8 \pm 13.2$ | $98.2 \pm 11.2$ | $4687.6 \pm 279.5$ | $3372.9 \pm 238.8$ | $8997.5 \pm 655.2$ |
| | 20 | $9939.3 \pm 1.7$ | $46.4 \pm 14.5$ | $98.0 \pm 12.0$ | $4569.2 \pm 339.5$ | $3846.8 \pm 363.9$ | $9157.2 \pm 634.2$ |
| Flappy Bird | 10 | $1375.4 \pm 148.4$ | $932.8 \pm 142.9$ | $268.3 \pm 71.2$ | $832.9 \pm 317.8$ | $776.0 \pm 209.4$ | $675.1 \pm 199.5$ |
| | 20 | $1406.5 \pm 160.2$ | $885.5 \pm 157.7$ | $262.1 \pm 68.6$ | $841.9 \pm 232.4$ | $703.3 \pm 189.5$ | $645.8 \pm 189.6$ |
| Lava | 10 | $3185.0 \pm 334.1$ | $6815.0 \pm 334.1$ | $0.0 \pm 0.0$ | $2137.8 \pm 254.4$ | $2137.8 \pm 254.4$ | $2137.8 \pm 254.4$ |
| | 20 | $3273.5 \pm 317.1$ | $6726.5 \pm 317.1$ | $0.0 \pm 0.0$ | $2233.5 \pm 244.9$ | $2233.5 \pm 244.9$ | $2233.5 \pm 244.9$ |

Table 2: Average agent and environment errors identified by AIProbe, across domains and models. "Both" refers to a model with known defects in state representation and reward function.

**AIProbe's planner vs. BFS** We compare AIProbe's heuristic search planner's efficiency to BFS based on the number of feasible and infeasible tasks identified, and timeouts when using 30-minute per-task cutoff. As shown in Table 4, AIProbe matches or outperforms BFS with notably fewer timeouts. Bipedal domain in excluded due to non-determinism. While any planner can serve as the baseline, the results underscore the importance of fast planning for reliable error attribution in large settings.

## Related Work

**Automated testing of autonomous systems** Manual testing of autonomous systems across all deployment scenarios is infeasible, motivating automated approaches (Karimoddini et al. 2022). Fuzz testing is a software testing technique that involves testing the system on a large number of random inputs to uncover bugs, crashes, security vulnerabilities, or other unexpected behavior. Prior fuzz-testing methods such as CureFuzz (He et al. 2024) and MDPFuzz (Pang, Yuan, and Wang 2022), generate diverse input scenarios using techniques such as curiosity-driven exploration to uncover edge cases that lead to agent failure. Search-based methods (Tappler et al. 2022, 2024) create adversarial environment-task configurations but require access to the agent's internal model, limiting applicability to black-box systems. Differential testing has also been used to evaluate model updates to the agent (Nayyar, Verma, and Srivastava 2022). These approaches primarily focus on *detecting model-specific failures* without explicitly addressing the feasibility of tasks within the environment itself. Further, many of them require *access* to the agent's internal model. In contrast, AIProbe can detect both environment and agent errors in black-box systems.

| | | **Execution Anomalies Detected** | | | | |
|---|---|---|---|---|---|---|
| **Domain** | **Technique** | **Base** | **Inacc. State** | **Inacc. Reward** | **Both** | **State Coverage** |
| ACAS Xu | LLM | $\mathbf{4890.2 \pm 237.6}$ | $\mathbf{5545.0 \pm 0.0}$ | $\mathbf{5578.0 \pm 0.0}$ | $\mathbf{5545.0 \pm 0.0}$ | $1.0e^{-4} \pm 0.0$ |
| | **AIProbe (10 seeds)** | $483.0 \pm 53.3$ | $139.9 \pm 9.4$ | $138.7 \pm 12.2$ | $115.5 \pm 12.05$ | $\mathbf{8.1e^{-3} \pm 4.8e^{-4}}$ |
| | **AIProbe (20 seeds)** | $493.4 \pm 45.9$ | $142.5 \pm 9.59$ | $137.4 \pm 13.6$ | $115.45 \pm 10.63$ | $\mathbf{8.2e^{-3} \pm 6.8e^{-4}}$ |
| Coop Navi | LLM | $137.0 \pm 0.0$ | $4603.0 \pm 0.0$ | $3882.0 \pm 0.0$ | $9150.0 \pm 0.0$ | $9.9e^{-9} \pm 0.0$ |
| | **AIProbe (10 seeds)** | $\mathbf{139.0 \pm 12.0}$ | $\mathbf{4612.6 \pm 52.3}$ | $3879.7 \pm 46.5$ | $\mathbf{9177.1 \pm 36.4}$ | $\mathbf{9.9e^{-9} \pm 1.1e^{-10}}$ |
| | **AIProbe (20 seeds)** | $\mathbf{138.6 \pm 10.9}$ | $\mathbf{4615.6 \pm 1439.6}$ | $\mathbf{3893.2 \pm 1229.2}$ | $\mathbf{9203.6 \pm 2864.02}$ | $\mathbf{9.9e^{-9} \pm 1.4e^{-10}}$ |
| Flappy Bird | LLM | $\mathbf{7885.0 \pm 0.0}$ | $9350.0 \pm 0.0$ | $9578.0 \pm 0.0$ | $8294.0 \pm 0.0$ | $62.0 \pm 0.0$ |
| | **AIProbe (10 seeds)** | $1237.6 \pm 156.8$ | $\mathbf{9355.2 \pm 80.05}$ | $\mathbf{9605.0 \pm 97.4}$ | $\mathbf{8307.8 \pm 101.17}$ | $\mathbf{99.9 \pm 0.3}$ |
| | **AIProbe (20 seeds)** | $1194.6 \pm 166.8$ | $9334.85 \pm 93.4$ | $\mathbf{9582.1 \pm 110.1}$ | $\mathbf{8285.65 \pm 80.8}$ | $\mathbf{99.9 \pm 0.2}$ |
| Lava | LLM | $\mathbf{8793.0 \pm 0.0}$ | $\mathbf{9467.0 \pm 0.0}$ | $\mathbf{9467.0 \pm 0.0}$ | $\mathbf{9467.0 \pm 0.0}$ | $\mathbf{9.5e^{-5} \pm 0.0}$ |
| | **AIProbe (10 seeds)** | $6815.0 \pm 334.1$ | $6815.0 \pm 334.1$ | $6815.0 \pm 334.1$ | $6815.0 \pm 334.1$ | $8.9e^{-5} \pm 2.4e^{-5}$ |
| | **AIProbe (20 seeds)** | $6704.8 \pm 303.9$ | $6726.5 \pm 317.1$ | $6726.5 \pm 317.1$ | $6726.5 \pm 317.1$ | $8.9e^{-5} \pm 2.1e^{-5}$ |
| BipedalWalker | LLM | $\mathbf{8101.2 \pm 0}$ | $10000 \pm 0$ | $10000 \pm 0$ | $10000 \pm 0$ | $2.4e^{-3} \pm 0.0$ |
| | **AIProbe (10 seeds)** | $7880 \pm 211.4$ | $\mathbf{10000 \pm 0}$ | $\mathbf{10000 \pm 0}$ | $\mathbf{10000 \pm 0}$ | $\mathbf{3.0e^{-3} \pm 1.0e^{-4}}$ |
| | **AIProbe (20 seeds)** | $7890 \pm 166.8$ | $\mathbf{10000 \pm 0}$ | $\mathbf{10000 \pm 0}$ | $\mathbf{10000 \pm 0}$ | $\mathbf{3.0e^{-3} \pm 1.0e^{-4}}$ |

Table 3: Average anomalies and state coverage using AIProbe and LLM-generated configurations.

| **Domain** | **Baseline Planner** | **#Feasible** | **#Infeasible** | **#Timeout** |
|---|---|---|---|---|
| ACAS Xu | BFS | 9975 | 3 | 22 |
| | **AIProbe search** | **9977** | **22** | **1** |
| Coop Navi | BFS | 9872 | 0 | 128 |
| | **AIProbe search** | **9939** | **34** | **27** |
| Flappy Bird | BFS | 1411 | 20 | 8569 |
| | **AIProbe search** | **1472** | **817** | **7711** |
| Lava | BFS | 3062 | 6938 | 0 |
| | **AIProbe search** | **3062** | **6938** | **0** |

Table 4: Number of feasible, infeasible and timed-out (30-min cutoff) tasks identified by BFS and our heuristic search.

**Model cards** To improve the transparency of machine learning systems, model cards have been introduced to document the training and evaluation settings (Mitchell et al. 2019; Crisan et al. 2022). Our testing framework enables principled, exhaustive testing of autonomous agents, providing the data to create model cards for autonomous agents.

## Conclusion and Future Work

We present AIProbe, a novel framework to assess both agent reliability and environment suitability. Our results in discrete and continuous domains show that AIProbe outperforms the state-of-the-art methods in detecting substantially more execution anomalies and attributing them to agent or environment errors, while achieving high state space coverage. AIProbe inherits some limitations of search-based planners: state space explosion in high-dimensional settings and inability to handle stochastic environments where the same plan can lead to success or failure outcome. In the future, we plan to address these limitations and broaden AIProbe's applicability to more complex real-world domains where high-fidelity simulators may be unavailable.

## Acknowledgments

This work was supported in part by NSF award 2416459.

## References

Brockman, G.; Cheung, V.; Pettersson, L.; Schneider, J.; Schulman, J.; Tang, J.; and Zaremba, W. 2016. OpenAI Gym. arXiv:1606.01540.

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

Corsi, D.; Marchesini, E.; and Farinelli, A. 2021. Formal verification of neural networks for safety-critical tasks in deep reinforcement learning. In *PMLR*.

Crisan, A.; Drouhard, M.; Vig, J.; and Rajani, N. 2022. Interactive model cards: A human-centered approach to model documentation. In *2022 ACM Conference on Fairness, Accountability, and Transparency*, 427–439.

Hafner, D.; Lillicrap, T. P.; Ba, J.; and Norouzi, M. 2020. Dream to Control: Learning Behaviors by Latent Imagination. In *ICLR*.

He, J.; Yang, Z.; Shi, J.; Yang, C.; Kim, K.; Xu, B.; Zhou, X.; and Lo, D. 2024. Curiosity-Driven Testing for Sequential Decision-Making Process. In *ICSE*.

Julian, K. D.; Lopez, J.; Brush, J. S.; Owen, M. P.; and Kochenderfer, M. J. 2016. Policy compression for aircraft collision avoidance systems. In *IEEE/AIAA 35th DASC*.

Karimoddini, A.; Khan, M. A.; Gebreyohannes, S.; Heiges, M.; Trewhitt, E.; and Homaifar, A. 2022. Automatic test and evaluation of autonomous systems. *IEEE Access*, 10: 72227–72238.

Korf, R. E. 1985. Depth-First Iterative-Deepening: An Optimal Admissible Tree Search. *Artificial Intelligence*.

Kuter, U.; Nau, D. S.; Reisner, E.; and Goldman, R. P. 2008. Using Classical Planners to Solve Nondeterministic Planning Problems. In *ICAPS 2008*.

Loh, W.-L. 1996. On Latin hypercube sampling. *The annals of statistics*, 24(5): 2058–2080.

Lowe, R.; Wu, Y.; Tamar, A.; Harb, J.; Abbeel, P.; and Mordatch, I. 2017. Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments. *NeurIPS*.

McKeeman, W. M. 1998. Differential testing for software. *Digital Technical Journal*, 10(1): 100–107.

Mitchell, M.; Wu, S.; Zaldivar, A.; Barnes, P.; Vasserman, L.; Hutchinson, B.; Spitzer, E.; Raji, I. D.; and Gebru, T. 2019. Model cards for model reporting. In *conference on fairness, accountability, and transparency*, 220–229.

Motwani, M.; and Brun, Y. 2019. Automatically generating precise Oracles from structured natural language specifications. In *ICSE*.

Nayyar, R. K.; Verma, P.; and Srivastava, S. 2022. Differential Assessment of Black-Box AI Agents. *AAAI*.

OpenAI. 2024. GPT-4o System Card. arXiv:2410.21276.

Pang, Q.; Yuan, Y.; and Wang, S. 2022. MDPFuzz: testing models solving Markov decision processes. In *31st ACM SIGSOFT International Symposium on Software Testing and Analysis*.

Puterman, M. L. 1990. Markov decision processes. *Handbooks in operations research and management science*, 2: 331–434.

Ramakrishnan, R.; Kamar, E.; Dey, D.; Horvitz, E.; and Shah, J. 2020. Blind spot detection for safe sim-to-real transfer. *JAIR*.

Saisubramanian, S.; Kamar, E.; and Zilberstein, S. 2020. A Multi-Objective Approach to Mitigate Negative Side Effects. In *IJCAI*.

Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A.; and Klimov, O. 2017. Proximal Policy Optimization Algorithms. ArXiv:1707.06347 [cs].

Shea-Blymyer, C.; and Abbas, H. 2024. Formal Ethical Obligations in Reinforcement Learning Agents: Verification and Policy Updates. In *AAAI/ACM AIES*.

Simon, M. 2019. Inside the Amazon Warehouse Where Humans and Machines Become One. https://www.wired.com/story/amazon-warehouse-robots/.

Solow, W.; Saisubramanian, S.; and Fern, A. 2025. WOFOSTGym: A Crop Simulator for Learning Annual and Perennial Crop Management Strategies. *Reinforcement Learning Journal*.

Tappler, M.; Córdoba, F. C.; Aichernig, B.; and Könighofer, B. 2022. Search-Based Testing of Reinforcement Learning. In *ECAI*.

Tappler, M.; Pferscher, A.; Aichernig, B. K.; and Könighofer, B. 2024. Learning and repair of deep reinforcement learning policies from fuzz-testing data. In *ICSE*.

Tasfi, N. 2016. PyGame Learning Environment. https://github.com/ntasfi/PyGame-Learning-Environment.

Yurtsever, E.; Lambert, J.; Carballo, A.; and Takeda, K. 2020. A survey of autonomous driving: Common practices and emerging technologies. *IEEE Access*, 8: 58443–58469.