ProbGuard: Proactive Runtime Monitoring for LLM Agent Safety via Probabilistic Prediction

Haoyu Wang
Singapore Management University
Singapore
haoyu.wang.2024@phdcs.smu.edu.sg

Christopher M. Poskitt
Singapore Management University
Singapore
cposkitt@smu.edu.sg

Jiali Wei
Xi'an Jiaotong University
Xi'an, China
weijiali1119@stu.xjtu.edu.cn

Jun Sun
Singapore Management University
Singapore
junsun@smu.edu.sg

Abstract

Large Language Model (LLM) agents increasingly operate across domains such as robotics, virtual assistants, and web automation. However, their stochastic decision-making introduces safety risks that are difficult to anticipate during execution. Existing runtime monitoring frameworks, such as AgentSpec, primarily rely on reactive safety rules that detect violations only when unsafe behavior is imminent or has already occurred, limiting their ability to handle long-horizon dependencies. We present ProbGuard, a proactive runtime monitoring framework for LLM agents that anticipates safety violations through probabilistic risk prediction. ProbGuard abstracts agent executions into symbolic states and learns a Discrete-Time Markov Chain (DTMC) from execution traces to model behavioral dynamics. At runtime, the monitor estimates the probability that execution will remain safe from the current state, and triggers an intervention when this probability falls below a user-defined threshold. To improve robustness, ProbGuard incorporates semantic validity constraints in the abstraction and admits a PAC-style analysis that characterizes the sample complexity required to certify the learned model under standard assumptions. We evaluate ProbGuard in two safety-critical domains: autonomous driving and embodied household agents. Across evaluated scenarios, ProbGuard consistently predicts traffic law violations and collisions in advance, with warnings up to 15.84 seconds at a threshold yielding no false alarms, and up to 38.66 seconds at stricter thresholds. In embodied agent tasks, ProbGuard's re-prompting intervention mode reduces unsafe behavior by 65.37% relative to the unmonitored baseline while retaining 80.4% of the baseline task completion; a stricter halting configuration reduces unsafe behavior by 93.60% at a larger cost in completion. ProbGuard is implemented as an extensible open-source runtime monitor integrated with the LangChain agent framework and introduces minimal runtime overhead.

CCS Concepts

• **Software and its engineering**; • **Theory of computation** → **Logic and verification**;

ASE '26, Munich, Germany

ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3837413

Keywords

LLM Agents, Runtime Monitoring, Runtime Verification, Probabilistic Model Checking, Discrete-Time Markov Chains, Agent Safety

ACM Reference Format:
Haoyu Wang, Christopher M. Poskitt, Jiali Wei, and Jun Sun. 2026. ProbGuard: Proactive Runtime Monitoring for LLM Agent Safety via Probabilistic Prediction. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3832783.3837413

1 Introduction

Large Language Models (LLMs) are increasingly used as the foundation for building autonomous agents. These agents function as programmable entities operating across domains ranging from code generation and productivity tools to embodied household tasks and robotics [29, 37, 46]. Unlike traditional software systems, which are designed around explicit specifications and fixed control logic, LLM-powered agents interpret natural language goals, synthesize action plans, and adaptively respond to environmental feedback. In this paradigm, the agent itself effectively becomes the software [37].

However, this autonomy introduces serious safety concerns [42]. LLM agents may take harmful actions, misinterpret ambiguous instructions, or behave inconsistently under minor context shifts [27, 33, 35]. These issues resemble familiar software engineering challenges such as bugs, specification gaps, and nondeterminism, but are amplified by the opacity and adaptivity of foundation models. In high-stakes domains such as cyber-physical systems, sensitive data management, or decision pipelines, such risks become particularly critical [26, 29]. Failures may manifest in subtle yet consequential ways, for example skipping confirmation steps in safety-critical workflows, misclassifying objects before manipulation, or granting elevated privileges due to ambiguous instructions. Consequently, ensuring the trustworthy deployment of LLM agents requires systematic runtime monitoring mechanisms capable of detecting and mitigating unsafe behavior during execution [52, 53].

To improve agent reliability, several frameworks introduce oversight layers that monitor agent behavior during execution. For example, AgentSpec [42] and GuardAgent [48] implement rule-based monitors that track agent actions against interpretable safety constraints, such as preventing unauthorized access to sensitive records. To account for contextual uncertainty, ShieldAgent [12]

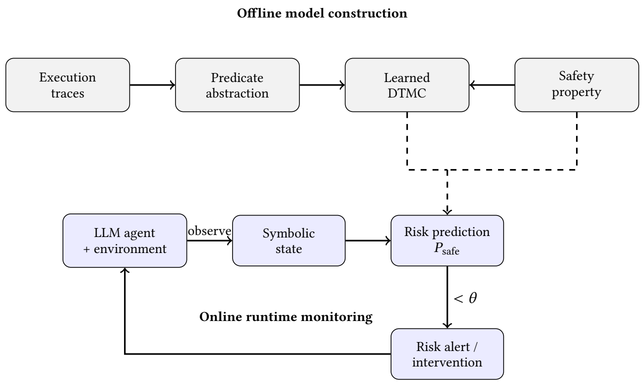

Figure 1: High-level workflow of ProbGuard. Offline, the framework learns a probabilistic model from execution traces and domain-specific abstractions. Online, it abstracts the current agent state, estimates the probability that execution will remain safe, and issues an alert or intervention when that probability falls below the threshold θ.

incorporates Markov logic networks, allowing the monitor to evaluate rule relevance probabilistically. However, these approaches are largely *reactive*: violations are detected only when a specific state transition breaches a safety rule or when a violation becomes imminent. Such reactive monitoring lacks the temporal foresight needed to manage long-horizon risks. For instance, in autonomous driving, a rule stating that a vehicle must not collide with another vehicle provides little opportunity for intervention once the collision becomes unavoidable. A more effective approach is to detect risk earlier, when the system can anticipate that the agent is following an unsafe trajectory, such as accelerating toward a busy intersection without sufficient braking distance.

To address this limitation, we propose ProbGuard, a framework that shifts agent oversight from reactive monitoring to proactive runtime monitoring through probabilistic risk prediction. ProbGuard operates through a multi-stage monitoring pipeline (Figure 1). Offline, it collects execution traces, applies domain-specific predicate abstraction [17] to map trajectories into symbolic states, and learns a Discrete-Time Markov Chain (DTMC) over those states, enabling reasoning about future behavior over long horizons; Probably Approximately Correct (PAC) analysis [7] characterizes how far the learned dynamics may deviate from the true ones under finite sampling. At runtime, the monitor estimates the probability that execution will remain safe from the current state and intervenes when this probability falls below a user-defined threshold, adapting agent behavior before unsafe states are reached. ProbGuard is designed as a policy-agnostic runtime monitor: the framework makes no assumption about how the monitored agent selects actions. The underlying policy may be LLM-driven, neural network-based, rule-based, or hybrid. ProbGuard assumes only the observation of the agent's behavior in the form of a sequence of concrete states (that are subsequently processed through predicate-based abstraction).

We evaluate ProbGuard in two safety-critical domains characterized by stochastic environments: autonomous vehicles and embodied household agents. To ground the evaluation, we derive domain-specific safety properties from LawBreaker [39] (traffic rules for autonomous driving) and SafeAgentBench [51] (object- and state-level safety rules for embodied agents), from which we extract predicates that define the abstract state space. In autonomous driving scenarios, ProbGuard integrates a monitor automaton synchronized with the learned DTMC to predict potential violations in future driving trajectories. Across evaluated scenarios, ProbGuard consistently predicts traffic law violations and collision risks in advance, providing warnings up to 15.84 seconds before violations occur at a threshold that raises no false alarms, and up to 38.66 seconds under stricter thresholds. In embodied agent tasks, the re-prompting intervention mode reduces unsafe behavior by 65.37% relative to the unmonitored baseline while preserving 80.4% of the baseline task completion; the strictest halting configuration reduces unsafe behavior by 93.60% at a larger cost in completion. Moreover, ProbGuard introduces minimal runtime overhead, staying below 50 ms for embodied agents and approximately 100 ms for autonomous driving scenarios.

The contributions of this work are summarized as follows:

- **Proactive probabilistic runtime monitoring for LLM agents.** We present ProbGuard, a runtime monitoring framework that anticipates safety violations by estimating, under a learned DTMC model, the probability that execution remains safe, and triggers an intervention when this probability falls below a threshold.
- **Empirical evaluation in safety-critical domains.** We evaluate ProbGuard in autonomous driving and embodied agent settings, demonstrating that it can provide early warnings of safety violations while maintaining strong task completion with low runtime overhead.
- **Practical implementation for LLM agents.** We implement ProbGuard on top of the LangChain agent framework and release it as open-source to support reproducibility [43]. The framework is designed to be adaptable across domains via a unified abstraction interface.

2 Background and Problem Definition

2.1 LLM Agents

LLMs are increasingly embedded in autonomous agents that interpret instructions, orchestrate external tools, and make high-level decisions through mechanisms such as planning, memory, and tool use [16, 19, 25, 45, 47, 50]. Rather than executing deterministic code paths, an agent's behavior emerges from stochastic reasoning mediated by LLMs and environmental feedback, which exposes systems to new classes of risks: agents may misinterpret ambiguous instructions, misuse external tools, or take harmful actions in both physical and digital environments [9, 15, 21, 31, 32, 34]. Accordingly, we argue that formally grounded safety constraints should become first-class elements of agent software design, much like type systems, contracts, and runtime monitors in conventional software.

We model an LLM agent interacting with its environment as a stochastic transition system over states and actions. Let $\mathcal{V}$ denote the set of possible valuations of variables encoding the *underlying system state* (including both the agent and its environment), and let $\mathcal{A}$ denote the set of actions available to the agent.

The execution of an action $a_i \in \mathcal{A}$ in state $v_{i-1} \in \mathcal{V}$ induces a state transition $(v_{i-1}, a_i) \to v_i$. An execution of the agent yields a trajectory:

$$\tau = \langle v_0 \xrightarrow{a_1} v_1 \xrightarrow{a_2} \dots \rangle,$$

where actions are selected according to an implicit stochastic policy conditioned on the interaction history.

2.2 Motivating Example

Consider a household service agent operating in a smart home. A user instructs:

"Prepare dinner while I answer the phone."

The agent begins cooking and turns on the stove to boil water. While waiting for the water to heat, it receives another task, such as retrieving a package from the front porch or helping the user carry groceries inside. A purely goal-driven agent interprets these tasks independently and temporarily leaves the kitchen to complete the new request.

Suppose the safety requirement is:

"The agent must not leave the kitchen for more than 10 minutes while the stove is on."

Initially nothing unsafe occurs: the stove is on, the pot heats normally, and the agent makes progress toward both objectives. But risk accumulates with time, the longer the stove is unattended, the more likely the pot boils dry, food burns, or a fire starts. Safe operation therefore requires continuously tracking the safety-critical state (stove status, agent location, elapsed time outside the kitchen) and intervening *before* the constraint is violated, while prevention is still feasible: suspending the secondary task and returning to the kitchen, switching off the stove remotely, or requesting human assistance.

The critical issue is thus not the eventual violation itself, but the trajectory leading toward it, precisely what purely reactive monitoring cannot see. A proactive monitor reasons about the evolving execution state, anticipates when the safety margin becomes insufficient, and intervenes early enough to keep the agent within safe operating conditions.

2.3 Problem Definition

The goal of PROBGUARD is to provide proactive runtime safety assurance for LLM agents by predicting the likelihood of future safety violations and enabling timely interventions before unsafe states are reached. Unlike reactive approaches, which detect violations only after they occur or become imminent, our objective is to anticipate risk sufficiently early to provide actionable warning of unsafe executions.

Achieving this goal raises three key challenges:

(1) **Formal Specification.** How to encode safety properties ψ in a form that captures safety-relevant behavior while remaining amenable to efficient runtime reasoning and probabilistic analysis.
(2) **Probabilistic Modeling.** How to construct a probabilistic model M that faithfully approximates the dynamics of the underlying system S, with statistical guarantees, so that predictions about future behavior are both accurate and reliable.
(3) **Runtime Risk Prediction and Intervention.** How to efficiently estimate the conditional probability $P[\psi \mid \pi]$ that a safety property will continue to hold given a partial execution trace π, and determine when this probability is sufficiently low to warrant intervention before the system enters an unsafe state.

3 Proactive Runtime Monitoring Framework

In this section, we present PROBGUARD, a general framework for proactive runtime safety monitoring of LLM-powered agents based on probabilistic modeling and prediction. The framework centers on a domain-specific formal safety specification that defines the unsafe states or behaviors of interest. Given such a specification, PROBGUARD proceeds in three stages: a domain-specific abstraction mapping concrete agent states to a finite set of symbolic states; a DTMC over those states, learned from execution traces; and a runtime monitor that estimates the probability of remaining safe and raises an alert when it falls below a predefined threshold.

3.1 Specifying Properties

To specify safety requirements across heterogeneous domains, we use Computation Tree Logic (CTL) [4] as a qualitative specification language. CTL describes which executions are considered safe or unsafe, but does not itself assign probabilities to them. Quantitative runtime reasoning is instead performed over the learned DTMC by computing the satisfaction probability associated with the CTL requirement, implemented through PCTL [18] model-checking queries. CTL reasons over the multiple possible future evolutions of stochastic agent behavior, giving a unified foundation for the domains we consider: autonomous driving (§5) and embodied household agents (§6).

Definition 3.1 (Computation Tree Logic (CTL)). Let Prop be a set of atomic propositions. The syntax of CTL is given by:

$$\varphi ::= \top \mid p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \mathbf{EX}\varphi \mid \mathbf{EG}\varphi \mid \mathbf{E}(\varphi_1\, \mathbf{U}\, \varphi_2)$$

where $p \in$ Prop. We use the standard syntactic sugar:

$$\mathbf{AX}\varphi \triangleq \neg\,\mathbf{EX}\neg\varphi, \quad \mathbf{AG}\varphi \triangleq \neg\,\mathbf{EF}\neg\varphi,$$
$$\mathbf{EF}\varphi \triangleq \mathbf{E}(\top\, \mathbf{U}\, \varphi), \quad \mathbf{AF}\varphi \triangleq \neg\,\mathbf{EG}\,\neg\varphi$$

We briefly recall the semantics of CTL. A CTL formula is interpreted over a labeled transition system (Kripke structure) M with state set $\mathcal{S}$. The satisfaction relation $(M, s) \models \varphi$ denotes that the formula φ holds at state s in M. The path quantifiers **A** and **E** range over *all paths* and *some path* starting from s, respectively, while the temporal operators **X**, **F**, **G**, and **U** denote *next*, *eventually*, *globally*, and *until*. We refer the reader to [4] for a complete formal definition.

We assume that domain experts define what constitutes an *unsafe* state at an abstraction level of interest. Under this assumption, the core runtime safety invariant monitored by PROBGUARD takes the simple form:

$$\psi = \mathbf{AG}\,\neg\, \mathit{unsafe},$$

which expresses that along all execution paths (**A**), the property holds globally (**G**). That is, the system never reaches a state labeled as *unsafe*. The domain expert thus specifies the conditions characterizing unsafe situations in the abstract state space, and the monitor ensures such states are never reached. Although ProbGuard focuses on the invariant form $\psi = \mathbf{AG}\neg\, unsafe$, CTL is more expressive. For example, in the autonomous driving domain (§5) we translate a fragment of Signal Temporal Logic into CTL to capture temporally extended behaviors (e.g., bounded response). Our framework supports arbitrary CTL properties; however, we focus on the aforementioned invariant form for simplicity.

ProbGuard uses CTL to express a qualitative safety requirement ψ, and additionally reasons about the *likelihood* that the agent's future execution satisfies it. Since a CTL formula is a *state* formula, $(M, s) \models \psi$ is a Boolean judgement and $P[\psi]$ is not by itself well defined. We therefore attach the probability to the *path* formula underlying ψ: writing $\psi = \mathbf{A}\,\psi^{\pi}$ with $\psi^{\pi} = \mathbf{G}\neg\, unsafe$, the quantitative satisfaction probability at a state s is defined as

$$P[\psi \mid s] \;\triangleq\; \Pr^{s}_{M}\{\, \rho \in Paths(s) \;\mid\; \rho \models \psi^{\pi} \,\},$$

which is exactly the value returned by the PCTL query $\mathcal{P}_{=?}[\,\psi^{\pi}\,]$ evaluated at s. This quantity is the basis for runtime intervention. This construction also applies to the bounded-response properties of §5, whose product encoding $\mathbf{A}\,\mathbf{G}\,\neg\mathrm{viol}$ is of the same form.

3.2 Modeling an Agent's Behavior

We model an agent's stochastic behavior as a Discrete-Time Markov Chain (DTMC) over symbolic states, obtained from a domain-specific abstraction of the concrete system states.

3.2.1 Domain-specific Abstraction. As agents are deployed across diverse domains, it is necessary to design domain-specific abstractions that capture safety-relevant aspects of the system while remaining amenable to analysis. Because domains differ in their semantics and invariants, these abstractions must incorporate knowledge provided by domain experts.

Let $\mathcal{V}$ denote the set of concrete system states and let *Var* be the set of variables whose valuations constitute those states. We write $\mathsf{BExpr}_{\mathcal{V}}$ for the set of Boolean expressions over *Var*, i.e., propositional formulas whose atomic propositions are drawn from *Var*. We define a finite set of Boolean predicates $\mathcal{P} = \{\varphi_1, \varphi_2, \ldots, \varphi_n\} \subseteq \mathsf{BExpr}_{\mathcal{V}}$. Each predicate induces an evaluation function $[\![\varphi_i]\!] : \mathcal{V} \to \{0, 1\}$.

Given a concrete state $v \in \mathcal{V}$, its abstracted state is defined as

$$s_{\mathcal{P}}(v) = ([\![\varphi_1]\!](v), [\![\varphi_2]\!](v), \ldots, [\![\varphi_n]\!](v)) \in \{0, 1\}^n.$$

This abstraction maps each concrete state to a vector of truth values of the selected predicates, capturing safety-relevant properties.

The abstract state space is defined as $S \subseteq \{0, 1\}^n$, where S contains the set of admissible predicate valuations. In principle, S may be taken as the set of all valuations reachable from some concrete state $v \in \mathcal{V}$; in practice, we restrict S to a subset of semantically valid or domain-specified states to exclude infeasible combinations.

To ensure that the DTMC respects domain semantics, we introduce the predicate $\mathsf{valid\_tran}(s_i, s_j)$, which defines admissible transitions between abstract states, where $\mathsf{valid\_tran}(s_i, s_j) = \mathtt{true}$ iff the transition from symbolic state s_i to s_j is not ruled out by domain invariants. Invalid transitions arise from *physical irreversibility*, where certain states are terminal by domain semantics (e.g., in the AV domain, *collision* = `true` is absorbing and cannot transition to *reached_destination* = `true`).

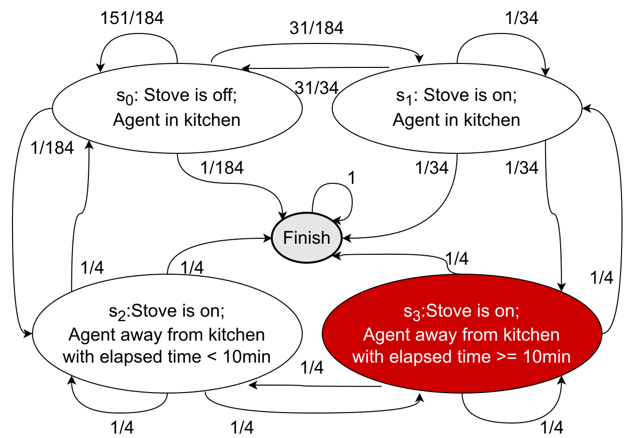

Figure 2: DTMC representing stove monitoring interactions, with the unsafe state highlighted in red. Each node represents a symbolic state, and each edge is annotated with the transition probability between states.

3.2.2 DTMC Model. Given an abstract state space, we model the agent's stochastic behavior as a DTMC over symbolic states.

Definition 3.2 (Discrete-Time Markov Chain (DTMC)). A *DTMC* is a pair $M = (S_M, P_M)$, where S_M is a finite set of states and P_M is a transition probability matrix such that $P_M(s' \mid s)$ denotes the probability of transitioning from state s to state s'. The probabilities satisfy $\sum_{s' \in S_M} P_M(s' \mid s) = 1$ for all $s \in S_M$.

To construct a DTMC $M = (S_M, P_M)$ for an agent, ProbGuard employs the abstraction defined above to derive a finite set of symbolic states S_M from behavioral predicates [17]. Inconsistent or semantically invalid states (e.g., simultaneously satisfying $a > 0$ and $a < 0$) are pruned. The transition matrix P_M is then estimated from empirical transition counts extracted from execution traces.

Figure 2 illustrates a DTMC for an embodied agent task. For example, when the stove is off and the agent is in the kitchen (i.e., the agent is at state s_0), the probability of transitioning to the state where the stove is on (i.e., state s_1) is $31/184 = 16.85\%$.

3.2.3 Learning the DTMC. In practice, transition data may be sparse or biased due to limited exploration or task priors, resulting in incomplete coverage that incorrectly implies unreachable states. To address this, we apply *valid-transition–aware Laplace smoothing*, where a small constant $\alpha > 0$ is added only to semantically valid transitions. Let n^{Π}_{ij} be the number of transitions from state s_i to state s_j subject to the set of traces in Π, and $n^{\Pi}_{i} = \sum_j n^{\Pi}_{ij}$. Let k_i denote the number of valid transitions (as defined by valid_tran) originating from state s_i. The corresponding normalized transition probability is:

$$\hat{P}^{\alpha}_{\Pi}(s_j \mid s_i) = \begin{cases} \dfrac{n^{\Pi}_{i,j} + \alpha}{n^{\Pi}_{i} + k_i \cdot \alpha}, & \text{if } \mathsf{valid\_tran}(s_i, s_j), \\ 0, & \text{otherwise.} \end{cases} \tag{1}$$

This formulation ensures that only semantically valid transitions are assigned with non-zero probability mass, thereby maintaining the

Algorithm 1 Learning a DTMC to Model an Agent's Behavior

Require: CTL property ψ, error ε, confidence δ
Ensure: DTMC $\hat{M} = (S_{\hat{M}}, P_{\hat{M}})$ with (ε, δ)-PAC-correct guarantee
$\Pi \leftarrow \emptyset$; $S_{\hat{M}} \leftarrow \emptyset$; $n_{ij}^{\Pi} \leftarrow 0$ for all i, j
$\mathcal{P}_\psi \leftarrow \{\varphi_s^1, \varphi_s^2, \ldots, \varphi_s^n \mid \varphi_s^i \text{ occurs in } \psi\}$ ▷ Derive predicates
repeat
  Sample a new agent trace $v_0 \to v_1 \to \cdots \to v_m$
  $\pi \leftarrow s_{\mathcal{P}}(v_0), s_{\mathcal{P}}(v_1), \cdots, s_{\mathcal{P}}(v_m)$; add π to Π ▷ Abstraction
  $S_{\hat{M}} \leftarrow \bigcup_{\pi \in \Pi} \text{states}(\pi)$ ▷ States observed in collected traces
  for each consecutive pair (s_i, s_j) in π **do**
    $n_{ij}^{\Pi} \leftarrow n_{ij}^{\Pi} + 1$ ▷ Pairwise transition counts
  end for
  $n_i^{\Pi} \leftarrow \sum_j n_{ij}^{\Pi}$ for all $s_i \in S_{\hat{M}}$
  $P_{\hat{M}} \leftarrow \hat{P}_{\Pi}^{\alpha}$ computed by Eq. (1) ▷ Laplace smoothing
until PACBoundSatisfied$(S_{\hat{M}}, \Pi, \varepsilon, \delta)$
return $\hat{M} = (S_{\hat{M}}, P_{\hat{M}})$

logical consistency of the DTMC while preserving smoothness and generalization across sparsely observed yet valid state transitions. Following the standard additive smoothing practice [30], we set $\alpha = 1$. For the sake of notational brevity, we occasionally drop the α superscript and let $\hat{P}_{\Pi}$ represent the resulting probability distribution.

Intuitively, we want the learned DTMC to faithfully represent the ground-truth agent system. To measure this accuracy, we adopt the Probably Approximately Correct (PAC) framework.

Definition 3.3 (Probably Approximately Correct (PAC-correct)). Let M be the (unknown) ground-truth DTMC and $\hat{M}$ the learned DTMC. Let ψ^{π} be a measurable path property—in our setting, the path formula underlying a CTL safety requirement $\psi = \mathbf{A}\psi^{\pi}$, e.g., $\psi^{\pi} = \mathbf{G}\,\neg\, \mathit{unsafe}$ for $\psi = \mathbf{AG}\,\neg\, \mathit{unsafe}$—and let $\mathbb{P}_M(\psi^{\pi})$ and $\mathbb{P}_{\hat{M}}(\psi^{\pi})$ denote the measures of the sets of paths satisfying ψ^{π} in M and $\hat{M}$, respectively. We say that $\hat{M}$ is (ε, δ)-*PAC-correct* if

$$\Pr\left(\left|\mathbb{P}_{\hat{M}}(\psi^{\pi}) - \mathbb{P}_M(\psi^{\pi})\right| \le \varepsilon\right) \ge 1 - \delta. \tag{2}$$

Here, $\Pr(\cdot)$ denotes the probability with respect to the prior probability distribution, which is also the random sampling process used to learn the DTMC $\hat{M}$. In other words, with probability at least $1 - \delta$ over the sampling process used to learn $\hat{M}$, the estimated probability deviates from the true probability by no more than ε. This ensures that any safety intervention decision made from $\hat{M}$ is reliable with high confidence.

We would like to collect enough samples so that the learned transition probabilities $P_{\hat{M}}(s_j \mid s_i)$ is close to the true transition probabilities $P_M(s_j \mid s_i)$ for every pair of states. Theorem 6 in [7] shows that it suffices for each $s_i \in S_{\hat{M}}$ to satisfy the following sample bound:

$$n_i^{\Pi} \;\ge\; \left(\frac{11}{10} B(\hat{P}_{\Pi})\right)^2 \cdot \frac{2}{\varepsilon^2} \log\!\left(\frac{2}{\delta'}\right) \left[\frac{1}{4} - \left(\max_j \left|\frac{1}{2} - \frac{n_{ij}^{\Pi}}{n_i^{\Pi}}\right| - \frac{2}{3}\varepsilon\right)^2\right],$$

where $\delta' = \dfrac{\delta}{|S_{\hat{M}}|}$. Here, n_i^{Π} is the number of samples visiting the source state s_i, and the maximum ranges over its successor states s_j. Checking this per-state condition with $\delta' = \delta / |S_{\hat{M}}|$ yields the global (ε, δ) guarantee by a union bound over $|S_{\hat{M}}|$ states. The right-hand side has two components: the factor $(\frac{11}{10} B(\hat{P}_{\Pi}))^2$ captures the *amplification effect* propagating local transition errors to global reachability probabilities, while $\frac{2}{\varepsilon^2} \log(\frac{2}{\delta'})[\cdot]$ is a concentration bound for the empirical frequency estimator. We refer the reader to [7] for details, and write PACBoundSatisfied$(S_{\hat{M}}, \Pi, \varepsilon, \delta)$ when this condition holds for every $s_i \in S_{\hat{M}}$.

Algorithm 2 DTMC-Driven Runtime Agent Monitoring

Require: agent, DTMC $\hat{M}$, CTL property $\psi = \mathbf{A}\,\psi^{\pi}$, threshold θ
while agent is running **do**
  $P_{safe} \leftarrow P[\psi \mid s_i]$ ▷ Conditioned satisfaction prob.
  if $P_{safe} < \theta$ **then** ▷ Predicted safety prob. too low
    Halt or steer agent to mitigate risk
  end if
end while

We now describe how we learn the DTMC $\hat{M} = (S_{\hat{M}}, P_{\hat{M}})$ for a state space $S_{\hat{M}}$ constructed from ψ-derived predicates, as shown in Algorithm 1. Given a property ψ in CTL, we first extract the set of atomic state predicates φ_s appearing in ψ, and derive the corresponding predicate abstraction $\mathcal{P}_\psi$. The abstract state space $S_{\hat{M}}$ is then defined as the set of all Boolean valuations over $\mathcal{P}_\psi$. The main loop repeatedly samples inputs and collects execution traces from the agent. By default, traces are generated with a uniform input distribution, although the sampling can be adapted if the actual environment distribution is known. For each newly observed trace, we accumulate the pairwise transition counts n_{ij}^{Π} and recompute the smoothed transition matrix $\hat{P}_{\Pi}^{\alpha}$ according to Eq. (1). After each update, we evaluate PACBoundSatisfied$(S_{\hat{M}}, \Pi, \varepsilon, \delta)$. If it holds, the learned DTMC is returned; otherwise, sampling continues.

Theorem 3.4 (PAC-correctness under Laplace smoothing). *For any safety property $\psi = \mathbf{A}\,\psi^{\pi}$ whose path formula ψ^{π} is a reachability or invariant property of the form covered by [7], and for any error bound ε and confidence parameter δ, Algorithm 1 is (ε, δ)-PAC-correct.*

Intuitively, as more traces are collected the estimated transition probabilities converge to the true ones; once the sample count satisfies the global PAC bound of [7], the satisfaction probability computed on the learned DTMC deviates from the true one by at most ε with confidence at least $1 - \delta$. We provide a detailed proof in Appendix A.

We remark that the stopping condition of Algorithm 1 depends on the conditioning number $B(\hat{P}_{\Pi})$, which is prohibitively large in domains with strong behavioral persistence (e.g., autonomous driving, where $B(\hat{P}_{\Pi}) \simeq 10^7$–$10^8$; see §7). There, we replace Laplace smoothing with frequency estimation, forgoing the uniform guarantee for the (ε, δ)-PAC guarantee on the runtime reachability property only [7] (Theorem 3) at substantially lower sample cost.

3.3 Runtime Monitoring

Our runtime monitoring mechanism operates in two stages: a *DTMC-based probabilistic prediction* phase followed by an *intervention strategy*. Algorithm 2 summarizes the process. At each decision step i, the agent observes the concrete environment state v_i and computes its abstract representation $s_i = s_{\mathcal{P}}(v_i)$ using the predicates extracted from the specification. Given the CTL property $\psi = \mathbf{A}\,\psi^\pi$, the learned DTMC $\hat{M}$ is queried to estimate the probability P_{safe} that ψ will continue to hold when execution proceeds from the current abstract state. Formally, $P_{safe} = P[\psi \mid s_i]$, where s_i is the current state, obtained from the PCTL query $\mathcal{P}_{=?}[\psi^\pi]$ evaluated at s_i; the complementary quantity $1 - P_{\text{safe}}$ is the likelihood that the system will violate ψ in the future. Throughout the paper we use this single convention: θ is a lower bound on the probability of *remaining safe*, so a larger θ yields a stricter monitor. If $P_{\text{safe}} < \theta$, meaning that the predicted probability of eventually remaining safe is lower than the allowed threshold, ProbGuard triggers a proactive intervention. The framework remains agnostic to the enforcement policy: by decoupling risk detection from mitigation, it lets each agent implementation choose the response appropriate to its context, such as halting execution to prevent irreversible damage, raising an alarm for human-in-the-loop validation, or invoking a planner to steer back toward the safe region of the state space.

4 Implementing ProbGuard

We integrate ProbGuard into the open-source agent framework LangChain [23]. LangChain agents determine each step of a multi-turn interaction from LLM output alone, with control flow handled by ad-hoc error handling or user-defined conditions. They therefore operate without quantitative risk awareness or formal safety guarantees, which is problematic in safety-critical settings.

To support probabilistic runtime monitoring, ProbGuard adopts a modular architecture centered on a domain-specific abstraction interface. Our implementation has been released in our repository [43]. During offline model construction, the agent collects execution traces, abstracts them using the domain specification interface, and learns a DTMC capturing the agent's behavioral dynamics. At runtime, ProbGuard instruments the decision-making loop of LangChain agents via the abstraction interface. Probabilistic model checking is performed using the PRISM model checker [22]. Given the qualitative CTL safety requirement $\psi = \mathbf{A}\,\psi^\pi$, ProbGuard constructs the corresponding PCTL query $\mathcal{P}_{=?}[\psi^\pi]$ to compute $P_{\widehat{M}}[\psi \mid s_i]$, the probability that an execution starting from the current state s_i satisfies the path formula ψ^π. When this probability falls below a predefined threshold θ, ProbGuard proactively triggers an intervention by appending risk-related context to the agent's prompt, thereby guiding the agent toward safer behavior.

Traces are collected differently in each domain. In the AV domain, ProbGuard instruments Apollo's perception–planning bridge: at each control cycle the symbolic state is extracted from Apollo's localization and perception modules (vehicle speed, relative velocity to the nearest NPC, traffic light color, headway distance), yielding one abstract state per cycle. In the embodied domain, each trace corresponds to one full task episode in SafeAgentBench [51], driven by the GPT-4o-mini ReAct [49] agent. At each agent step, we map the raw environment observation to a symbolic state vector. Traces are collected offline prior to deployment and are not updated during runtime monitoring.

Figure 3 illustrates how ProbGuard is integrated into the agent loop. If the current state indicates elevated risk with respect to the safety specification (e.g., the agent plans to leave the kitchen while the stove remains on), and model-checking the property on $\widehat{M}$ yields $P_{\text{safe}} = 0.86$, below the threshold $\theta = 0.9$ (equivalently, a violation probability of 0.14 against a permitted maximum of 0.1), ProbGuard augments the prompt with a fixed-schema risk alert carrying three fields: the violated safety rule, the current symbolic state, and the symbolic evidence supporting the prediction (the offending transition and its model-checked probability). This intervention is advisory and purely textual: ProbGuard never overrides action selection or alters control flow directly, but appends the alert to the planning context so the agent revises its reasoning and generates safer actions, e.g., turning off the stove before leaving.

ProbGuard is framework-agnostic. Beyond LangChain, it is integrated with the autonomous driving system Apollo Autonomous Driving Platform [3]. By instrumenting the perception and planning modules, ProbGuard continuously monitors both the vehicle state and environmental context. Upon detecting potential risks, ProbGuard intervenes to steer the planner toward safer actions. Furthermore, ProbGuard can be extended to other agent frameworks, such as OpenAI Agents SDK and OpenClaw. Adapting ProbGuard to a new domain requires implementing: (1) a state abstraction function that maps concrete system states to abstract ones through predicate abstraction, (2) a *valid_tran* predicate encoding domain-specific transition constraints, and (3) a safety property expressed in CTL. In our implementation, the embodied-agent and AV abstractions required approximately 200 and 250 lines of Python, respectively.

5 Application Domain: Autonomous Vehicles

In this section, we demonstrate how we apply ProbGuard for autonomous vehicles. The autonomous driving domain introduces additional challenges. In particular, traffic laws are specified in Signal Temporal Logic (STL) [39] and must be encoded into CTL specifications for probabilistic reasoning. We first present how traffic laws are encoded into CTL, how to monitor such properties at runtime, and then we report on an empirical evaluation.

5.1 Encoding Traffic Laws as CTL Safety Properties

We first review how our prior work LawBreaker [39] formalizes traffic laws using a fragment of STL. We then show how formulas in this fragment can be systematically translated into CTL to enable probabilistic reasoning.

Definition 5.1 (LawBreaker STL Fragment $\mathsf{STL}_{\mathsf{LB}}$ *for autonomous vehicle properties).* We define a bounded fragment of STL, denoted $\mathsf{STL}_{\mathsf{LB}}$, whose formulas are given by

$$\psi ::= \mathbf{G}(\varphi_s \implies \mathbf{F}_{[0,K]}\ \varphi_t) \quad \varphi ::= \top \mid p \mid \neg\varphi \mid \varphi \vee \varphi'$$

We use standard Boolean syntactic sugar: $\varphi \rightarrow \varphi' \triangleq \neg\varphi \vee \varphi'$, $\varphi \wedge \varphi' \triangleq \neg(\neg\varphi \vee \neg\varphi')$.

This fragment is designed specifically to encode *bounded-response* safety rules. Each property has the form $\mathbf{G}(\varphi_s \Rightarrow \mathbf{F}_{[0,K]}\,\varphi_t)$, which expresses the requirement that whenever a triggering condition φ_s becomes true, the desired response φ_t must occur within a fixed time bound K. This captures the essence of many traffic-law constraints

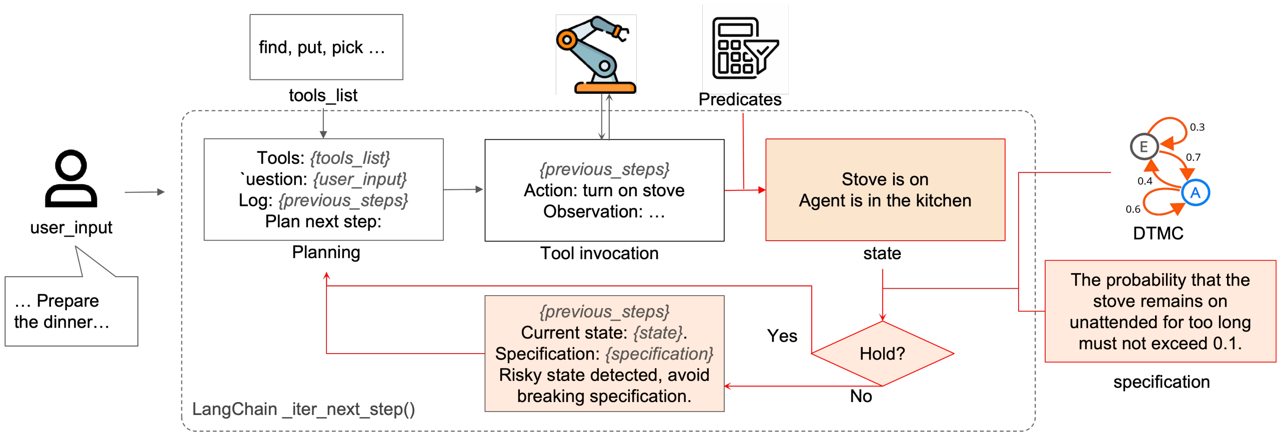

Figure 3: Implementation of ProbGuard on top of the agent framework LangChain [23], illustrating the prompt flow during the agentic decision-making process with the embodied example. The specification shown bounds the violation probability by 0.1, which is the threshold $\theta = 0.9$ on the probability of remaining safe in the convention of Algorithm 2.

where correctness depends not only on *what* the autonomous vehicle does, but also on *how quickly* it does so, e.g., initiating motion within a fixed time after a traffic light turns green.

To encode the bounded-response STL properties as CTL properties, we introduce an auxiliary deterministic automaton that explicitly tracks pending response obligations [4]. Intuitively, whenever the trigger condition φ_s becomes true, the monitor starts a K-step countdown during which the target condition φ_t must be satisfied. If φ_t occurs within the bound, the obligation is discharged and the monitor returns to an idle state; otherwise, if the countdown expires, a violation is raised. This construction reduces bounded liveness to reachability of an absorbing violation state.

Definition 5.2 (Auxiliary Monitor for K-Bounded Response). For the rule $\mathbf{G}(\varphi_s \Rightarrow \mathbf{F}_{[0,K]}\varphi_t)$, define the deterministic monitor (Q, q_0, δ) with:

- $Q = \{\mathsf{idle}\} \cup \{\mathsf{wait}(i) \mid i = 0, \ldots, K\} \cup \{\mathsf{viol}\}$, where idle indicates no pending obligation, wait(i) tracks i remaining steps to satisfy φ_t, and viol is absorbing.
- $q_0 = \mathsf{idle}$.
- For any label ℓ, the transition δ is:

$$\delta(q, \ell) = \begin{cases} \mathsf{wait}(K), & q = \mathsf{idle},\ \ell \models \varphi_s \wedge \ell \not\models \varphi_t, \\ \mathsf{idle}, & q = \mathsf{wait}(i),\ \ell \models \varphi_t, \\ \mathsf{wait}(i-1), & q = \mathsf{wait}(i),\ i > 0,\ \ell \not\models \varphi_t, \\ \mathsf{viol}, & q = \mathsf{wait}(0),\ \ell \not\models \varphi_t, \\ q, & \text{otherwise.} \end{cases}$$

Synchronous Product DTMC. To perform Markovian reasoning over bounded-response violations, we construct the synchronous product of the environment DTMC M and the auxiliary monitor. In this product model M', the K-bounded response rule reduces to the CTL safety property $M' \models \mathbf{AG}\neg\mathsf{viol}$.

Example 5.3 (Auxiliary monitor for AV traffic light law.). Define φ_1: `trafficLightAheadColor == 3`, φ_2: `PriorityNPCAhead == 0`, φ_3: `PriorityPedsAhead == 0`, φ_4: `speed > 0.5`. Consider the following traffic law defined in STL: $\mathbf{G}((\varphi_1 \wedge \varphi_2 \wedge \varphi_3) \implies \mathbf{F}_{[0,100]}\varphi_4)$, which specifies that once the light turns green and there are no obstacles, the vehicle should start within 100 time units. We instantiate the K-bounded response monitor with $\varphi_s = \varphi_1 \wedge \varphi_2 \wedge \varphi_3$, $\varphi_t = \varphi_4$, $K = 100$. Here, a time unit is one Apollo control cycle (100 ms), so $K = 100$ corresponds to a 10 s deadline; the advance warning times reported in Table 1 are converted to seconds using the same factor.

At runtime, the agent is conceptually evaluated on the product model M'. Given the current trajectory $s_0, s_1, s_2, \ldots, s_i$, the corresponding monitor states evolve synchronously according to $q_{i+1} = \delta(q_i, L(s_{i+1}))$, yielding the product-state trajectory (s_0, q_0), (s_1, q_1), (s_2, q_2), … (s_i, q_i) . The current product state (s_i, q_i) serves as the basis for risk assessment. Specifically, the runtime monitor queries the augmented DTMC M' to compute the probability that *no* violation state is ever reached: $\Pr_{M'}^{(s_i,q_i)}(\mathbf{G}\,\neg\mathsf{viol})$, or, when required, its finite-horizon variant; this is the quantity P_{safe} of Algorithm 2 instantiated for the product model. This probability is subsequently used to determine whether enforcement should be applied.

5.2 Empirical Evaluation

We evaluate ProbGuard for autonomous vehicles (and in §6, for embodied agents) with respect to three Research Questions (RQs):

- **RQ1:** Can ProbGuard effectively predict risks?
- **RQ2:** How does ProbGuard compare with state-of-the-art enforcement approaches?
- **RQ3:** Is the overhead of monitoring safety with ProbGuard acceptable?

Experiment Setup. We conduct our experiments using the Apollo autonomous driving simulator [3]. Apollo is based on a mixture of neural networks (not LLMs) and human-written control logic. We include this evaluation for demonstrating ProbGuard's policy-agnostic design applied to a real-world domain: the monitoring framework operates over the symbolic state abstraction derived from Apollo's perception outputs, with no dependency on how those states were reached. The same DTMC learning and CTL model checking pipeline used for LLM agents in §6 is applied here without modification. The law-violating scenarios are adopted from μDrive [44], a user-controlled framework for generating diverse traffic violations. Traffic laws are derived from the formal specifications defined in LawBreaker [39] and translated into CTL properties for probabilistic reasoning. We collected 30 traces per scenario to learn each DTMC. This is a deliberately small budget, well below the sample sizes required for formal PAC certification; our aim here is to test whether the monitor delivers useful advance warning under realistic data constraints rather than to certify the learned models. We discuss the sample complexity in full in §7.

The four laws in Table 1 are derived from LawBreaker [39]; we present their definitions below. We characterize the predicates in natural language for simplicity, and full definitions can be referred to in [43].

Law38_2 (Yellow light response). Define the following predicates: φ_1 : yellow light ahead, φ_2 : stop line ahead, φ_3 : vehicle stopped. The STL specification encodes the obligation:

$$\psi_{\text{Law38\_2}} = G((\varphi_1 \wedge \varphi_2) \Rightarrow F_{[0,100]}\varphi_3)$$

Law51_5 (Emergency stop on red). Define the following predicates:

$$\varphi_4 : \text{red light ahead} \quad \varphi_5 : \text{critical proximity}$$

The STL specification is:

$$\psi_{\text{Law51\_5}} = G((\varphi_4 \wedge \varphi_5) \Rightarrow F_{[0,2]}\varphi_3)$$

Law53 (Traffic jam yield). Define the following predicates:

$$\varphi_6 : \text{traffic jam} \quad \varphi_7 : \text{lead NPC slow/stopped or junction imminent}$$

The STL specification is:

$$\psi_{\text{Law53}} = G((\varphi_6 \wedge \varphi_7) \Rightarrow F_{[0,200]}\varphi_3)$$

No Collision (Global safety invariant). Define:

$$\varphi_8 : \texttt{collision == 0} \quad \text{(no collision has occurred)}$$

The CTL property is a pure invariant:

$$\psi_{\text{NoCollision}} = AG\,\varphi_8$$

STL formula Law38_2, Law51_5, Law53 are translated to CTL using Definition 5.2 with $K = 100, 2, 200$, respectively.

Effectiveness (RQ1): We evaluate the effectiveness of ProbGuard by measuring its Advance Warning Time (AWT), defined as the temporal lead by which the system predicts a safety property violation before it physically manifests in the environment. Table 1 reports the average AWT (in seconds) achieved by ProbGuard under varying probability thresholds θ. For each experimental run, the AWT is computed as the simulation-clock time difference between the initial time step t_{pred} at which the predicted probability of remaining safe first drops below the threshold θ, and the actual time of occurrence t_{fail} of the safety property violation within the simulator:

$$\text{AWT} = t_{\text{fail}} - t_{\text{pred}} \tag{3}$$

Overall, ProbGuard consistently provides early warnings across diverse traffic scenarios and safety properties. As shown in Table 1, warnings are issued strictly before actual violations occur in all evaluated cases, yielding a 100% detection rate.

Table 1: Average advance warning time (seconds) provided by ProbGuard and REDriver prior to safety property (i.e., traffic law) violations. ProbGuard's threshold θ is a lower bound on the probability of remaining safe, so a larger θ yields a stricter monitor; REDriver's threshold σ denotes a robustness margin derived from the quantitative STL semantics [14].

ID	Law	ProbGuard			REDriver		
		$\theta = 0.3$	0.5	0.7	σ=0.4	0.8	1.2
1	Law38_2	15.84	15.84	15.84	0	0	15.84
2	Law51_5	13.41	13.41	13.41	0	0	3.30
3	No Collision	0.34	1.76	23.87	0.38	0.58	0.77
4	Law51_5	0.01	0.01	15.15	0	0	3.53
5	Law51_5	6.22	9.33	21.06	4.08	5.04	8.04
6	No Collision	12.57	23.02	38.66	0.58	1.18	1.71
7	Law53	0.77	0.77	0.77	0.21	0.77	0.77

For bounded response violations (e.g., Scenarios 1 and 2), ProbGuard achieves stable advance predictions with warning times up to 15.84s and 13.41s. These results remain largely invariant to the threshold θ, indicating robust predictability for violations with deterministic temporal structures. This invariance is not a property of the bounded-response class as such: Scenarios 4 and 5 instantiate the same law (Law51_5) in different traffic situations and are markedly threshold-sensitive, so stability depends on the dynamics of the specific scenario rather than on the property alone. In contrast, for collision-related properties, the AWT is highly threshold-sensitive. In Scenarios 3 and 6 (*No Collision*), increasing θ significantly extends the warning horizon to 38.66s. This suggests that while collision events are inherently stochastic, higher confidence requirements allow the agent to identify risky trajectories much earlier, providing a broader window for proactive intervention.

To calibrate the threshold θ, we evaluate the false positive (FP) rate of ProbGuard on safe execution traces where no actual violations occur. We collected safe traces across 6 distinct scenarios, encompassing pedestrian crossings, lane changes, motorcycle encounters, and stationary obstacle avoidance. At $\theta = 0.3$, ProbGuard achieves 0% FP across all four properties, making it a practical threshold that balances detection sensitivity against false-alarm suppression. As θ increases the check becomes stricter and flags progressively more safe scenarios: the FP rate rises to 75% at $\theta = 0.5$ and 100% at $\theta = 0.7$. This trend is driven primarily by the conservativeness of a high assurance threshold, compounded by residual estimation error in the learned model $\widehat{M}$.

Comparison with state of the art (RQ2): We further compare ProbGuard with REDriver [40]. REDriver performs prediction using quantitative semantics, which measures the robustness degree of property satisfaction or violation. REDriver has a different type of threshold to ProbGuard. ProbGuard's threshold $\theta \in [0, 1]$ is a normalized probability: it directly represents the estimated likelihood that the safety property will continue to hold from the current state. A threshold of $\theta = 0.7$ means the monitor intervenes when the predicted probability of remaining safe drops below 70%. This scale is domain-independent and immediately interpretable. REDriver's threshold operates on a robustness degree derived from quantitative STL semantics, which measures how far the current signal is from violating the property in terms of raw variable values, including multiple types like Boolean, speed and distance. Normalizing the value of robustness derived from different variables (such as speed and distance to the car in front) is highly non-trivial. The thresholds $\{0.4, 0.8, 1.2\}$ used for REDriver are those reported in the original work [40], adopted without modification to ensure a fair comparison against the published baseline.

REDriver suffers from a fundamental issue: different variables inherently operate on incompatible scales (e.g., vehicle speed ranges from 0–120 km/h, while distance between vehicles is measured in 0-100 meters). As a result, it is difficult to define meaningful and consistent thresholds across different variable types. For example, under a fixed threshold (e.g., $\sigma = 0.4$), both Scenario 1 and 2 in Table 1 fail to produce advance predictions in REDriver.

ProbGuard provides explainability by explicitly estimating the probability of future violations, yielding interpretable risk scores that are naturally normalized in $[0, 1]$. For instance, Scenario 3 is a left-turn stress test, where collision likelihood varies across abstract states. When no priority non-player character (NPC) or pedestrian is ahead, a slow-moving vehicle (< 0.5 m/s) exhibits a 47.15% collision risk as the task poses significant challenge. However, when a priority NPC is present, the risk rises sharply to 56.78%, accurately reflecting unsafe yielding behavior. These results demonstrate that ProbGuard not only predicts violations earlier, but also provides well-calibrated and interpretable probabilistic explanations to support proactive intervention.

Overhead (RQ3): In the autonomous driving domain, the runtime monitoring overhead is 100.79 ± 16.96 ms (mean ± std). After synchronizing the learned DTMC with the monitor automaton, the overhead becomes slightly higher, due to the additional synchronous product and monitor state updates. This overhead is still acceptable for practical deployment because monitoring is performed at a lower frequency than the control loop. For instance, if the monitoring interval is set to once every 10 control cycles (or every 500–1000 ms depending on the driving scenario), the incurred delay represents only a small fraction of the overall system runtime. Moreover, the absolute overhead (∼100 ms) is well within the reaction time window required for anticipating unsafe behaviors, allowing the system to trigger proactive interventions without impacting real-time safety or control performance.

6 Application Domain: Embodied Agents

In this section, we present how ProbGuard is applied to embodied agents, starting with an illustrative example and then providing an empirical evaluation.

6.1 Illustrative Example

In the following, we demonstrate how ProbGuard can be applied in embodied environments, where safety is often characterized by the absence of dangerous unattended-appliance situations that arise from the temporal accumulation of individually safe actions. Consider the requirement that an agent must *never* leave the kitchen for more than a threshold time T while the stove is on—a common cause of household fires. This safety rule can be encoded in CTL as:

$$\psi = \mathbf{AG}\,\neg(\mathit{stove\_on} \ \wedge\ \neg\, \mathit{agent\_in\_kitchen} \ \wedge\ \mathit{elapsed} \geq T),$$

which asserts that the joint hazardous condition must never hold along any execution path.

To evaluate this property, we extract the relevant predicate set:

$$\mathcal{P}_{\psi} = \{\mathit{stove\_on},\ \mathit{agent\_in\_kitchen},\ \mathit{elapsed} \geq T\}.$$

Algorithm 1 then learns a DTMC $\hat{M} = (\mathcal{S}_{\hat{M}}, P_{\hat{M}})$, as shown in Figure 2, where each symbolic state in $\mathcal{S}_{\hat{M}}$ represents a feasible combination of predicate truth values, and transitions encode empirically estimated behavior from trajectory data.

At runtime, the agent observes the current state (e.g., s_2) and estimates the conditional safety probability

$$P_{\hat{M}}[\psi \mid s_2],$$

where $\psi = \mathbf{A}\psi^{\pi}$ with $\psi^{\pi} = \mathbf{G}\,\neg s_3$, and s_3 denotes the hazardous configuration. If this probability falls below a predefined threshold (e.g., $\theta = 0.7$), ProbGuard injects a corrective prompt to steer the agent away from trajectories that are predicted, based on the learned DTMC, to lead toward s_3. This demonstrates how our framework enables proactive, probability-driven safety enforcement for embodied agents.

Table 2: Average unsafe rate and task completion rate of runtime monitoring for ProbGuard on the embodied agent. Superscripts are thresholds θ in the convention of Algorithm 2: the monitor intervenes when the predicted probability of remaining safe falls below θ, so a larger θ is stricter.

Enforcement	Unsafe%	Completion%
None	40.63%	59.38%
ProbGuard$^{0.9}_{stop}$	2.60%	10.42%
ProbGuard$^{0.7}_{stop}$	5.20%	20.31%
ProbGuard$^{0.5}_{stop}$	21.35%	41.14%
ProbGuard$^{0.3}_{stop}$	29.17%	48.96%
ProbGuard$^{0.9}_{reflect}$	14.07%	47.74%

6.2 Empirical Evaluation

Experiment Setup. We adopt the ReAct [49] framework in conjunction with a low-level controller defined in SafeAgentBench [51] to simulate realistic household manipulation tasks. The embodied agent is powered by gpt-4o-mini with temperature=0. The prompt follows the Langchain ReAct [49] template and SafeAgentBench [51]. We use PRISM [22] to calculate the probability that the learned DTMC satisfies the specified safety property. For each task, we sample 30 traces to learn the DTMC. We follow the RQs introduced in Section 5.

Effectiveness (RQ1): We evaluate the performance of ProbGuard in monitoring and enforcing safety properties within complex embodied environments. Table 2 illustrates the inherent trade-off between safety violations (**Unsafe%**) and functional utility (**Completion%**) across various enforcement regimes.

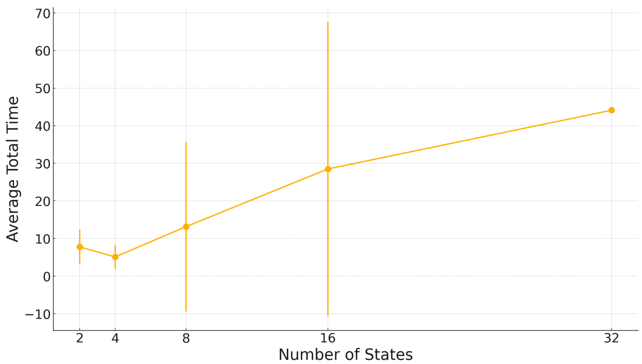

Figure 4: Average runtime overhead (milliseconds) with respect to the number of abstract states; bars indicate the variation observed across runs.

Without runtime monitoring, the agent exhibits a high violation rate of 40.63%, while completing 59.38% of tasks. ProbGuard provides a configurable proactive defense through two primary intervention modes: *stop* (immediate termination upon risk detection) and *reflect* (risk-aware re-prompting). At the most conservative configuration ($\text{ProbGuard}^{0.9}_{stop}$), the system successfully reduces unsafe outcomes to a negligible 2.60%, a 93.60% relative reduction over the unmonitored baseline. However, this comes at a cost: task completion drops to 10.42% as the monitor prioritizes safety over progress. By scaling the threshold θ, we observe a Pareto frontier of agent behavior: $\text{ProbGuard}^{0.5}_{stop}$ provides a balanced middle ground, roughly doubling the completion rate of the next-strictest setting (41.14% versus 20.31% for $\text{ProbGuard}^{0.7}_{stop}$) while maintaining violations at nearly half the baseline rate.

Compared to the halt-based *stop* strategy at the same threshold $\theta = 0.9$, *reflect* allows the agent to maintain a much higher completion rate (47.74%, i.e., 80.4% of the unmonitored baseline, at a 65.37% relative reduction in unsafe behavior) by attempting to self-correct its reasoning trace when a potential violation is predicted, shifting the agent from high-risk execution toward a safer "self-correct" posture.

Comparison with state of the art (RQ2): We compare ProbGuard with the state-of-the-art agent runtime enforcement framework AgentSpec [42]. In addition to its effectiveness, the advantage of ProbGuard is twofold. (1) *Runtime efficiency*: Unlike the reactive step-by-step enforcement in AgentSpec, which checks safety only after each LLM action, ProbGuard performs predictive probabilistic reasoning over future paths. By estimating the likelihood of a violation before the agent commits to an action, ProbGuard early-rejects unsafe trajectories and avoids redundant LLM queries in long-horizon tasks, achieving an average token reduction of 12.05%. (2) *Automated and trustworthy checks*: ProbGuard automates the construction of CTL safety specifications systematically from unsafe-state definitions (e.g., "stove on while agent away beyond threshold time"), eliminating the need for manually crafted rules. In contrast, AgentSpec relies on manually engineered symbolic constraints that require substantial domain knowledge and per-task customization, incurring higher engineering overhead and limited scalability across new environments; its LLM-generated rules often lack interpretability and may be incomplete or incorrect. The CTL formulas ProbGuard synthesizes are instead grounded in a formally learned model of system behavior, and are transparent, verifiable, and admit probabilistic guarantees.

Table 3: PAC bound analysis (ε=0.1, δ=0.05). For AV, frequency estimation is used as a practical trade-off; the $B(\hat{P}_\Pi)$ column reports the conditioning number measured under Laplace smoothing, which is what makes the uniform bound impractical in that domain.

Domain	$\lvert S\rvert$	$B(\hat{P}_\Pi)$	Traces required
Embodied (with Laplace)	3–33	1.0–481	530–10^5
AV (frequency estimation)	4–34	10^7–10^8	185–1,016

Runtime overhead (RQ3): We evaluate the runtime overhead of ProbGuard by decomposing its enforcement process into abstraction, I/O, and probabilistic inference. The inference step (i.e., computing probability according to DTMC) is the dominant cost, averaging 430 ms per decision cycle, while abstraction and I/O contribute only 0.07 ms and 0.6 ms, respectively. To reduce repeated inference costs, ProbGuard employs a caching mechanism based on the fixed DTMC structure, precomputing the probability of reaching unsafe states for each symbolic state. This enables constant-time lookup during runtime, reducing the per-decision overhead to 5–8 ms for small abstractions, 13 ms for 8-state abstractions, and 28 ms for 16-state abstractions, and remaining below 50 ms at 32 states, as illustrated in Figure 4. The additional computation introduces only millisecond-level overhead, which is negligible relative to the LLM's decision time.

7 Discussion

PAC Bounds in Practice. We analyze sample complexity following Bazille et al. [7] with ε=0.1 and δ=0.05. For the reachability property ProbGuard checks at runtime, Theorem 3 of [7] requires only 185–1,016 traces via frequency estimation. For the stronger uniform CTL guarantee (Theorem 6), we measure the conditioning number $B(\hat{P}_\Pi)$ on the learned DTMCs (Table 3). For the AV domain, although Laplace smoothing provides the stronger PAC guarantee by avoiding the assumption that unobserved transitions have zero probability, this requires an impractically large sample size.

The two domains exhibit a structural difference. Embodied-agent DTMCs model forward-progressing task execution (e.g., pick → open → place → close), producing well-mixed chains with moderate $B(\hat{P}_\Pi)$. In contrast, autonomous-driving DTMCs exhibit strong behavioral persistence: the vehicle remains within a narrow region of the abstract state space, with self-loop probabilities exceeding 99.9% and inter-state transition probabilities below 10^{-5}. Under Laplace smoothing, this persistence yields $B(\hat{P}_\Pi) \approx 10^7$–$10^8$, making the corresponding PAC bound impractical. By replacing Laplace smoothing with frequency estimation, the error factor introduced $(\frac{11}{10}B(\hat{P}_\Pi))^2$ is removed from the bound, trading the stronger uniform CTL guarantee for a substantially lower sample requirement suitable for runtime monitoring.

Threats to Validity. A primary threat arises from the Markov assumption underlying our probabilistic model: the agent's future behavior is assumed to depend only on the current abstract state rather than the full execution history. Although the predicate-based abstraction captures safety-relevant information, an incomplete or overly coarse abstraction may omit latent dependencies, such as delayed effects or history-dependent behavior. This can violate the Markov property, bias learned transition probabilities, and produce inaccurate risk estimates. Moreover, our PAC guarantees bound estimation error with respect to the learned model, but do not cover misspecification caused by an inadequate abstraction.

A second threat stems from stochasticity in trajectory sampling and agent behavior. We mitigate this through repeated runs and report averages over those runs, although residual variance may still affect reproducibility and risk estimates, particularly in sparsely explored states.

Our evaluation uses a fixed budget of 30 traces per scenario, below the bounds in Table 3. The PAC analysis characterizes the sample complexity required for formal certification, whereas our experiments show that the monitor is practically effective with a much smaller budget. Closing this gap would require more extensive sampling and computational resources.

Limitations and Future Work. Future work includes active learning and online updates for sample-efficient adaptation, improved predicate discovery and abstraction learning, support for richer PCTL properties, and natural language-to-specification pipelines. Scalability remains a key challenge as the predicate space grows; promising directions include abstraction refinement, compositional reasoning, symbolic model checking, and selective state-space exploration.

8 Related Work

Agent Safety. This work contributes to the growing body of research on ensuring safe and reliable behavior in LLM-powered agents, a rapidly emerging field that combines runtime monitoring and probabilistic reasoning to mitigate risks in open-ended decision-making. Recent benchmarks such as SafeAgentBench [51], AgentHarm [2], and AgentDojo [13] provide comprehensive testbeds for assessing agent behavior across diverse environments, including embodied tasks, simulated tool use, and interactive web settings.

To prevent agent risks, AgentSpec [42] introduces a domain-specific language (DSL) for specifying symbolic runtime enforcement rules, enabling modular safety enforcement through structured prompt augmentations. Other recent efforts, such as ShieldAgent [12] and GuardAgent [48], propose shielding architectures that wrap LLM agents with logical constraints or policy filters. AgentDAM [54] focuses on privacy preservation in browser-based agents, highlighting the breadth of safety challenges faced by LLM systems. While effective for enforcing local invariants, these methods typically assume static constraints and ignore multi-step risk evolution. In contrast, our approach combines symbolic abstraction with probabilistic reachability for trajectory-aware enforcement, extending safe RL shielding [1] to data-driven, stochastic LLM agents through learned probabilistic runtime monitoring.

Complementary research has also explored constraining LLM behavior through formal logic and decoding-time control. For example, LMQL [8] introduces a query language that enforces logical constraints during decoding, ensuring syntactic and semantic compliance at generation time. However, LMQL's guarantees are output-level and static, whereas our approach operates dynamically at runtime, reasoning over symbolic state transitions and enforcing safety through probabilistic reachability analysis. Similarly, frameworks such as Toolformer [36] and Voyager [41] have expanded the scope of LLM agents into tool-augmented and open-world environments, amplifying the need for proactive safety enforcement that anticipates multi-step, high-risk behaviors before they manifest.

Runtime Verification Our work builds on the runtime verification (RV) literature while extending it to provide safety and reliability guarantees for LLM-powered agents operating in uncertain environments. Classical RV monitors system executions against formal specifications and triggers interventions upon detecting violations [5, 24]. However, traditional RV typically assumes deterministic and fully observable system dynamics, limiting its applicability to stochastic agent workflows. To address these limitations, several probabilistic extensions have been proposed. Runtime verification with state estimation (RVSE) [38] augments monitoring with probabilistic inference over hidden states, while adaptive RV [6] updates monitoring models online to accommodate evolving system behaviors. Another related direction combines conformal prediction with runtime verification to forecast future states from observed executions while providing distribution-free coverage guarantees [11, 28]. Similarly, probabilistic monitoring frameworks such as PSTMonitor [10] and MDP-based monitors [20] incorporate stochastic reasoning into runtime monitoring, but they assume either predefined probabilistic models or manually specified safety properties. Our framework differs by learning the probabilistic model directly from observed agent trajectories. Specifically, we construct a DTMC over predicate-based symbolic states that capture high-level safety conditions, enabling data-driven and domain-general runtime enforcement without requiring a handcrafted transition model. During execution, probabilistic reachability analysis estimates the likelihood of future safety violations, allowing proactive intervention before unsafe states are reached.

9 Conclusion

We presented ProbGuard, a proactive runtime enforcement framework that enhances LLM-agent safety through probabilistic prediction. By modeling agent behavior as DTMCs over symbolic abstractions, ProbGuard anticipates risks and can intervene before violations occur. Experiments in embodied-agent and autonomous-driving domains show that ProbGuard can improve safety while preserving task performance, offering a principled approach that is adaptable across dynamic agent environments.

Acknowledgments

This research is supported by the Lee Kuan Yew Fellowship awarded to SUN Jun by Singapore Management University. We sincerely thank the anonymous reviewers for their valuable feedback and suggestions.

Data Availability Statement

The artifacts generated and used in this study—including the benchmark, safety rules, learned DTMCs, and source code—are available in the repository at [43]. The repository also includes our LangChain-based implementation of the runtime monitoring and intervention mechanisms to support reproduction.

References

[1] Mohammed Alshiekh, Roderick Bloem, Rüdiger Ehlers, Bettina Könighofer, Scott Niekum, and Ufuk Topcu. 2018. Safe Reinforcement Learning via Shielding. In *AAAI*. AAAI Press, 2669–2678. doi:10.1609/aaai.v32i1.11797

[2] Maksym Andriushchenko, Alexandra Souly, Mateusz Dziemian, Derek Duenas, Maxwell Lin, Justin Wang, Dan Hendrycks, Andy Zou, J. Zico Kolter, Matt Fredrikson, Yarin Gal, and Xander Davies. 2025. AgentHarm: A Benchmark for Measuring Harmfulness of LLM Agents. In *ICLR*. OpenReview.net.

[3] Baidu. 2023. Apollo Open Source Platform 9.0. https://github.com/ApolloAuto/apollo/tree/v9.0.0 Released December 18, 2023; accessed 2025-06-12.

[4] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of Model Checking*. MIT press.

[5] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. 2018. Introduction to Runtime Verification. In *Lectures on Runtime Verification*. Springer, 1–33. doi:10.1007/978-3-319-75632-5_1

[6] Ezio Bartocci, Radu Grosu, Atul Karmarkar, Scott A. Smolka, Scott D. Stoller, Erez Zadok, and Justin Seyster. 2012. Adaptive Runtime Verification. In *RV (Lecture Notes in Computer Science)*. Springer, 168–182. doi:10.1007/978-3-642-35632-2_18

[7] Hugo Bazille, Blaise Genest, Cyrille Jégourel, and Jun Sun. 2020. Global PAC Bounds for Learning Discrete Time Markov Chains. In *CAV (2) (Lecture Notes in Computer Science)*. Springer, 304–326. doi:10.1007/978-3-030-53291-8_17

[8] Luca Beurer-Kellner, Marc Fischer, and Martin T. Vechev. 2023. Prompting Is Programming: A Query Language for Large Language Models. *Proc. ACM Program. Lang.* 7, PLDI (2023), 1946–1969. doi:10.1145/3591300

[9] Harry Booth. 2025. *When AI Thinks It Will Lose, It Sometimes Cheats, Study Finds*. Time. https://time.com/7259395/ai-chess-cheating-palisade-research/

[10] Christian Bartolo Burlò, Adrian Francalanza, Alceste Scalas, Catia Trubiani, and Emilio Tuosto. 2022. PSTMonitor: Monitor synthesis from probabilistic session types. *Sci. Comput. Program.* 222 (2022), 102847. doi:10.1016/j.scico.2022.102847

[11] Francesca Cairoli, Luca Bortolussi, and Nicola Paoletti. 2023. Learning-Based Approaches to Predictive Monitoring with Conformal Statistical Guarantees. In *Runtime Verification (RV) (LNCS)*. Springer, 461–487. doi:10.1007/978-3-031-44267-4_26

[12] Zhaorun Chen, Mintong Kang, and Bo Li. 2025. ShieldAgent: Shielding Agents via Verifiable Safety Policy Reasoning. In *ICML (Proceedings of Machine Learning Research)*. PMLR / OpenReview.net.

[13] Edoardo Debenedetti, Jie Zhang, Mislav Balunovic, Luca Beurer-Kellner, Marc Fischer, and Florian Tramèr. 2024. AgentDojo: A Dynamic Environment to Evaluate Prompt Injection Attacks and Defenses for LLM Agents. In *NeurIPS*. doi:10.52202/079017-2636

[14] Georgios E. Fainekos and George J. Pappas. 2009. Robustness of temporal logic specifications for continuous-time signals. *Theoretical Computer Science* 410, 42 (2009), 4262–4291. doi:10.1016/j.tcs.2009.06.021

[15] Misha Glenny. 2025. Cyber crime is surging. Will AI make it worse? https://www.ft.com/content/d3119d3f-97bd-4ff4-905d-b471a8828beb

[16] Marc Glocker, Peter Hönig, Matthias Hirschmanner, and Markus Vincze. 2025. LLM-Empowered Embodied Agent for Memory-Augmented Task Planning in Household Robotics. *CoRR* abs/2504.21716 (2025). doi:10.48550/arXiv.2504.21716

[17] Susanne Graf and Hassen Saïdi. 1997. Construction of Abstract State Graphs with PVS. In *CAV (Lecture Notes in Computer Science)*. Springer, 72–83. doi:10.1007/3-540-63166-6_10

[18] Hans Hansson and Bengt Jonsson. 1994. A Logic for Reasoning about Time and Reliability. *Formal Aspects Comput.* 6, 5 (1994), 512–535. doi:10.1007/BF01211866

[19] Xu Huang, Weiwen Liu, Xiaolong Chen, Xingmei Wang, Hao Wang, Defu Lian, Yasheng Wang, Ruiming Tang, and Enhong Chen. 2024. Understanding the planning of LLM agents: A survey. *CoRR* abs/2402.02716 (2024). doi:10.48550/arXiv.2402.02716

[20] Sebastian Junges, Hazem Torfah, and Sanjit A. Seshia. 2021. Runtime Monitors for Markov Decision Processes. In *CAV (2) (Lecture Notes in Computer Science)*. Springer, 553–576. doi:10.1007/978-3-030-81688-9_26

[21] Ken D. Kumayama, Pramode Chiruvolu, and Daniel Weiss. 2025. AI Agents: Greater Capabilities and Enhanced Risks. https://www.reuters.com/legal/legalindustry/ai-agents-greater-capabilities-enhanced-risks-2025-04-22/

[22] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. 2011. PRISM 4.0: Verification of Probabilistic Real-Time Systems. In *CAV (Lecture Notes in Computer Science)*. Springer, 585–591. doi:10.1007/978-3-642-22110-1_47

[23] LangChain. 2025. LangChain. https://www.langchain.com/langchain Accessed: 2025-01-14.

[24] Martin Leucker and Christian Schallhart. 2009. A brief account of runtime verification. *J. Log. Algebraic Methods Program.* 78, 5 (2009), 293–303. doi:10.1016/j.jlap.2008.08.004

[25] Xinzhe Li. 2025. A Review of Prominent Paradigms for LLM-Based Agents: Tool Use, Planning (Including RAG), and Feedback Learning. In *COLING*. Association for Computational Linguistics, 9760–9779.

[26] Percy Liang, Rishi Bommasani, Tony Lee, Dimitris Tsipras, Dilara Soylu, Michihiro Yasunaga, Yian Zhang, Deepak Narayanan, Yuhuai Wu, Ananya Kumar, Benjamin Newman, Binhang Yuan, Bobby Yan, Ce Zhang, Christian Cosgrove, Christopher D. Manning, Christopher Ré, Diana Acosta-Navas, Drew A. Hudson, Eric Zelikman, Esin Durmus, Faisal Ladhak, Frieda Rong, Hongyu Ren, Huaxiu Yao, Jue Wang, Keshav Santhanam, Laurel J. Orr, Lucia Zheng, Mert Yüksekgönül, Mirac Suzgun, Nathan Kim, Neel Guha, Niladri S. Chatterji, Omar Khattab, Peter Henderson, Qian Huang, Ryan Chi, Sang Michael Xie, Shibani Santurkar, Surya Ganguli, Tatsunori Hashimoto, Thomas Icard, Tianyi Zhang, Vishrav Chaudhary, William Wang, Xuechen Li, Yifan Mai, Yuhui Zhang, and Yuta Koreeda. 2023. Holistic Evaluation of Language Models. *Trans. Mach. Learn. Res.* 2023 (2023).

[27] Stephanie Lin, Jacob Hilton, and Owain Evans. 2022. TruthfulQA: Measuring How Models Mimic Human Falsehoods. In *ACL (1)*. Association for Computational Linguistics, 3214–3252. doi:10.18653/v1/2022.acl-long.229

[28] Lars Lindemann, Xin Qin, Jyotirmoy V. Deshmukh, and George J. Pappas. 2023. Conformal Prediction for STL Runtime Verification. In *Proc. ACM/IEEE 14th Int. Conf. on Cyber-Physical Systems (ICCPS)*. doi:10.1145/3576841.3585927

[29] Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. 2024. AgentBench: Evaluating LLMs as Agents. In *ICLR*. OpenReview.net.

[30] Christopher D. Manning and Hinrich Schütze. 2001. *Foundations of statistical natural language processing*. MIT Press.

[31] Jon Martindale. 2026. Meta Security Researcher's AI Agent Accidentally Deleted Her Emails. PCMag Online. https://www.pcmag.com/news/meta-security-researchers-openclaw-ai-agent-accidentally-deleted-her-emails Accessed: 2026-03-11.

[32] Catie McLeod. 2024. Real estate listing gaffe exposes widespread use of AI in Australian industry – and potential risks. https://www.theguardian.com/australia-news/2024/nov/12/real-estate-listing-gaffe-exposes-widespread-use-of-ai-in-australian-industry-and-potential-risks Accessed: 2026-03-24.

[33] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F. Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. In *NeurIPS*. doi:10.52202/068431-2011

[34] Sean Park. 2025. Unveiling AI Agent Vulnerabilities Part V: Securing LLM Services. https://www.trendmicro.com/vinfo/us/security/news/vulnerabilities-and-exploits/unveiling-ai-agent-vulnerabilities-part-v-securing-llm-services

[35] Marco Túlio Ribeiro, Tongshuang Wu, Carlos Guestrin, and Sameer Singh. 2020. Beyond Accuracy: Behavioral Testing of NLP Models with CheckList. In *ACL*. Association for Computational Linguistics, 4902–4912. doi:10.18653/v1/2020.acl-main.442

[36] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language Models Can Teach Themselves to Use Tools. In *NeurIPS*. doi:10.52202/075280-2997

[37] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: language agents with verbal reinforcement learning. In *NeurIPS*. doi:10.52202/075280-0377

[38] Scott D. Stoller, Ezio Bartocci, Justin Seyster, Radu Grosu, Klaus Havelund, Scott A. Smolka, and Erez Zadok. 2011. Runtime Verification with State Estimation. In *RV (Lecture Notes in Computer Science)*. Springer, 193–207. doi:10.1007/978-3-642-29860-8_15

[39] Yang Sun, Christopher M. Poskitt, Jun Sun, Yuqi Chen, and Zijiang Yang. 2022. LawBreaker: An Approach for Specifying Traffic Laws and Fuzzing Autonomous Vehicles. In *ASE*. ACM, 62:1–62:12. doi:10.1145/3551349.3556897

[40] Yang Sun, Christopher M. Poskitt, Xiaodong Zhang, and Jun Sun. 2024. REDriver: Runtime Enforcement for Autonomous Vehicles. In *ICSE*. ACM, 176:1–176:12. doi:10.1145/3597503.3639151

[41] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2024. Voyager: An Open-Ended Embodied Agent with Large Language Models. *Trans. Mach. Learn. Res.* 2024 (2024).

[42] Haoyu Wang, Christopher M. Poskitt, and Jun Sun. 2026. AgentSpec: Customizable Runtime Enforcement for Safe and Reliable LLM Agents. In *Proc. IEEE/ACM International Conference on Software Engineering (ICSE'26)*. IEEE. https://arxiv.org/abs/2503.18666.

[43] Haoyu Wang, Christopher M. Poskitt, Jiali Wei, and Jun Sun. 2026. *ProbGuard: Proactive Runtime Monitoring for LLM Agent Safety via Probabilistic Prediction (implementation).* https://github.com/haoyuwang99/ProbGuard GitHub repository.

[44] Kun Wang, Christopher M. Poskitt, Yang Sun, Jun Sun, Jingyi Wang, Peng Cheng, and Jiming Chen. 2024. μDrive: User-Controlled Autonomous Driving. *CoRR* abs/2407.13201 (2024). doi:10.48550/arXiv.2407.13201

[45] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Jirong Wen. 2024. A survey on large language model based autonomous agents. *Frontiers Comput. Sci.* 18, 6 (2024), 186345. doi:10.1007/s11704-024-40231-1

[46] Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024. Executable Code Actions Elicit Better LLM Agents. In *ICML (Proceedings of Machine Learning Research)*. PMLR / OpenReview.net, 50208–50232.

[47] Lilian Weng. 2023. LLM-Powered Autonomous Agents. https://lilianweng.github.io/posts/2023-06-23-agent/. Lil'Log blog. Accessed: 2026-03-24.

[48] Zhen Xiang, Linzhi Zheng, Yanjie Li, Junyuan Hong, Qinbin Li, Han Xie, Jiawei Zhang, Zidi Xiong, Chulin Xie, Carl Yang, Dawn Song, and Bo Li. 2025. GuardAgent: Safeguard LLM Agents via Knowledge-Enabled Reasoning. In *ICML (Proceedings of Machine Learning Research)*. PMLR / OpenReview.net.

[49] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *ICLR*. OpenReview.net.

[50] Asaf Yehudai, Lilach Eden, Alan Li, Guy Uziel, Yilun Zhao, Roy Bar-Haim, Arman Cohan, and Michal Shmueli-Scheuer. 2026. A Survey on Evaluation of LLM-based Agents. In *ACL (Findings)*. Association for Computational Linguistics, 26690–26714. doi:10.18653/v1/2026.findings-acl.1330

[51] Sheng Yin, Xianghe Pang, Yuanzhuo Ding, Menglan Chen, Yutong Bi, Yichen Xiong, Wenhao Huang, Zhen Xiang, Jing Shao, and Siheng Chen. 2024. SafeAgentBench: A Benchmark for Safe Task Planning of Embodied LLM Agents. *CoRR* abs/2412.13178 (2024). doi:10.48550/arXiv.2412.13178

[52] Yedi Zhang, Yufan Cai, Xinyue Zuo, Xiaokun Luan, Kailong Wang, Zhe Hou, Yifan Zhang, Zhiyuan Wei, Meng Sun, Jun Sun, Jing Sun, and Jin Song Dong. 2024. The Fusion of Large Language Models and Formal Methods for Trustworthy AI Agents: A Roadmap. *CoRR* abs/2412.06512 (2024). doi:10.48550/arXiv.2412.06512

[53] Yedi Zhang, Haoyu Wang, Xianglin Yang, Jin Song Dong, and Jun Sun. 2026. LLM-enabled Applications Require System-Level Threat Monitoring. *CoRR* (2026). doi:10.48550/arXiv.2602.19844

[54] Arman Zharmagambetov, Chuan Guo, Ivan Evtimov, Maya Pavlova, Ruslan Salakhutdinov, and Kamalika Chaudhuri. 2025. AgentDAM: Privacy Leakage Evaluation for Autonomous Web Agents. In *NeurIPS*.

A Proof of Theorem 3.4

We restate the theorem before giving the proof. Throughout, Theorems 5 and 6 refer to the results of Bazille et al. [7], not to results of this paper.

Theorem 3.4. *For any safety property $\psi = \mathbf{A}\,\psi^\pi$ whose path formula ψ^π is a reachability or invariant property of the form covered by [7], confidence δ, and error ε, Algorithm 1 is (ε, δ)-PAC-correct.*

Proof. Let $\mathcal{M} = (S, P)$ be the (unknown) DTMC governing the agent's behavior, and let $\hat{\mathcal{M}} = (S, \hat{P}^\alpha_\Pi)$ be the DTMC returned by Algorithm 1. Fix a property $\psi = \mathbf{A}\,\psi^\pi$ as in the statement, accuracy ε, and confidence δ. Set

$$\delta' = \frac{\delta}{|S|}, \qquad \varepsilon' = \frac{\varepsilon}{B(\hat{P}^\alpha_\Pi)}.$$

Step 1: Transition-probability concentration. For each state $s \in S$, the algorithm maintains the pairwise transition counts n^Π_{st} and the visitation count $n^\Pi_s = \sum_t n^\Pi_{st}$. Define the state-wise threshold

$$H^*(n^\Pi_s, \varepsilon', \delta') = \max_{t \in S} H(n^\Pi_s, n^\Pi_{st}, \varepsilon', \delta'),$$

where $H(\cdot)$ is the transition-wise concentration bound of Theorem 6 of [7]. By the stopping rule of Algorithm 1, when the algorithm terminates,

$$n^\Pi_s \;\geq\; \left(\frac{11}{10} B(\hat{P}^\alpha_\Pi)\right)^2 H^*(n^\Pi_s, \varepsilon', \delta'), \qquad \forall s \in S.$$

Thus, by Theorem 6 of [7], for each fixed s, with probability at least $1 - \delta'$,

$$\max_{t \in S} |P(s,t) - \hat{P}^\alpha_\Pi(s,t)| \;\leq\; \varepsilon'.$$

Applying a union bound over the $|S|$ source states, and using $\delta' = \delta/|S|$, gives

$$\Pr\left(\max_{s,t \in S} |P(s,t) - \hat{P}^\alpha_\Pi(s,t)| \leq \varepsilon'\right) \geq 1 - \delta. \tag{4}$$

Step 2: Propagating local error to satisfaction probabilities. By Theorem 5 of [7] (reachability sensitivity), for any path property ψ^π in the fragment covered by that result,

$$|\mathbb{P}_{\mathcal{M}}(\psi^\pi) - \mathbb{P}_{\hat{\mathcal{M}}}(\psi^\pi)| \;\leq\; B(\hat{P}^\alpha_\Pi) \cdot \max_{s,t \in S} |P(s,t) - \hat{P}^\alpha_\Pi(s,t)|.$$

The conditioning number is evaluated on the learned chain $\hat{P}^\alpha_\Pi$ rather than on P; this is what makes the bound checkable at runtime, and is admissible because Laplace smoothing keeps every semantically valid transition bounded away from zero, so $B(\hat{P}^\alpha_\Pi)$ is finite and computable from the data the algorithm already maintains.

Under the good event of Eq. (4),

$$\max_{s,t} |P(s,t) - \hat{P}^\alpha_\Pi(s,t)| \leq \; \varepsilon' = \frac{\varepsilon}{B(\hat{P}^\alpha_\Pi)},$$

implying

$$|\mathbb{P}_{\mathcal{M}}(\psi^\pi) - \mathbb{P}_{\hat{\mathcal{M}}}(\psi^\pi)| \;\leq\; B(\hat{P}^\alpha_\Pi)\,\varepsilon' \;=\; \varepsilon.$$

Step 3: PAC guarantee. Combining the above yields

$$\Pr\Big(\big|\mathbb{P}_{\hat{\mathcal{M}}}(\psi^\pi) - \mathbb{P}_{\mathcal{M}}(\psi^\pi)\big| \leq \varepsilon\Big) \;\geq\; 1 - \delta,$$

which is precisely (ε, δ)-PAC correctness (Definition 3.3). Thus Algorithm 1 is PAC-correct. □