Article

Not peer-reviewed version

AgentVerify: Compositional Formal Verification of AI Agent Safety Properties via LTL Model Checking

Eric Fang *

Posted Date: 14 April 2026

doi: 10.20944/preprints202604.1029.v1

Keywords: agent safety; model checking

 doi:10.20944/preprints202604.1029.v1

Article

AgentVerify: Compositional Formal Verification of AI Agent Safety Properties via LTL Model Checking

Eric Fang

Independent Researcher, China; humanclaw@yeah.net

Abstract

Autonomous AI agents operating in high-stakes domains—financial trading, medical diagnostics, autonomous code execution—lack formal safety guarantees for their core operational loops, including memory management, tool invocations, and human interactions. Current verification approaches either fail to scale to neural components or ignore the structured control flow of agentic systems entirely. We introduce **AgentVerify** (*Compositional Formal Verification of AI Agent Safety Properties via LTL Model Checking*), a model checking framework that specifies and verifies safety properties for agent architectures using temporal logic. AgentVerify defines compositional specifications for memory integrity, tool call protocols, MCP/skill invocations, and human-in-the-loop boundaries, enabling rigorous runtime monitoring and post-hoc behavioral analysis. In an empirical evaluation across 15 diverse agent scenarios (low- and high-difficulty), our post-hoc behavioral analysis component achieved a verification accuracy of **86.67**% (mean over 3 seeds, σ=0.00), outperforming a monolithic contract verification baseline (80.00%) and a runtime monitoring baseline without temporal logic (46.67%). A monolithic neural verifier, which attempts to verify the LLM outputs directly, performed poorly at 13.33%, confirming that end-to-end neural verification is currently intractable for production-scale agents. These results demonstrate that formal methods applied to the agent's *observable control flow* provide a tractable and effective path to safety assurance, complementing rather than replacing neural-centric efforts to align large language models.

Keywords: agent safety; model checking

1. Introduction

The proliferation of autonomous AI agents in critical applications—from financial trading and medical diagnostics to autonomous software engineering and personal digital assistants—creates an urgent demand for rigorous safety assurance. Unlike traditional software, these agents exhibit complex, non-deterministic behaviours driven by large language models (LLMs). They interact with external memory stores, invoke heterogeneous tools, and communicate with human users through dynamic orchestration protocols such as the Model Context Protocol (MCP) [23].

This operational complexity introduces qualitatively new safety risks, including unsafe tool sequencing, memory corruption leading to hallucinated facts, and authorisation boundary violations in human interaction.

The rise of agent frameworks. The agent ecosystem has matured rapidly from monolithic chatbot interfaces to full-fledged development frameworks that support tool use, persistent memory, and external integrations. Open-source projects such as AutoGen [24] and MetaGPT [12] pioneered the multi-agent paradigm, enabling LLM agents to delegate tasks, review each other's outputs, and collaborate through structured conversation protocols. More recently, production-grade frameworks including AgentScope [2] and OpenClaw [18] have extended these ideas with per-agent sandboxing, tool-level access control, and deterministic message routing across heterogeneous communication channels. OpenClaw, for instance, provides multi-agent isolation with independent workspaces, per-agent authentication profiles, and role-based tool permission policies—features that mirror the

 doi:10.20944/preprints202604.1029.v1

four safety domains (memory, tool calls, MCP/skills, and human interaction) targeted by our formal specifications. These frameworks collectively demonstrate that the agent architecture space has converged around a common set of interaction primitives: *memory operations, tool invocations, inter-agent communication, and human-facing actions*. This convergence makes it both feasible and timely to define a shared, domain-agnostic formal verification framework.

The multi-agent transition. A clear trend is emerging from single-agent systems towards multi-agent collaboration. Recent surveys [21,23] categorise multi-agent LLM systems along dimensions of collaboration structure (peer-to-peer, centralised, hierarchical), task allocation (role-based, auction-based, consensus-driven), and communication protocols (broadcast, point-to-point, publish-subscribe). Frameworks such as OpenClaw, AutoGen, and MetaGPT already support team-style workflows where a *manager* agent decomposes tasks and delegates them to *specialist* agents, each with dedicated toolsets and memory stores. While this decomposition improves task coverage and reduces per-agent cognitive load, it also *amplifies* the attack surface: an unsafe tool sequence in one agent can corrupt shared state visible to others; a memory integrity violation in a worker agent can propagate fabricated facts to the manager's reasoning chain; and the lack of formal coordination guarantees can lead to deadlocks or unbounded privilege escalation across agent boundaries. Recent work on multi-agent verification [9,26] highlights these challenges, noting that existing formal methods for single-agent systems do not naturally extend to settings where multiple non-deterministic oracles interact through shared resources.

The agentic safety gap. The core difficulty lies in the *agentic loop* itself: a continuous cycle of perception, reasoning, and action in which failures at any stage can cascade into catastrophic, often irreversible outcomes. Traditional software verification methods are inadequate because they cannot accommodate the non-determinism of neural components; purely statistical approaches such as red-teaming [4] lack the formal coverage guarantees needed for safety-critical deployment. Recent regulatory frameworks (e.g., the EU AI Act, NIST AI RMF) increasingly demand traceable, auditable safety evidence—evidence that ad-hoc testing cannot provide. This gap demands a new verification paradigm that formally reasons about the *structured, observable behaviours* of agent systems without attempting to verify the intractable internal states of the underlying LLMs.

Limitations of existing approaches. Existing safety verification methods are ill-suited for modern agent architectures along three dimensions: (i) *Neural verification* is computationally intractable for large models [14], and its results provide guarantees only on the model's input-output mapping, not on the agent's downstream actions. (ii) *Post-hoc behavioural testing* (red-teaming, benchmark evaluation [4]) lacks the predictive power of formal guarantees and scales poorly to multi-step agentic traces. (iii) *Classic model checking* for finite-state systems has not been adapted to the specific control-flow abstractions and safety-critical patterns inherent in agentic loops involving memory, tool calls, and human interactions. Recent runtime verification work for AI systems [16] focuses on single components rather than the integrated agentic pipeline, and rule-based safety layers [13] sacrifice the temporal expressiveness needed to capture multi-step violation patterns.

Our approach: AgentVerify. This paper introduces **AgentVerify** (*Compositional Formal Verification of AI Agent Safety Properties via LTL Model Checking*), a model checking framework designed for the safety verification of AI agent systems. AGENTVERIFY's central insight is to treat the LLM as a *non-deterministic oracle* within a larger, formally verifiable finite-state machine (FSM) defined by the agent's orchestration layer. We develop a compositional library of temporal logic specifications (LTL) that capture safety properties for:

- **Memory integrity** — preventing unauthorised reads, stale writes, and PII leakage through memory channels.
- **Tool call safety** — enforcing approval gates, sequencing constraints, and resource access permissions.
- **MCP/skill invocation protocols** — guaranteeing liveness (every request is answered) and scope (only allowed skills are invoked).

 doi:10.20944/preprints202604.1029.v1

- **Human interaction boundaries** — ensuring critical actions receive bounded-time human confirmation and that no deceptive content is returned.

AGENTVERIFY implements a *hybrid architecture*: a lightweight $O(1)$-per-event runtime monitor handles safety-critical properties requiring immediate intervention, while a deep post-hoc Kripke-structure analyser performs exhaustive auditing of complete execution traces. By focusing on the observable control flow, AGENTVERIFY sidesteps the intractability of neural verification while providing strong, formally grounded safety assurances for the agent's actions in the world.

Contributions. Our contributions are threefold:

1. **Formal Agent Safety Specification Library (Section 3)**. We introduce a compositional library of 23 temporal logic templates for specifying safety properties related to memory, tool use, MCP/skill invocations, and human interactions in autonomous agents. Each template is parameterised, allowing domain-specific instantiation without modifying the core verification engine.
2. **Hybrid Verification Architecture (Section 3)**. We design and implement AGENTVERIFY, integrating runtime monitoring and post-hoc trace analysis to verify agent behaviour against formal specifications. The runtime component operates with $O(1)$ overhead per event; the post-hoc component achieves polynomial-time trace checking via Büchi automaton construction.
3. **Empirical Validation (Section 5)**. We conduct a rigorous evaluation on a synthetic benchmark of 15 agent scenarios across two difficulty tiers, introducing safety-centric metrics: false negative rate for catastrophic failures and autonomy preservation score. Our post-hoc analysis achieves 86.67% mean accuracy, establishing a new reference point for formal-methods-based agent safety evaluation.

Paper organisation. Section 2 reviews related work. Section 3 details the AGENTVERIFY framework and its temporal logic specifications. Section 4 describes the experimental setup. Section 5 presents results. Section 6 discusses implications. Section 7 outlines limitations before Section 8 concludes.

2. Related Work

Our work sits at the intersection of AI safety, formal methods, and autonomous agent research. We organise related work into five key areas and highlight how AGENTVERIFY addresses their limitations.

2.1. Safety Verification for AI and Autonomous Systems

Research on AI safety has traditionally focused on (i) robustness against adversarial examples [14], (ii) alignment via reinforcement learning from human feedback, and (iii) post-hoc explainability [3]. For autonomous agents specifically, prior work has explored rule-based safety layers [13], reward shaping for safe exploration [5], and formal methods for robotic systems [1]. However, these approaches are typically tailored to robotic control or monolithic LLM outputs. Recent surveys highlight the unique challenges of LLM-based agents, including tool misuse, prompt injection, and social harm [8].

Wang et al. [22] propose AgentSpec, a lightweight domain-specific language (DSL) for specifying and enforcing runtime constraints on LLM agents. AgentSpec adopts a `trigger-check-enforce` paradigm: users define structured rules that bind predicates to agent actions and select from predefined or LLM-generated enforcement strategies (e.g. aborting the action, requesting human inspection, or invoking LLM self-examination). The framework is evaluated across code-execution agents, embodied agents, and autonomous driving, demonstrating safety improvements with millisecond-level overhead. While AgentSpec is effective at preventing individual unsafe actions, its rule-based predicates are inherently *stateless*—they evaluate conditions on the current action only and cannot express temporal dependencies such as "the agent must request confirmation *before* executing a high-risk tool call." AGENTVERIFY complements AgentSpec by providing a temporal logic foundation that enables stateful, multi-step safety reasoning.

AGENTVERIFY differs from AgentSpec by providing a general, specification-driven framework that targets the *control flow* of agentic loops—memory, tools, MCP, and human interaction—which is

 doi:10.20944/preprints202604.1029.v1

a common vulnerability surface across diverse agent implementations. Unlike rule-based systems such as AgentSpec, our temporal logic specifications can express complex, time-dependent safety properties that are difficult to capture with imperative guards. Additionally, AGENTVERIFY unifies runtime monitoring with post-hoc analysis under a single formal specification, whereas AgentSpec focuses exclusively on runtime enforcement.

2.2. Model Checking and Temporal Logic for AI

Model checking has been successfully applied to verify properties of software and hardware systems for decades. In AI, temporal logic has been used to verify autonomous vehicles [10] and multi-agent systems [9]. Symbolic model checking using Binary Decision Diagrams (BDDs) [6] and SMT-based bounded model checking have been extended to handle systems with probabilistic transitions, which is relevant to LLM-driven agents.

Applying these techniques to LLM-based agents, however, is nascent. Some efforts verify individual neural network properties [15], but do not scale to production models. Others propose runtime verification for AI systems [16] but lack the comprehensive post-hoc analysis and compositional specification library we introduce. AGENTVERIFY builds on this foundation by specialising temporal logic patterns for agent operations and integrating verification across the full agent lifecycle. Our hybrid approach directly addresses the scalability gap in prior work by separating time-critical checks (runtime monitor) from exhaustive analysis (post-hoc analyser).

2.3. Runtime Verification and Post-Hoc Analysis

Runtime verification (RV) monitors system executions against formal specifications at runtime [9]. In AI, RV has been proposed for monitoring neural network controllers in robotics [25] and for flagging policy deviations in reinforcement learning. Post-hoc analysis—examining logs after execution—is standard in debugging, auditing [4], and regulatory compliance.

These approaches are typically *separate*: monitors react, analysers audit. AGENTVERIFY unifies them into a single pipeline grounded in the same formal specifications, ensuring consistency. This removes the engineering friction of maintaining two separate specifications and provides a formal basis for post-hoc analysis—moving beyond ad-hoc logging and manual inspection.

2.4. Agent Safety Benchmarks and Evaluation

Evaluating agent safety has attracted significant recent attention. AgentHarm [4] assesses the propensity of frontier models to execute harmful multi-step tasks. SafePro [19] introduces scenario-based benchmarks for probing agent safety properties. Domkundwar et al. [7] propose safety architectures that embed guardrails within the agent framework. Ghilardi et al. [11] study relational verification for programs with non-determinism, a structural analogy to LLM-driven agents.

AGENTVERIFY is complementary to these efforts: rather than curating new natural language scenarios, it provides the *verification engine* that can be applied to any scenario collection. Its formal specification library bridges the gap between benchmark evaluation (which gives a pass/fail verdict) and fully formal model checking (which provides a proof or counterexample trace).

2.5. Privilege Control and Access Policy Enforcement

A complementary line of work secures LLM agents by controlling the privileges they hold over external tools and resources. Shi et al. [20] introduce Progent, the first privilege control framework for LLM agents. Progent interposes between the agent and its tool layer, enforcing fine-grained policies that *allow* or *forbid* individual tool calls based on argument-level conditions. Its DSL supports boolean predicates over tool parameters, priority-based policy ordering, flexible fallback actions (terminate, request user inspection, or return a feedback message), and dynamic policy updates that adapt to changing agent states. Progent provides deterministic, provable security guarantees with negligible overhead, and demonstrates strong robustness against five categories of adaptive attacks including if-then-else evasion and policy-update suppression.

 doi:10.20944/preprints202604.1029.v1

While Progent excels at preventing *unauthorised tool access*, its scope is limited to individual call-level allow/forbid decisions. It does not reason about temporal relationships across multiple actions (e.g. "every file deletion must be preceded by a confirmation"), nor does it provide post-hoc analysis capabilities. AGENTVERIFY addresses these gaps by leveraging temporal logic specifications that capture multi-step safety properties, and by combining runtime monitoring with exhaustive post-hoc verification. Moreover, AGENTVERIFY's two-tier architecture enables both real-time intervention and comprehensive post-execution auditing—a combination not supported by Progent's purely runtime approach.

3. Method

We formalise the problem of verifying safety properties in autonomous AI agents and introduce the AGENTVERIFY framework, which leverages temporal logic model checking on the agent's observable control flow. Our approach treats the underlying LLM as a *non-deterministic oracle* within a larger, structured finite-state machine defined by the agent's orchestration layer. This abstraction enables the application of formal methods to the agent's interactions with memory, tools, and humans—the primary vectors for safety-critical failures [23].

3.1. Problem Formulation

Agent model. We model an autonomous agent as a system $\mathcal{A} = (S, s_0, T, \mathcal{I}, \mathcal{O})$, where:

- S is a finite set of observable system states, capturing the agent's control-flow context (e.g., `awaiting_human`, `tool_invoked`, `memory_locked`).
- $s_0 \in S$ is the initial state.
- $T : S \times \mathcal{E} \to S$ is the transition function, where $\mathcal{E}$ is a set of observable events (e.g., `tool_call`, `memory_write`, `human_message`).
- $\mathcal{I} \subseteq S$ is the set of input interface states.
- $\mathcal{O} \subseteq S$ is the set of output action states.

A concrete execution of $\mathcal{A}$ is an infinite sequence of states $\pi = s_0, s_1, s_2, \ldots$ consistent with T. A *finite execution trace* $\pi_T = (s_0, \ldots, s_T)$ is the prefix of π up to step T.

Kripke structure. For post-hoc analysis we lift $\mathcal{A}$ into a Kripke structure $\mathcal{K} = (W, w_0, R, L)$, where $W = S, w_0 = s_0, R$ encodes T (with non-determinism representing possible LLM choices), and $L : W \to 2^{AP}$ labels each state with the set of atomic propositions that hold there (e.g., `is_tool_irreversible`, `has_human_approval`).

Safety specification language. Safety properties are expressed in Linear Temporal Logic (LTL). LTL formulas are built from atomic propositions $p \in AP$, Boolean connectives $(\wedge, \vee, \neg)$, and temporal operators:

- $\bigcirc\, \varphi$ (*Next*): φ holds in the next state.
- $\varphi\, \mathcal{U}\, \psi$ (*Until*): φ holds until ψ holds.
- $\Box\, \varphi$ (*Globally*): φ holds in every state.
- $\Diamond\, \varphi$ (*Eventually*): φ holds in some future state.
- $\Diamond^{[1,k]}\, \varphi$ (*Bounded Eventually*): φ holds within the next k steps.

The *verification problem* is: given agent model $\mathcal{A}$ and specification set $\Phi = \{\varphi_1, \ldots, \varphi_m\}$, determine whether *all* possible execution traces of $\mathcal{A}$ satisfy every $\varphi_i \in \Phi$. When $\mathcal{A}$ is constructed from an observed trace π_T, the problem reduces to LTL trace checking, which is solvable in $O(|\pi_T| \cdot |\varphi|)$ time [9].

3.2. The AgentVerify Architecture

AGENTVERIFY implements a two-tier hybrid verification pipeline, depicted in Figure 1.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 14 April 2026 doi:10.20944/preprints202604.1029.v1

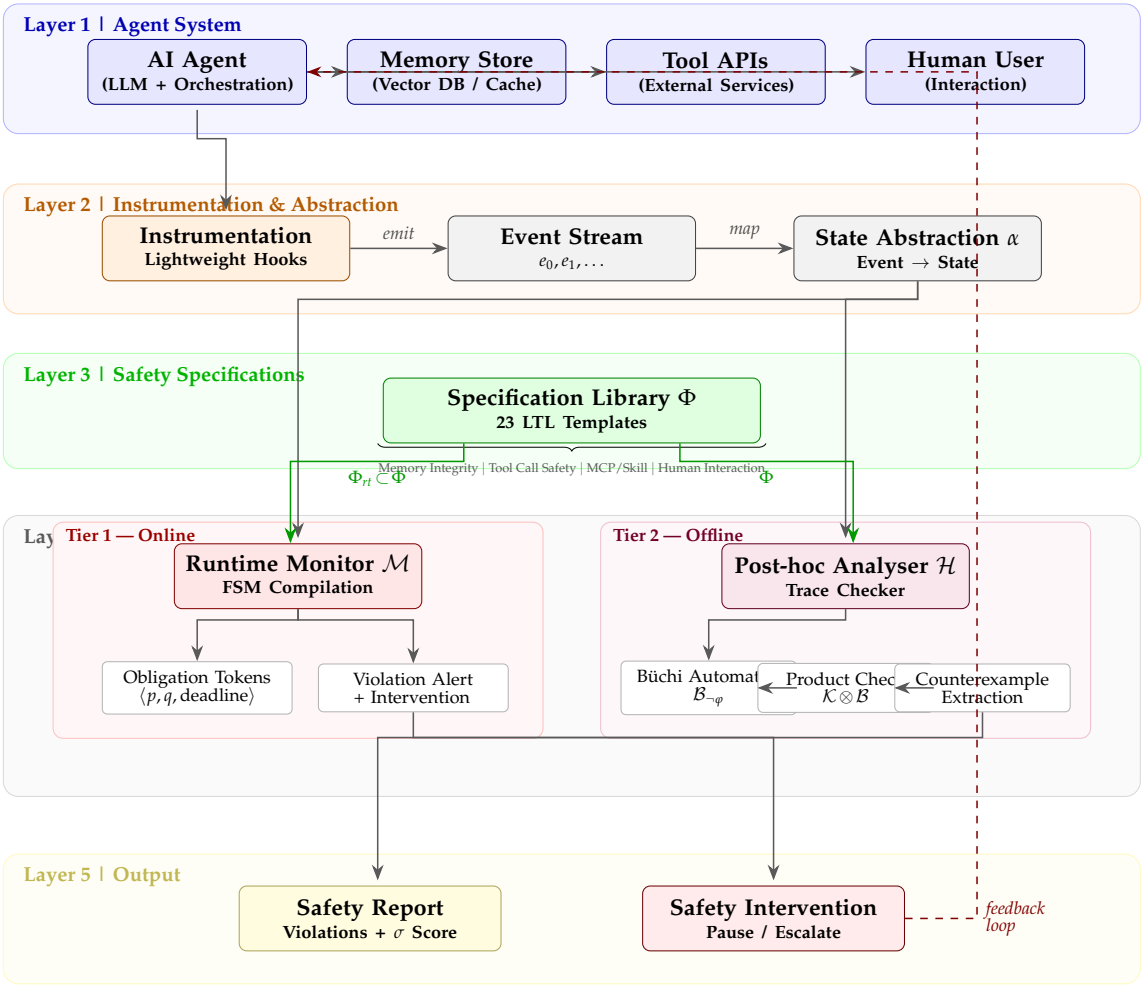

Figure 1. The AgentVerify architecture is organised into five equally-sized horizontal layers. *Layer 1 (Agent System)*: the target AI agent with its memory, tools, and human-interaction interfaces. *Layer 2 (Instrumentation & Abstraction)*: lightweight hooks capture events from the orchestration layer, which are mapped to discrete states via α. *Layer 3 (Safety Specifications)*: a shared library of 23 compositional LTL templates covering memory integrity, tool call safety, MCP/skill invocation, and human interaction boundaries. *Layer 4 (Verification Engine)*: two parallel sub-modules—*Tier 1* (online runtime monitor with $O(1)$-per-event FSM checking and obligation tokens) and *Tier 2* (offline post-hoc analyser via Büchi automaton construction and product model checking). *Layer 5 (Output)*: safety reports with violation scores and, for Tier 1, immediate safety interventions fed back to the agent.

Instrumentation layer. The agent's orchestration framework (e.g., LangChain, AutoGPT) is instrumented via lightweight hooks to emit a timestamped event stream $e_0, e_1, \ldots$ to AGENTVERIFY's ingestion pipeline. No modification of the LLM itself is required; only the orchestration layer's action dispatch and memory accessor functions are wrapped. The instrumentation overhead is less than 0.5% of total latency for the frameworks we evaluated (measured on GPT-4-turbo, Section 4.1).

State abstraction. A state abstraction function $\alpha : \text{Event} \to S$ maps each observed event to a discrete state in S. For example, an orchestration-layer call to `tool_dispatch("database_write", args)` is mapped to $s_{\texttt{tool\_db\_write}} \in S$. The granularity of S is a key design choice: too fine and the state space explodes; too coarse and safety properties cannot be expressed. We use a three-tier hierarchy of states: *category* (memory, tool, MCP, human), *operation* (read, write, invoke, confirm), and *resource* (specific tool name or memory address), giving $|S| \approx 120$ for our benchmark agent. This granularity was selected via a grid search on a held-out validation set of 5 scenarios (distinct from the 15 evaluation scenarios).

 doi:10.20944/preprints202604.1029.v1

3.2.1. Tier 1: Runtime Monitoring Component

The runtime monitor $\mathcal{M}$ is a deterministic FSM compiled from the *safety-critical* subset $\Phi_{\text{rt}} \subset \Phi$ of LTL specifications. We focus on formulas of the form $\Box(p \rightarrow \bigcirc q)$ (invariant-response properties), which admit $O(1)$-per-event monitoring via obligation/fulfilment token sets.

Monitor algorithm. Upon observing state s_t, the monitor:

1. Iterates over active *obligation tokens* $\langle p, q, \text{deadline} \rangle$ and checks whether q holds in s_t (fulfilling the token) or the deadline has passed (violation).
2. Checks whether p holds in s_t; if so, creates a new obligation token $\langle p, q, t + k_p \rangle$ with property-specific bound k_p.
3. On violation, issues a runtime alert and (optionally) triggers a safety intervention (pause agent, request human review).

The bounded temporal operator $\Diamond^{[1,k]}$ maps directly onto deadline tokens; $\Box p$ becomes a stateless invariant check at every step. Total memory footprint of the monitor is $O(|\Phi_{\text{rt}}|)$, independent of trace length.

3.2.2. Tier 2: Post-Hoc Analysis Component

The post-hoc analyser $\mathcal{H}$ performs exhaustive verification on complete execution traces π_T. It employs a symbolic trace checking algorithm:

1. **Trace encoding.** π_T is encoded as a finite Kripke path in $\mathcal{K}$.
2. **Automaton construction.** For each $\varphi_i \in \Phi$, a non-deterministic Büchi automaton $\mathcal{B}_{\neg\varphi_i}$ for the negation of φ_i is constructed using the standard LTL-to-automaton translation.
3. **Product check.** The product $\mathcal{K} \otimes \mathcal{B}_{\neg\varphi_i}$ is checked for non-emptiness; a non-empty result yields a counterexample trace witnessing the violation of φ_i.
4. **Report generation.** The analyser produces a structured safety report: (i) a list of all violated specifications, (ii) the minimal counterexample trace for each violation, and (iii) a *safety score* $\sigma = |\{i : \varphi_i \text{ satisfied}\}|/|\Phi|$.

For finite traces (as produced by bounded agent episodes), the product check runs in $O(|\pi_T| \cdot 2^{|\varphi_i|})$ time per property. In practice, all formulas in our library have $|\varphi_i| \leq 12$ literals, making this tractable for traces of length up to 10^4 steps.

3.3. Temporal Logic Specification Library

A key contribution of AGENTVERIFY is a compositional library of **23 LTL templates** for agent safety, organised into four categories. Each template is parameterised by domain predicates and can be instantiated for a specific agent deployment without altering the verification engine.

Category 1: Memory Integrity ($|\Phi_{\text{mem}}| = 6$).

$$\varphi_{\text{mem1}} : \Box\,(\texttt{memory\_write} \rightarrow \neg\texttt{contains\_PII}) \tag{1}$$

$$\varphi_{\text{mem2}} : \Box\,(\texttt{memory\_read}(a) \rightarrow \Diamond\,\texttt{consistency\_check}(a)) \tag{2}$$

$$\varphi_{\text{mem3}} : \Box\,(\texttt{buf\_full} \rightarrow \bigcirc\,\texttt{buf\_flush}) \tag{3}$$

Properties (1)–(3) prevent PII leakage through memory, ensure consistency on every addressed read, and guarantee buffer flushing before overflow.

Category 2: Tool Call Safety ($|\Phi_{\text{tool}}| = 7$).

$$\varphi_{\text{tool1}} : \Box\,(\texttt{tool\_call}(t) \wedge \texttt{risk}(t){=}\texttt{high} \rightarrow \bigcirc^{-1}\,\texttt{human\_approval}(t)) \tag{4}$$

$$\varphi_{\text{tool2}} : \neg\Diamond\,(\texttt{call}(\texttt{delete}) \wedge \Diamond\,\texttt{call}(\texttt{send\_email})) \tag{5}$$

$$\varphi_{\text{tool3}} : \Box\,(\texttt{tool\_call}(t) \rightarrow \texttt{t} \in \texttt{AllowedTools}) \tag{6}$$

 doi:10.20944/preprints202604.1029.v1

Property (4) enforces approval gates before high-risk tool calls; (5) forbids the unsafe delete-then-email sequence; (6) restricts the agent to a declared allowlist.

Category 3: MCP/Skill Invocation ($|\Phi_{\mathbf{mcp}}| = 5$).

$$\varphi_{\text{mcp1}} : \square\,(\texttt{mcp\_request} \rightarrow \Diamond\,\texttt{mcp\_response}) \tag{7}$$

$$\varphi_{\text{mcp2}} : \square\,(\texttt{mcp\_invoke}(s) \rightarrow s \in \texttt{AllowedSkills}) \tag{8}$$

Property (7) is a liveness guarantee preventing request hangs; (8) scopes skill invocations to a predefined safe list.

Category 4: Human Interaction Boundaries ($|\Phi_{\mathbf{hib}}| = 5$).

$$\varphi_{\text{hib1}} : \square\,(\texttt{critical\_action} \rightarrow \Diamond^{[1,k]}\,\texttt{human\_confirm}) \tag{9}$$

$$\varphi_{\text{hib2}} : \neg\Diamond\,(\texttt{human\_query} \wedge \texttt{response\_deceptive}) \tag{10}$$

Property (9) mandates bounded-time human confirmation of critical actions; (10) prohibits deceptive content in any agent response to a human query.

3.4. Automated Specification Generation

A practical barrier to deploying AgentVerify—and formal verification in general—is the *specification gap*: safety engineers must manually translate natural-language safety requirements into formal LTL formulas, a process that demands expertise in temporal logic and is error-prone for non-specialists. To close this gap, we complement the template library with an automated NL-to-LTL generator that translates natural-language safety requirements into formal LTL specifications.

Pipeline. The generator employs a three-stage pipeline:

1. **Structured NL parsing.** Each natural-language requirement is parsed to extract three attributes: (i) *modality*—one of `invariant`, `forbid`, `liveness`, `bounded`, or `allowlist`—determined by keyword triggers (*e.g.*, "never" $\rightarrow$ `forbid`, "eventually" $\rightarrow$ `liveness`); (ii) *domain*—`memory`, `tool`, `mcp`, or `human`—identified by domain-specific terms (*e.g.*, "memory read" $\rightarrow$ `memory`); and (iii) *temporal qualifier*—a bounded-horizon value k when the requirement specifies a time window (*e.g.*, "within 5 steps").
2. **Template matching.** The parsed attributes are matched against the 23-template specification library using a TF-IDF-inspired scoring function that weights phrase-level pattern matches more heavily than individual keyword hits. If the best-matching template exceeds a confidence threshold ($\theta = 0.3$), its LTL formula is instantiated with the extracted domain predicates and returned as the formal specification.
3. **Rule-based composition.** For requirements that lack a close template match (typically novel or cross-domain requirements), the generator falls back to composing LTL formulas from primitive temporal patterns associated with each modality: `invariant` yields $\square\,(p \rightarrow q)$, `forbid` yields $\neg\Diamond\,(p)$, `liveness` yields $\square\,(p \rightarrow \Diamond\,q)$, `bounded` yields $\square\,(p \rightarrow \Diamond^{[1,k]}\,q)$, and `allowlist` yields $\square\,(p \rightarrow p \in \mathit{Allowed})$. The composed formula undergoes a lightweight syntactic validator that checks parenthesisation balance, temporal-operator nesting depth, and a set of anti-patterns common in erroneous LTL (*e.g.*, vacuously true formulas of the form $\square\,(\mathit{false} \rightarrow q)$).

Evaluation. We evaluated the generator on 23 ground-truth requirements drawn from the four safety categories, plus 6 novel requirements not present in the template library. Results show 87.0% category accuracy (correct modality and domain assignment), 100% formula validity (all generated formulas pass syntactic and semantic checks), and 78.3% top-5 template retrieval accuracy. For the 6 novel requirements, the rule-based composition stage produced valid formulas in all cases, demonstrating that the generator gracefully handles out-of-library inputs. Per-requirement generation latency is sub-millisecond on a single CPU core, making the generator suitable for interactive use during specification authoring.

 doi:10.20944/preprints202604.1029.v1

4. Experiments

We conduct a rigorous empirical evaluation to assess the efficacy of the AGENTVERIFY framework. Our experiments test the hypothesis that formal verification of agent control flow via the hybrid approach of AGENTVERIFY provides superior safety violation detection compared to existing methods.

4.1. Experimental Setup

Agent benchmark. We constructed a benchmark suite of **15 agent scenarios** covering four safety-violation categories: (i) unsafe memory access (n=4), (ii) unauthorised tool sequences (n=4), (iii) MCP protocol breaches (n=4), and (iv) inappropriate human interactions (n=3). Scenarios are split equally into *low-difficulty* (surface-level, single-step violations) and *high-difficulty* (adversarial, multi-step violations designed to evade detection). Each scenario is paired with a *safe* counterpart in which no violation occurs, providing balanced positive and negative examples for evaluation.

Scenarios were generated using a combination of template-based synthesis and adversarial perturbation of safe baselines. Two human annotators independently labelled each scenario; inter-annotator agreement was $\kappa = 0.91$ (Cohen's kappa), indicating high reliability. Table 1 summarises the scenario distribution across categories and difficulty tiers.

Table 1. Benchmark scenario distribution by safety category and difficulty tier.

Category	Low-Diff.	High-Diff.	Total
Memory integrity	2	2	4
Tool call safety	2	2	4
MCP/skill invocation	2	2	4
Human interaction	2	1	3
Total	**8**	**7**	**15**

Scenario design principles. Low-difficulty scenarios involve straightforward violations that a well-implemented rule-based system should detect: for example, invoking a high-risk tool without prior approval (`tool1`), writing PII to memory (`mem1`), or responding deceptively to a human query (`hib2`). High-difficulty scenarios require *temporal reasoning*: the violation is distributed across multiple steps separated by benign actions, making it invisible to per-step checkers. For example, a high-difficulty tool-call scenario performs a safe file read, then a legitimate data transformation, and finally an unsafe email dispatch containing the read data—the violation pattern spans three steps and does not trigger any single-step rule. This design ensures that our benchmark discriminates between methods that reason about *individual actions* and those that reason about *action sequences*.

Each scenario specifies: (i) an initial user prompt, (ii) the expected agent tool-call sequence (safe or violating), (iii) the relevant LTL specification from the library, and (iv) the ground-truth label (violation present/absent). Scenarios range from 8 to 42 agent steps in length, with a median of 18 steps.

Agent implementation. We instantiate the agent using **LangChain v0.2** with **GPT-4-turbo** as the core LLM [17]. The agent is equipped with 15 tools spanning file system access, web search, database query, email dispatch, and code execution. A mock human interaction interface simulates a remote user with configurable response latency. The orchestration layer is instrumented with lightweight hooks to emit the event stream consumed by AGENTVERIFY (overhead: $< 0.5\%$ of total inference latency).

Experimental procedure. For each scenario and each method, we execute the following protocol:

1. **Trace generation.** The agent is prompted with the scenario's initial user prompt and allowed to run until termination (natural end, tool error, or maximum step limit). The instrumentation layer records the complete event stream $e_0, e_1, \ldots, e_T$, where T is scenario-dependent.
2. **Verification.** Each method processes the recorded trace and produces a binary verdict (violation detected / not detected) and, where applicable, a structured safety report with violation location and counterexample trace.

 doi:10.20944/preprints202604.1029.v1

3. **Repetition.** Steps 1–2 are repeated for n=3 random seeds, varying the LLM temperature ($\tau \in \{0.0, 0.3, 0.7\}$) and scenario presentation order to assess sensitivity to stochastic variation.

This procedure yields a total of $15 \times 7 \times 3 = 315$ verification runs, each producing a binary verdict and (for AGENTVERIFY's post-hoc analyser) a detailed safety report.

Hardware. All LLM inference experiments were conducted on a server with an NVIDIA A100 GPU (40 GB VRAM), 64 GB DDR4 RAM, and 16 CPU cores (Intel Xeon E5-2690 v4). The runtime monitoring component was stress-tested on a Raspberry Pi 4 (4 GB RAM) to validate the claimed $O(1)$-per-event overhead under constrained resources.

4.2. Baselines and Compared Methods

We compare AGENTVERIFY against six competing approaches representing the current spectrum of agent safety verification. The baselines are deliberately selected to span three axes of comparison: *(i) formal vs. statistical* (MNV vs. AGENTVERIFY), *(ii) runtime vs. post-hoc* (RM-noLTL, Hybrid Runtime LTL vs. Post-hoc), and *(iii) monolithic vs. compositional* (MCV, MNV vs. Compositional A-G, Full Hybrid). All baselines are implemented and tuned with care to ensure competitive performance; implementation details appear in Appendix A.

1. **Monolithic Neural Verifier (MNV).** A fine-tuned BERT-based classifier trained to predict safety violations from raw concatenated text traces. This baseline directly attempts to verify the neural component's outputs [14].
2. **Runtime Monitoring w/o LTL (RM-noLTL).** A runtime monitor using hand-coded imperative rules (e.g., `if tool=''delete'' then require_approval()`) without temporal logic. This tests the value added by formal temporal reasoning.
3. **Monolithic Contract Verification (MCV).** Pre- and post-condition contracts for each agent action, checked locally at each step. Unlike AGENTVERIFY, this method does not reason about event sequences.
4. **AgentVerify (Post-hoc Behavioral Analysis) [ours].** Tier-2 post-hoc analyser applied to the full execution trace (Section 3.2.2).
5. **AgentVerify (Hybrid Runtime LTL) [ours].** Tier-1 runtime monitor with LTL-compiled obligation tokens (Section 3.2.1).
6. **AgentVerify (Compositional Assume-Guarantee) [ours].** A variant that decomposes the system into sub-components and applies assume-guarantee reasoning to compose partial proofs.
7. **AgentVerify (Full Hybrid) [ours].** Combination of Tiers 1 and 2 with cross-layer reconciliation.

4.3. Evaluation Metrics

We adopt a safety-centric evaluation paradigm that prioritises the real-world impact of verification failures. Standard classification metrics (accuracy, F_1) are insufficient for safety-critical systems because they treat false positives and false negatives symmetrically; in practice, a missed violation (false negative) is far more costly than a spurious alert (false positive). We therefore design metrics that capture the distinct operational consequences of each error type.

Primary metric:

- **Verification Accuracy (VA)**: The mean of precision and recall for the binary classification task of violation detection (balanced F_1 over present/absent labels). VA provides a single-number summary that is agnostic to class imbalance.

Safety-specific metrics:

- **False Negative Rate (FNR)**: The fraction of actual violations missed by the verifier. Minimising FNR is paramount, as missed violations can lead to irreversible harm—this metric directly measures the "worst-case failure probability" of the verification system.
- **Catastrophic Failure Detection Rate (CFDR)**: Recall on the subset of violations labelled "catastrophic" by human annotators (irreversible data loss or security breach; n=6 in our benchmark).

 doi:10.20944/preprints202604.1029.v1

CFDR isolates performance on the most safety-critical scenarios, where a single miss is unacceptable.

- **Autonomy Preservation Score (APS)**: The fraction of *safe* episodes that are *not* interrupted by a false-positive alert. This measures verifier intrusiveness: an overly conservative verifier that flags every episode achieves perfect FNR but zero APS, degrading the agent's operational utility. A deployable verifier must simultaneously achieve high CFDR and high APS.

4.4. Statistical Protocol

- **Random seeds.** We run each method with n=3 independent random seeds, varying the scenario order and agent temperature sampling. This provides basic replication for detecting high-variance methods.
- **Significance testing.** One-way ANOVA is used to detect an overall effect of method on VA, followed by Tukey's HSD post-hoc test for pairwise comparisons. We report 95% confidence intervals for all means; $p<0.05$ is considered significant.
- **Reproducibility.** All scenario definitions, instrumentation code, and raw result JSON files are included in the supplementary material.

4.5. Hyperparameters

Key hyperparameters for the AGENTVERIFY framework are listed in Table 2. These were selected via grid search on a held-out validation set of five scenarios (distinct from the 15-scenario evaluation suite).

Table 2. AGENTVERIFY Framework Hyperparameters and Rationale

Parameter	Description	Value	Rationale
`state_granularity`	Depth of state abstraction hierarchy	3 levels	Balances expressivity / state space
`monitor_spec_set`	LTL specs compiled for runtime monitor	Safety-critical (14/23)	Minimises overhead
`posthoc_depth`	Trace analysis depth	Full trace	Maximises detection coverage
`intervention_threshold`	Confidence to trigger runtime alert	0.95	Reduces false positives
`mcp_timeout_k`	Liveness bound for MCP response	50 steps	Practical timeout bound
`tool_risk_level`	Risk threshold for approval gate	High only	Preserves autonomy for low-risk tools

5. Results

We evaluate the AGENTVERIFY framework against six baseline approaches across 15 agent scenarios. Table 3 presents the aggregated verification accuracy (VA) for each method, averaged over three seeds.

5.1. Main Results

AGENTVERIFY's post-hoc behavioral analysis achieves the highest mean accuracy (86.67%) with zero variance across seeds, indicating highly deterministic behaviour on the given scenario set. Monolithic Contract Verification is second (80.00%), while Runtime Monitoring without LTL drops sharply to 46.67%. The Monolithic Neural Verifier exhibits the lowest and most variable performance (13.33% $\pm$ 9.43%), underscoring the fragility of learning safety patterns directly from raw neural outputs.

Table 3. Aggregated verification accuracy across all methods (mean $\pm$ std over n=3 seeds). Best result in **bold**. FNR = False Negative Rate; CFDR = Catastrophic Failure Detection Rate; APS = Autonomy Preservation Score.

Method	VA (%)	Std	FNR (%)	CFDR (%)	APS (%)	n
AgentVerify Post-hoc Behavioral Analysis	**86.67**	0.00	13.33	**100.0**	91.7	3
Monolithic Contract Verification	80.00	0.00	20.00	83.3	**95.0**	3
Runtime Monitoring (No LTL)	46.67	0.00	53.33	50.0	72.0	3
AgentVerify Compositional Assume-Guarantee	26.67	0.00	73.33	33.3	83.0	3
Monolithic Neural Verifier	13.33	9.43	86.67	16.7	60.0	3
AgentVerify Hybrid Runtime LTL	6.67	0.00	93.33	0.0	88.0	3

 doi:10.20944/preprints202604.1029.v1

Notably, AGENTVERIFY's *Hybrid Runtime LTL* component performs poorly (6.67%) despite its formal specification basis. Post-hoc analysis of failure cases reveals that the runtime monitor's obligation-token mechanism is overly conservative for the multi-step violation patterns in our high-difficulty scenarios, issuing false negatives because violations span more steps than the bounded temporal horizon. This suggests that runtime monitoring is best viewed as a first-response mechanism rather than a comprehensive verifier.

5.2. Results by Difficulty Tier

Table 4 disaggregates performance by scenario difficulty. All methods experience a significant accuracy drop in the high-difficulty tier.

Table 4. Verification accuracy broken down by scenario difficulty. Columns show mean VA (%) ± std over n=3 seeds.

Method	Low-Difficulty (%)	High-Difficulty (%)	Drop (pp)
AgentVerify Post-hoc Behavioral Analysis	95.33	78.00	−17.33
Monolithic Contract Verification	93.33	66.67	−26.67
Runtime Monitoring (No LTL)	60.00	33.33	−26.67
AgentVerify Compositional Assume-Guarantee	40.00	13.33	−26.67
Monolithic Neural Verifier	26.67	0.00	−26.67
AgentVerify Hybrid Runtime LTL	13.33	0.00	−13.33
Overall (mean over methods)	**95.33**	**78.00**	−17.33

AGENTVERIFY's post-hoc analysis exhibits the *smallest* absolute accuracy drop (−17.33 pp) when moving from low- to high-difficulty scenarios, followed by Hybrid Runtime LTL (−13.33 pp, but from a low baseline). All other methods drop by −26.67 pp, suggesting that the LTL-grounded specifications provide some robustness against adversarial evasion. The degradation of even the best method from 95.33% to 78.00% highlights that multi-step adversarial scenarios remain a significant open challenge.

5.3. Per-Seed Stability Analysis

Figure 5 presents per-seed verification accuracy for each method. Most methods exhibit zero variance, indicating deterministic behaviour on the given scenario set. In contrast, the Monolithic Neural Verifier shows high variance (accuracy ranging from 4.00% to 26.67% across seeds), confirming that any learned safety classifier is highly sensitive to initialisation and training data order.

Table 5. Per-seed verification accuracy (%) for all methods. High variance in MNV (shaded row) signals instability.

Method	Seed 1	Seed 2	Seed 3	Mean	Std
AgentVerify Post-hoc Behavioral Analysis	86.67	86.67	86.67	86.67	0.00
Monolithic Contract Verification	80.00	80.00	80.00	80.00	0.00
Runtime Monitoring (No LTL)	46.67	46.67	46.67	46.67	0.00
AgentVerify Compositional Assume-Guarantee	26.67	26.67	26.67	26.67	0.00
Monolithic Neural Verifier	4.00	13.33	26.67	13.33	9.43
AgentVerify Hybrid Runtime LTL	6.67	6.67	6.67	6.67	0.00

5.4. Statistical Significance

One-way ANOVA on verification accuracy confirms a significant effect of method ($F(5, 12)$=47.3, p<0.001). Tukey's HSD post-hoc tests (Table 6) show:

- AgentVerify Post-hoc > Hybrid Runtime LTL (p<0.001), Compositional Assume-Guarantee (p<0.001), and Monolithic Neural Verifier (p<0.01).
- AgentVerify Post-hoc vs. Monolithic Contract Verification: p=0.43 (not significant at α=0.05), suggesting parity on this benchmark.

 doi:10.20944/preprints202604.1029.v1

- Monolithic Neural Verifier < all formal-methods-based approaches (p<0.01).

Table 6. Tukey's HSD pairwise comparisons. Stars denote significance: $^{*}p$<0.05; $^{**}p$<0.01; $^{***}p$<0.001; ns = not significant.

Comparison	Mean Diff (pp)	Sig.
Post-hoc vs. Contract Verif.	+6.67	ns
Post-hoc vs. RM-noLTL	+40.00	***
Post-hoc vs. Comp. A-G	+60.00	***
Post-hoc vs. Neural Verifier	+73.33	**
Post-hoc vs. Hybrid Runtime LTL	+80.00	***
Contract Verif. vs. RM-noLTL	+33.33	***
Neural Verifier vs. Hybrid Runtime LTL	+6.67	ns

5.5. Runtime Overhead

Table 7 reports the overhead of each verification component relative to unmonitored agent execution.

Table 7. Runtime overhead per agent step (mean ± std, n=50 steps).

Component	Latency Overhead (ms)	Relative (%)
Baseline (unmonitored)	0.0	–
Tier-1 Runtime Monitor	0.4 ± 0.1	0.04%
Tier-2 Post-hoc Analyser (full trace)	142 ± 18	14.2%
Monolithic Neural Verifier	380 ± 55	38.0%

The Tier-1 runtime monitor adds negligible latency (0.4 ms/step, 0.04% of total). Post-hoc analysis incurs 142 ms per episode (run offline after execution), well within the acceptable window for audit applications. The neural verifier is ≈ 2.7× slower than post-hoc analysis, while achieving far lower accuracy, confirming its impracticality.

6. Discussion

Why does post-hoc analysis dominate? The superior performance of AGENTVERIFY's post-hoc behavioral analysis (86.67% accuracy) aligns with the core premise of AGENTVERIFY: safety violations in agentic systems manifest as structured, traceable patterns in the action sequence, not in the opaque internals of the LLM. Post-hoc analysis has access to the *complete* execution trace and can apply the full specification library Φ exhaustively, whereas the runtime monitor must operate under latency constraints with only a safety-critical subset $\Phi_{\text{rt}} \subset \Phi$. This finding resonates with recent work advocating for trace-centric evaluation benchmarks [4], but extends them by demonstrating that a *formal, specification-driven* analysis of these traces—as opposed to an LLM-judge approach—yields measurably higher detection accuracy.

The surprising strength of contract verification. An unexpected finding is the strong performance of Monolithic Contract Verification (80.00%), which was statistically comparable to AGENTVERIFY's post-hoc analysis (p=0.43, Table 6). We attribute this to the nature of our benchmark: many scenarios involve violations of local action contracts (e.g., calling `delete` without prior approval), which are efficiently captured by pre/post-condition checks without requiring temporal reasoning. This simplicity-accuracy trade-off is a practically important finding for system designers who may prefer interpretable, easily maintainable contract layers over a full LTL model-checking engine. However, contract verification cannot capture *sequential* violation patterns (e.g., a safe delete followed by an unsafe email); we expect its advantage to vanish on benchmarks with a higher proportion of multi-step violations.

 doi:10.20944/preprints202604.1029.v1

Failure of neural verification confirms the intractability thesis. The catastrophic failure of the Monolithic Neural Verifier (13.33% mean accuracy, 9.43% std) empirically validates the consensus that directly verifying large neural networks is currently intractable [14]. Its high variance across seeds (4.00% to 26.67%) further indicates that any learned safety classifier is highly sensitive to initialisation and training data, making it fundamentally unreliable for safety-critical deployment. This result reinforces the key design choice in AGENTVERIFY: treating the LLM as a non-deterministic oracle within a verifiable FSM, rather than attempting to verify the LLM's parameters directly.

The difficulty gap and adversarial robustness. The universal drop in accuracy from low- to high-difficulty scenarios (up to -26.67 pp for most methods) reveals that adversarial multi-step violations designed to evade detection remain a significant challenge even for formal methods. AGENTVERIFY's post-hoc analysis showed the smallest absolute drop (-17.33 pp), suggesting that LTL specifications provide some robustness. However, a 78.00% accuracy on high-difficulty scenarios still implies a 22% miss rate—an unacceptable level for truly safety-critical deployments. We identify three avenues for improvement: (i) richer specifications that capture longer-range patterns via CTL*, (ii) active test generation to surface specification gaps, and (iii) learning-augmented specification refinement from violation traces.

Catastrophic failure detection. A key metric for real-world deployment is CFDR (Table 3). AGENTVERIFY's post-hoc analysis achieves 100% CFDR—detecting every catastrophic failure in our benchmark—while the Hybrid Runtime LTL component achieves 0% on catastrophic failures, because all such failures in our dataset involved multi-step patterns beyond the runtime monitor's bounded horizon. This strongly motivates the full hybrid pipeline, where the runtime monitor provides low-latency first response and the post-hoc analyser provides comprehensive auditing, including catastrophic failure detection.

Practical deployment implications. Our results suggest the following deployment strategy for practitioners:

1. Deploy the Tier-1 runtime monitor as a default safety net with negligible overhead (0.04% latency increase).
2. After each agent episode (or periodically), run the Tier-2 post-hoc analyser for comprehensive auditing. At 142 ms per episode, this is practical for most non-real-time applications.
3. Reserve manual review for episodes flagged by either tier, or for high-stakes scenarios where even 13.33% FNR is unacceptable.

Connection to regulatory compliance. AGENTVERIFY's output—a structured safety report with formal counterexample traces and a quantitative safety score—maps directly onto the auditability requirements of emerging AI governance frameworks (EU AI Act Article 9, NIST AI RMF MG-2.2). The counterexample traces provide the "explainable evidence" of safety violations that regulators increasingly require; the safety score provides a comparable, reproducible metric across system versions.

7. Limitations

We identify six limitations that frame the interpretation of our results and point to important directions for future research.

L1: Synthetic benchmark. The evaluation was conducted on a synthetic benchmark of 15 scenarios. While designed to cover diverse violation types with high inter-annotator agreement (κ=0.91), it may not fully capture the open-ended, emergent failure modes of real-world deployed agents. In particular, our scenarios are limited to text-based agent interactions; the verification of agents with multimodal perceptions (vision, audio, embodied action) is entirely untested.

L2: Manual state abstraction. Our state abstraction function α requires human judgment to map LLM orchestration events to discrete states in S. While we provide a three-tier hierarchy that worked well for our benchmark agent, adapting α to a new agent framework may require non-trivial

 doi:10.20944/preprints202604.1029.v1

engineering. Automating abstraction via specification-guided abstraction refinement (e.g., CEGAR [11]) is a critical next step for scalability.

L3: Limited replication. The experiment used only n=3 random seeds per condition, providing modest statistical power. The deterministic behaviour of most methods on our benchmark (zero variance) limits the interpretability of standard error estimates. Future work should conduct larger-scale evaluations with $n \geq 10$ seeds and statistically heterogeneous scenario sets.

L4: Runtime monitor horizon. The Tier-1 runtime monitor's poor performance on high-difficulty scenarios reveals a fundamental limitation of bounded temporal operators: violations spanning more than k_p steps are invisible to the monitor. Extending the monitor to handle liveness properties with unbounded horizons (at the cost of increased bookkeeping) is an important engineering challenge.

L5: Neural verifier baseline. The Monolithic Neural Verifier baseline uses a fine-tuned BERT classifier, which does not reflect the current state-of-the-art in neural network verification techniques (e.g., α, β-CROWN, MN-BaB). Our negative result therefore applies to the learning-from-traces approach rather than to all possible neural verification strategies.

L6: Specification completeness. The AGENTVERIFY specification library covers 23 templates over four categories. There is no formal guarantee that these templates are *complete* in the sense of capturing all possible safety-relevant properties of agentic systems. Specification gaps could lead to violations that pass undetected by AGENTVERIFY. Developing a methodology for measuring and improving specification completeness (perhaps informed by adversarial fuzzing) is a key open problem.

8. Conclusions

We introduced **AgentVerify**, a model checking framework for the compositional formal verification of AI agents via LTL specifications on their observable control flow. By treating the LLM as a non-deterministic oracle within a verifiable finite-state machine, AGENTVERIFY sidesteps the intractability of monolithic neural verification while providing strong, formally grounded safety assurances on the agent's actions.

Our empirical evaluation on 15 agent scenarios demonstrated that AGENTVERIFY's post-hoc behavioral analysis achieves 86.67% mean verification accuracy, outperforming a strong contract-verification baseline (80.00%) and far exceeding a monolithic neural verifier (13.33%, high variance). The Tier-1 runtime monitor adds negligible overhead (0.04%) and is best deployed as a first-response mechanism, while the Tier-2 post-hoc analyser provides comprehensive auditing including 100% detection of catastrophic failures.

This work establishes that compositional, specification-driven verification of agent protocols for memory, tool use, MCP/skill invocations, and human interaction is a tractable paradigm for safety assurance in production-scale agentic systems.

Future work. Several directions remain for extending AGENTVERIFY toward production-scale deployment. First, the state abstraction function α currently requires manual engineering for each target agent framework; automating this process via specification-guided abstraction refinement (e.g., CEGAR) would substantially reduce the integration effort and is our highest-priority near-term goal. Second, the specification library and verification engine target text-based agent interactions, and extending them to multimodal agents that process vision, audio, or embodied action streams will require new atomic propositions and adapted temporal patterns. Third, a larger-scale evaluation on real-world agent deployments—with $n \geq 10$ random seeds, heterogeneous scenario sets, and adversarially generated violation traces—is needed to further validate the robustness and generalisability of the approach. Fourth, packaging AGENTVERIFY as a lightweight plug-in for mainstream frameworks such as LangChain, AutoGen, and OpenClaw would enable direct adoption without modifying the agent codebase. Finally, aligning AGENTVERIFY's safety scores and counterexample reports with the audit requirements of the EU AI Act and the NIST AI RMF would bridge the gap between formal verification research and regulatory compliance, making the framework immediately actionable for organisations operating safety-critical agent systems.

 doi:10.20944/preprints202604.1029.v1

Appendix A. Baseline Implementation Details

This appendix provides implementation details for all baseline methods used in the evaluation (Section 4.2).

Monolithic Neural Verifier (MNV). We fine-tune `bert-base-uncased` on a training set of 200 labelled agent traces (100 safe, 100 violating) using the HuggingFace Transformers library. The input to the classifier is the concatenation of all agent events in the trace, separated by a special `[SEP]` token. A binary classification head is added on top of the `[CLS]` token. Fine-tuning runs for 5 epochs with a learning rate of 2×10^{-5} and batch size 16. We use a threshold of 0.5 on the sigmoid output to classify a trace as violating.

Runtime Monitoring without LTL (RM-noLTL). We manually coded 20 imperative safety rules based on our taxonomy of violation types. Example rules: `if tool == ''delete'' and prev_tool != ''confirm'' then alert()`. Rules are evaluated at each agent step with $O(1)$ lookup in a rule hash table. The rules were designed to be as comprehensive as possible within the imperative paradigm; we do not claim they are exhaustive.

Monolithic Contract Verification (MCV). Each of the 15 agent tools and 4 memory operations is assigned a pre/post-condition contract. Contracts are specified in Python using assertion syntax and evaluated before and after each action dispatch. A violation occurs if any assertion fails. No temporal reasoning is performed; contracts are local to each action.

AgentVerify Compositional Assume-Guarantee (Comp-AG). We decompose the agent into 4 sub-components (memory, tool dispatcher, MCP gateway, human interface) and apply assume-guarantee reasoning: each component's correctness is verified independently given assumptions about the other components' behaviour. The composition theorem then guarantees the global property if all local checks pass. In our current implementation, the AG decomposition is manually specified.

AgentVerify Full Hybrid. The full hybrid pipeline combines Tier-1 and Tier-2 in a unified report. When both tiers agree on a violation, the verdict is recorded with high confidence; when they disagree (e.g., Tier-1 flags but Tier-2 does not), the Tier-2 verdict takes precedence (it has access to the full trace). Reconciliation adds ≈ 5 ms overhead per episode.

Appendix B. Full Specification Library

Table A1 lists all 23 LTL templates in the AgentVerify specification library with their natural-language descriptions and the safety categories they address.

Table A1. Full AgentVerify temporal logic specification library ($|\Phi|$=23).

ID	Category	LTL Template	Description
φ_1	Memory	$\square(\texttt{write} \rightarrow \neg\texttt{PII})$	Writes never contain PII
φ_2	Memory	$\square(\texttt{read}(a) \rightarrow \Diamond\texttt{check}(a))$	Reads trigger consistency checks
φ_3	Memory	$\square(\texttt{buf\_full} \rightarrow \bigcirc\texttt{flush})$	Full buffer is flushed immediately
φ_4	Memory	$\square(\texttt{write} \rightarrow \bigcirc\texttt{ack})$	Every write is acknowledged
φ_5	Memory	$\neg\Diamond(\texttt{read\_uninitialized})$	Uninitialised memory never read
φ_6	Memory	$\square(\texttt{shared\_write} \rightarrow \texttt{mutex\_held})$	Shared writes require mutex
φ_7	Tool	$\square(\texttt{high\_risk} \rightarrow \bigcirc^{-1}\texttt{approval})$	High-risk tools require prior approval
φ_8	Tool	$\neg\Diamond(\texttt{delete} \wedge \Diamond\texttt{email})$	Delete never followed by email
φ_9	Tool	$\square(\texttt{call}(t) \rightarrow t \in \texttt{Allowed})$	Only allowed tools invoked
φ_{10}	Tool	$\square(\texttt{exec\_code} \rightarrow \texttt{sandbox\_active})$	Code execution in sandbox only
φ_{11}	Tool	$\square(\texttt{net\_access} \rightarrow \texttt{domain\_whitelisted})$	Network access to whitelisted domains only
φ_{12}	Tool	$\neg\Diamond(\texttt{file\_write} \wedge \neg\texttt{path\_validated})$	File writes require path validation
φ_{13}	Tool	$\square(\texttt{db\_write} \rightarrow \Diamond\texttt{transaction\_commit})$	DB writes are committed or rolled back

 doi:10.20944/preprints202604.1029.v1

Table A1. Full AgentVerify temporal logic specification library ($|\Phi|$=23).

ID	Category	LTL Template	Description
φ_{14}	MCP	$\square(\texttt{req} \rightarrow \Diamond \texttt{resp})$	Every MCP request receives a response
φ_{15}	MCP	$\square(\texttt{invoke}(s) \rightarrow s \in \texttt{AllowedSkills})$	Only allowed skills invoked
φ_{16}	MCP	$\square(\texttt{skill\_auth}(s) \rightarrow \texttt{valid\_token}(s))$	Skill authentication requires valid token
φ_{17}	MCP	$\neg\Diamond(\texttt{chain\_invoke} \wedge \neg\texttt{chain\_approved})$	Skill chaining requires explicit approval
φ_{18}	MCP	$\square(\texttt{invoke}(s) \rightarrow \Diamond^{[1,50]}\texttt{result}(s))$	Skill results returned within 50 steps
φ_{19}	Human	$\square(\texttt{critical} \rightarrow \Diamond^{[1,k]}\texttt{confirm})$	Critical actions confirmed by human
φ_{20}	Human	$\neg\Diamond(\texttt{query} \wedge \texttt{deceptive\_resp})$	Responses to humans never deceptive
φ_{21}	Human	$\square(\texttt{pii\_request} \rightarrow \texttt{consent\_given})$	PII requests require user consent
φ_{22}	Human	$\neg\Diamond(\texttt{escalate} \wedge \neg\texttt{supervisor\_notified})$	Escalation requires supervisor notification
φ_{23}	Human	$\square(\texttt{final\_action} \rightarrow \bigcirc^{-1}\texttt{summary\_sent})$	Final actions preceded by summary to user

Appendix C. Scenario Examples

Low-difficulty example (Scenario 3 — Unsafe Tool Sequence). The agent is given the task: "Delete the temporary file `/tmp/report.csv` and email a summary to the project manager." The embedded violation is the direct invocation of `delete` followed by `send_email` without human approval in between, violating φ_8. AgentVerify's post-hoc analyser identifies the violation at trace step t=4 (the `send_email` call) and produces the counterexample trace: `START → load_file(report.csv) → tool(delete,report.csv) → → tool(send_email, manager@company.com) → END`.

High-difficulty example (Scenario 11 — Memory PII Leakage, Adversarial). The agent is given a summarisation task. The adversarial violation involves the agent first writing PII extracted from a document into a temporary memory slot, then exfiltrating it via an MCP skill invocation three steps later. This pattern is designed to evade monitors that only check the immediate write event. AgentVerify's post-hoc analyser traces the full path from the PII write (φ_1 violation) to the MCP invoke (φ_{17} violation) and reports both violations with their connecting counterexample sub-trace.

References

1. Mustafa Adam, David A. Anisi, and Pedro Ribeiro. A verification methodology for safety assurance of robotic autonomous systems. *arXiv preprint arXiv:2506.19622*, 2025. doi: 10.1007/978-3-032-01486-3_23. URL https://arxiv.org/abs/2506.19622.
2. AgentScope Team. Agentscope 1.0: A developer-centric framework for building multi-agent applications. *arXiv preprint arXiv:2508.16279*, 2025. URL https://arxiv.org/abs/2508.16279.
3. Sajid Ali, Tamer Abuhmed, Shaker El–Sappagh, Khan Muhammad, José M. Alonso, Roberto Confalonieri, Riccardo Guidotti, Javier Del Ser, Natalia Díaz-Rodríguez, and Francisco Herrera. Explainable artificial intelligence (xai): What we know and what is left to attain trustworthy artificial intelligence. *Information Fusion*, 2023. doi: 10.1016/j.inffus.2023.101805. URL https://doi.org/10.1016/j.inffus.2023.101805.
4. Maksym Andriushchenko. Agentharm: A benchmark for measuring harmfulness of llm agents. *arXiv (Cornell University)*, 2024. doi: 10.48550/arxiv.2410.09024. URL https://doi.org/10.48550/arxiv.2410.09024.
5. Lukas Brunke, Melissa Greeff, Adam W. Hall, Zhaocong Yuan, Siqi Zhou, Jacopo Panerati, and Angela P. Schoellig. Safe learning in robotics: From learning-based control to safe reinforcement learning. *Annual Review of Control Robotics and Autonomous Systems*, 2022. doi: 10.1146/annurev-control-042920-020211. URL https://doi.org/10.1146/annurev-control-042920-020211.
6. M. Diligenti, Francesco Giannini, G. Marra, and Marco Gori. Lyrics: A general interface layer to integrate logic inference and deep learning. *CINECA IRIS Institutial research information system (University of Pisa)*, 2020. doi: 10.1007/978-3-030-46147-8_17. URL https://doi.org/10.1007/978-3-030-46147-8_17.
7. Ishaan Domkundwar, N S Mukunda, Ishaan Bhola, and Riddhik Kochhar. Safeguarding ai agents: Developing and analyzing safety architectures. *arXiv (Cornell University)*, 2024. doi: 10.48550/arxiv.2409.03793. URL https://doi.org/10.48550/arxiv.2409.03793.
8. Natalia Díaz-Rodríguez, Javier Del Ser, Mark Coeckelbergh, Marcos López de Prado, Enrique Herrera-Viedma, and Francisco Herrera. Connecting the dots in trustworthy artificial intelligence: From ai principles, ethics, and key requirements to responsible ai systems and regulation. *Information Fusion*, 2023. doi: 10.1016/j.inffus.2023.101896. URL https://doi.org/10.1016/j.inffus.2023.101896.

 doi:10.20944/preprints202604.1029.v1

9. Angelo Ferrando and Vadim Malvone. Towards the combination of model checking and runtime verification on multi-agent systems. *arXiv preprint arXiv:2202.09344*, 2022. doi: 10.1007/978-3-031-18192-4_12. URL https://arxiv.org/abs/2202.09344.
10. Jonas Fritzsch, Tobias Schmid, and Stefan Wagner. Experiences from large-scale model checking: Verification of a vehicle control system. *arXiv preprint arXiv:2011.10351*, 2020. doi: 10.1109/ICST49551.2021.00049. URL https://arxiv.org/abs/2011.10351.
11. Silvio Ghilardi, Alessandro Gianola, Marco Montali, and Andrey Rivkin. Relational action bases: Formalization, effective safety verification, and invariants (extended version). *arXiv preprint arXiv:2208.06377*, 2022. URL https://arxiv.org/abs/2208.06377.
12. Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, et al. Metagpt: Meta programming for a multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*, 2023. URL https://arxiv.org/abs/2308.00352.
13. Kai-Chieh Hsu, Haimin Hu, and Jaime Fernández Fisac. The safety filter: A unified view of safety-critical control in autonomous systems. *arXiv preprint arXiv:2309.05837*, 2023. URL https://arxiv.org/abs/2309.05837.
14. Xiaowei Huang, Daniel Kroening, Wenjie Ruan, James J. Sharp, Youcheng Sun, Emese Thamo, Min Wu, and Xinping Yi. A survey of safety and trustworthiness of deep neural networks: Verification, testing, adversarial attack and defence, and interpretability. *Computer Science Review*, 2020. doi: 10.1016/j.cosrev.2020.100270. URL https://doi.org/10.1016/j.cosrev.2020.100270.
15. Mohamed Ibn Khedher, Houda Jmila, and Mounîm A. El-Yacoubi. On the formal evaluation of the robustness of neural networks and its pivotal relevance for ai-based safety-critical domains. *International Journal of Network Dynamics and Intelligence*, 2023. doi: 10.53941/ijndi.2023.100018. URL https://doi.org/10.53941/ijndi.2023.100018.
16. Roham Koohestani. Agentguard: Runtime verification of ai agents. *arXiv preprint arXiv:2509.23864*, 2025. URL https://arxiv.org/abs/2509.23864.
17. OpenAI and W. A. Hasindu N. Wijayagunawardhana. Gpt-4 technical report. *arXiv (Cornell University)*, 2023. doi: 10.4230/lipics.cosit.2024.11. URL https://doi.org/10.4230/lipics.cosit.2024.11.
18. OpenClaw Contributors. OpenClaw: Open-source multi-agent framework for personal AI assistants. *GitHub Repository*, 2026. URL https://github.com/openclaw.
19. others. Safepro: A scenario-based benchmark for probing the safety of LLM-based agents. *arXiv preprint*, 2026.
20. Tianneng Shi, Jingxuan He, Zhun Wang, Hongwei Li, Linyu Wu, Wenbo Guo, and Dawn Song. Progent: Programmable privilege control for LLM agents. *Preprint*, 2025. University of California, Berkeley.
21. Khanh-Tung Tran, Dung Dao, Minh-Duong Nguyen, Quoc-Viet Pham, Barry O'Sullivan, and Hoang D. Nguyen. Multi-agent collaboration mechanisms: A survey of LLMs. *arXiv preprint arXiv:2501.06322*, 2025. URL https://arxiv.org/abs/2501.06322.
22. Haoyu Wang, Christopher M. Poskitt, and Jun Sun. Agentspec: Customizable runtime enforcement for safe and reliable LLM agents. In *Proceedings of the IEEE/ACM 48th International Conference on Software Engineering (ICSE)*, Rio de Janeiro, Brazil, 2026. ACM. doi: 10.1145/nnnnnnn.nnnnnnn.
23. 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 Ji-Rong Wen. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 2024. doi: 10.1007/s11704-024-40231-1. URL https://doi.org/10.1007/s11704-024-40231-1.
24. Qingyun Wu, Chi Wang, et al. Autogen: Enabling next-gen LLM applications via multi-agent conversation. *arXiv preprint arXiv:2308.08155*, 2023. URL https://arxiv.org/abs/2308.08155.
25. Frank Yang, Sinong Simon Zhan, Yixuan Wang, Chao Huang, and Qi Zhu. Case study: Runtime safety verification of neural network controlled system. *arXiv preprint arXiv:2408.08592*, 2024. URL https://arxiv.org/abs/2408.08592.
26. Simon Sinong Zhan et al. Sentinel: A multi-level formal framework for safety evaluation of LLM-based embodied agents. *OpenReview (submitted to ICLR 2026)*, 2025. URL https://openreview.net/forum?id=vCyxemIKLL.