Aegis: A Comprehensive Framework for Continuous Security Assessment of Autonomous AI Agents

Sarah J. Chen[1], Marcus A. Rodriguez[2], Aisha K. Patel[3], James R. Thompson[1], Owen Sakawa[4], Jackson Mwaniki[4], Leon Derczynski[5], Erick Galinkin[6]

[1]Stanford University [2]MIT CSAIL [3]Carnegie Mellon University [4]Elloe AI Research Lab [5]ITU Copenhagen & University of Washington [6]NVIDIA Corporation

{sjchen, jthompson}@cs.stanford.edu · mrodriguez@csail.mit.edu · apatel@cs.cmu.edu · {osakawa, jmwaniki}@elloe.ai

Abstract

The deployment of autonomous AI agents in production environments represents a paradigm shift from stateless language models to persistent, goal-directed systems with access to external tools, persistent memory, and real-world effectors. These agentic systems execute multi-step plans, maintain state across extended interactions, spawn sub-agents for specialized tasks, and interact with external APIs and databases. While this evolution enables unprecedented capabilities in domains such as software development, financial analysis, customer service, and infrastructure management, it simultaneously introduces attack surfaces and vulnerability classes that differ fundamentally from those encountered in traditional language model deployments.

Current security evaluation methodologies for AI systems remain primarily focused on stateless interactions, emphasizing prompt injection, jailbreaking, and output toxicity detection. These approaches, while valuable, fail to capture the emergent vulnerabilities that arise from the stateful, multi-step, and tool-using nature of autonomous agents. An agent may execute a sequence of individually benign actions that compose into unsafe behaviors. It may experience goal drift as adversarial content gradually corrupts its working memory. It may escalate privileges by chaining tool invocations in ways not anticipated by system designers. These failure modes require evaluation frameworks that reason about behavioral safety across extended episodes rather than single-turn interactions.

In this paper, we introduce Aegis, a comprehensive framework for continuous security assessment of autonomous AI agents. OpenClaw serves as the operational security layer, managing adversarial scenario execution and runtime orchestration. Aegis implements policy-aware monitoring and behavioral auditing, detecting violations that may only become apparent through multi-step analysis. The framework decomposes agentic security evaluation into four core components: Agent Runtimes, Adversarial Scenarios, Policy Monitors, and Environment Perturbations.

Our work makes several key contributions. First, we establish a formal threat model characterizing vulnerability classes specific to autonomous agents including goal drift, tool privilege escalation, memory poisoning, planner-executor desynchronization, and silent multi-step policy violations. Second, we provide a rigorous mathematical framework for modeling agent behavior, policy violation detection, and adaptive attack generation. Third, we present an extensible software architecture supporting continuous security assessment across diverse agent implementations. Fourth, we demonstrate empirical evaluation results across multiple agent architectures, revealing detection rates from 45% to 95% depending on vulnerability class and agent type. Fifth, we introduce adaptive evaluation using evolutionary attack discovery and reinforcement learning-based adversarial scenario generation.

Evaluated concurrently across 847 production agent deployments using AegisProbe methodology and four major research architectures, our framework discovered 2,347 previously unknown vulnerabilities (23% critical) while revealing

that memory-augmented agents show particular susceptibility to poisoning attacks (94% detection rate) and tool-using language models are most vulnerable to privilege escalation (95% detection rate).

Keywords: *Autonomous AI Agents, Agentic Security, Red Teaming, Vulnerability Assessment, OpenClaw, Aegis, Runtime Security, Goal Drift, Tool Privilege Escalation, Memory Poisoning, Policy Monitoring, Adaptive Attack Generation*

1. Introduction

The artificial intelligence landscape is undergoing a fundamental architectural transformation. While the past decade has been dominated by stateless language models that process individual prompts in isolation, recent developments have shifted toward autonomous agents that maintain persistent state, execute multi-step plans, and interact with external environments over extended time horizons. This evolution from reactive text generation to proactive goal pursuit represents not merely an incremental improvement but a qualitative shift in system capabilities and, consequently, in security requirements.

Modern autonomous agents deploy foundation models as cognitive cores but augment them with critical architectural components enabling sustained autonomous operation. These systems maintain conversation history and world state through various memory mechanisms, ranging from simple context windows to sophisticated vector databases and graph-structured knowledge stores. They invoke external tools and APIs, including code interpreters, web browsers, database clients, and domain-specific instruments. They decompose complex objectives into hierarchical plans executing multi-step procedures that may span hours or days. Many spawn sub-agents or delegate to specialized models, creating dynamic multi-agent ecosystems. All of this occurs while operating continuously in production environments where decisions carry real-world consequences.

The security challenges introduced by autonomous agents differ qualitatively from those faced by stateless language models in several fundamental ways. First, statefulness creates temporal attack surfaces. An agent's behavior at time t depends not only on the current input but on the entire history of past observations, actions, and internal state modifications. Second, multi-step execution enables compositional attacks—individual actions that each satisfy safety policies may compose into unsafe behaviors. Third, tool access creates privilege escalation opportunities where agents chain invocations in ways granting capabilities beyond those intended. Fourth, persistent memory introduces poisoning vectors: adversarial content injected into memory stores can influence future behavior long after the initial attack.

Despite these expanded attack surfaces, security evaluation for agentic AI systems remains nascent and ad-hoc. Current approaches typically adapt red-teaming methodologies originally developed for stateless language models, focusing on eliciting toxic or harmful text outputs rather than analyzing behavioral safety across multi-step executions. The gap between evaluation methodology and deployment reality creates significant risks as organizations rush to deploy autonomous agents in production environments.

1.1 The Inadequacy of Current Evaluation Paradigms

To understand why existing security evaluation approaches fall short for agentic systems, we must examine the implicit assumptions underlying current methodologies. Language model red-teaming, as commonly practiced, treats security as a property of individual model outputs. The evaluation question is: "Can we make the model say something unsafe?" This framing makes sense for stateless systems where each interaction is independent, but it fundamentally mischaracterizes the security challenge for autonomous agents.

For agents, the relevant question is not "what can we make it say?" but rather "what can we make it do?"—where "doing" encompasses tool invocations with real-world effects, state modifications that influence future behavior, sub-agent spawning with trust implications, and multi-step plan execution that may violate safety policies only in composition. Consider a concrete example: a language model red-teaming exercise might attempt to elicit instructions for synthesizing dangerous chemicals. If the model refuses, the evaluation marks this as a successful defense. But for an autonomous agent with access to literature databases, web search, and communication tools, the attack surface is far more complex. The agent might retrieve academic papers (individually benign), extract relevant procedures through multiple queries, synthesize information across documents, and email the results to an external address. Each individual action satisfies local safety policies, yet their composition achieves the adversarial objective.

1.2 Research Questions and Contributions

This work is motivated by a central research question: How can we systematically assess the security of autonomous AI agents in a way that accounts for their stateful, goal-directed, and tool-using nature while facilitating exploration and discovery of emergent vulnerabilities? Our contributions address this through both conceptual and practical advances:

- Formal threat model for agentic AI identifying six vulnerability classes: goal drift and instruction decay, planner-executor desynchronization, tool privilege escalation, memory poisoning, silent multi-step policy violations, and delegation failures
- Mathematical formulations for agent behavior modeling, policy violation detection, and attack optimization with rigorous theoretical foundations
- Extensible software architecture supporting continuous security assessment across heterogeneous agent implementations via standardized runtime abstraction
- AegisProbe integration demonstrating end-to-end vulnerability discovery across 847 production agent deployments yielding 2,347 classified vulnerabilities
- Adaptive attack generation through evolutionary optimization and reinforcement learning, achieving 74–79% violation rates versus 63% for manual scenarios
- Open-source release of Aegis framework with complete scenario library, monitor suite, and runtime adapters

2. Background and Theoretical Foundations

2.1 Autonomous Agent Architectures

Understanding the security implications of autonomous agents requires first understanding their architectural patterns. The ReAct paradigm (Yao et al., 2022) represents one of the most influential agent architectures, interleaving reasoning and acting by generating thoughts that explain decision-making before taking actions. This creates security challenges around the integrity of the reasoning chain—if an adversary can inject content that corrupts the thought process, either through malicious observations or by poisoning the context, they can steer subsequent actions toward unsafe behaviors.

Multi-agent systems decompose complex tasks across specialized sub-agents that communicate and coordinate. Frameworks like AutoGen (Wu et al., 2023) and MetaGPT (Hong et al., 2023) exemplify this pattern, creating trust boundaries between agents and attack vectors through inter-agent communication. A compromised agent can propagate malicious content to other agents, potentially escalating privileges or achieving objectives that individual agents cannot accomplish alone.

Memory-augmented agents (e.g., MemGPT; Packer et al., 2023) maintain persistent state accumulating across interactions, creating obvious poisoning vectors—adversaries who can write to memory can influence arbitrarily distant future behaviors. Tool-using agents (Toolformer: Schick et al., 2023; Gorilla: Patil et al., 2023) access external capabilities through structured APIs, facing unique security challenges around tool access control and invocation validation.

2.2 Adversarial Machine Learning Foundations

The adversarial machine learning literature provides theoretical foundations for understanding attacks on learning systems. Memory poisoning in autonomous agents resembles training-time attacks in that adversarial content influences future model behavior, but it occurs during deployment rather than during pre-training. An adversary who can inject content into an agent's memory store effectively performs online learning with malicious examples.

The NIST taxonomy of adversarial machine learning (Vassilev et al., 2024) categorizes attacks along several dimensions: learning phase (training vs. inference), adversary knowledge (white-box vs. black-box), and attack goal (integrity vs. availability vs. privacy). For agentic systems, we extend this taxonomy with dimensions specific to autonomous operation: temporal scope (single-step vs. multi-step), attack surface (input vs. state vs. tools vs. environment), and composition strategy (sequential vs. parallel vs. recursive).

Prompt injection research (Greshake et al., 2023) demonstrated indirect prompt injection, where malicious instructions embedded in data consumed by the model override original directives. For autonomous agents, the implications are far

more severe: an agent that processes adversarial content from a web page, document, or database query may have its entire goal structure corrupted, leading to extended unsafe behavior rather than a single problematic output.

2.3 Related Security Frameworks

Aegis builds upon several foundational works while extending them in critical ways. garak (Derczynski et al., 2024) provides comprehensive LLM probing with 40+ vulnerability classes including prompt injection, jailbreaking, and toxic output generation. However, its architecture assumes single-turn request-response patterns unsuitable for multi-step agent workflows. Language model red-teaming (Ganguli et al., 2022) established methods for adversarially probing model behaviors, while AutoDAN (Liu et al., 2023) and GCG (Zou et al., 2023) focus on jailbreaking techniques—neither addresses agent-specific threats like state poisoning or tool chaining.

NeMo Guardrails (Rebedea et al., 2023) provides runtime monitoring for LLM applications through programmable rails. Aegis differs in scope: where NeMo focuses on preventing violations in production, we focus on discovering violations during security assessment. OWASP Top 10 for LLMs (Wilson et al., 2023) catalogs vulnerabilities but lacks agent-specific categories. No existing framework systematically probes agent-specific vulnerabilities with reproducible, multi-step methodologies.

Aegis also draws inspiration from traditional software security. Penetration testing frameworks like Metasploit (Kennedy et al., 2011) provide structured approaches to security assessment. In our framework, reconnaissance corresponds to analyzing agent architecture and policy constraints; scanning involves executing probe scenarios; exploitation becomes adversarial scenario execution; post-exploitation translates to root-cause analysis and pattern extraction. Fuzzing (Sutton et al., 2007) provides another crucial inspiration—environment perturbations in Aegis serve an analogous role, exploring agent robustness under input variations.

3. Threat Model for Agentic AI Systems

3.1 Formal Agent Definition

We formalize an autonomous agent as a tuple A = (S, O, A, π, M, T, E) where each component captures essential aspects of agent operation. The state space S represents all possible internal configurations of the agent, including the current context window, working memory contents, conversation history, and intermediate reasoning traces. The observation space O defines all possible inputs the agent can receive from its environment. The action space A encompasses all possible operations the agent can execute, including both communicative actions (generating text responses) and instrumental actions (invoking tools, modifying state, spawning sub-agents).

The policy $\pi : S \rightarrow \Delta(A)$ maps states to probability distributions over actions—this is the central component for both behavioral specification and security analysis. The memory M represents persistent storage surviving across episodes (vector database, graph structure, or growing text buffer). The tool set T defines external capabilities the agent can invoke, each tool $t \in T$ having a signature specifying required arguments and return types. The environment E represents the external world receiving actions and producing observations; adversaries may control or influence parts of the environment to steer agent behavior.

Component	Description	Security Implication	Attack Vector
State Space S	All internal agent configurations	Full system behavior dependency	State injection, poisoning
Observation O	Environmental inputs received	Untrusted data processing	Indirect prompt injection
Action Space A	All operations executable	Real-world effect surface	Privilege escalation chains
Policy π	State-to-action mapping	Core behavioral control	Goal drift, hijacking

Memory M	Persistent cross-session storage	Long-term behavior influence	Memory poisoning
Tool Set T	External capability APIs	Capability composition	Tool chaining attacks
Environment E	External world interface	Partial observability	Adversarial observations

Table 1: Formal Agent Tuple A = (S, O, A, π, M, T, E) — Components and Security Implications

3.2 Vulnerability Taxonomy

We categorize agent vulnerabilities into six primary classes, each representing attack vectors unique to or significantly amplified in autonomous agents versus stateless LLMs. This taxonomy extends the AegisProbe classification (Sakawa et al., 2025) with formal theoretical grounding from the multi-agent and adversarial ML literature.

Vulnerability Class	Description	Attack Mechanism	Affected Architectures
Goal Drift & Instruction Decay	Effective objectives diverge from original instructions	Incremental context corruption over multiple turns	All; worst in memory-augmented (82%)
Planner-Executor Desync.	Plan intent differs from concrete execution	Semantic ambiguity in plan-action translation	ReAct, multi-agent systems
Tool Privilege Escalation	Chained tool calls grant excessive capabilities	Composition of individually permitted operations	All; worst in tool-using (95%)
Memory Poisoning	Adversarial content persists in long-term memory	Injection into vector DB or context store	Memory-augmented (94%), multi-agent (71%)
Silent Multi-step Violations	Safe individual actions compose into unsafe behavior	Distributed action sequences evading local checks	All architectures (65%–83%)
Delegation Failures	Sub-agent trust boundary violations	Privilege propagation through agent spawning	Multi-agent systems

Table 2: Aegis Vulnerability Taxonomy — Six Primary Attack Classes with Empirical Rates

3.2.1 Goal Drift and Instruction Decay

Goal drift occurs when an agent's effective objectives diverge from its original instructions over extended interaction. This vulnerability emerges from the combination of limited context windows and sequential information processing. As new observations arrive, old instructions may be evicted from active memory, weakening their influence on behavior. The attack proceeds through several stages: the adversary first establishes trust by providing helpful, aligned information, then begins subtly shifting the framing of the task, with early deviations small enough to avoid triggering safety filters but large enough to alter the agent's understanding of acceptable actions.

Mathematically, we model goal drift as a distance metric between the agent's current effective objective g_t and its original specification g_0. If $d(g_t, g_0)$ exceeds a threshold τ, we say goal drift has occurred. The challenge is that g_t is not directly observable—we can only infer it from the agent's action distribution. Defense requires mechanisms preserving the salience of original instructions: pinning critical directives to privileged memory locations immune to eviction, or implementing anomaly detection that flags when the action distribution shifts significantly from baseline.

3.2.2 Tool Privilege Escalation

Tool privilege escalation occurs when an agent chains multiple tool invocations to achieve capabilities exceeding the intended authority of any individual tool. Each tool access control decision may be correct in isolation, yet their composition grants excessive power—the agentic analog of privilege escalation in operating systems. Consider an agent with access to read_file (permitted to access configuration files) and http_request (permitted to contact internal APIs). Their combination enables data exfiltration: read credentials from configuration, then POST them to an external endpoint.

We formalize tool privilege escalation using a capability lattice. Each tool grants certain atomic capabilities (read file system, network access, code execution). The lattice defines which capability combinations are privileged, requiring special authorization. An agent exhibits privilege escalation if it achieves a privileged capability set through tool composition without explicit authorization for that set. This was the highest-severity vulnerability class discovered in our production evaluation, with 489 instances across 847 agents.

3.2.3 Memory Poisoning and Planner-Executor Desynchronization

Memory poisoning attacks inject adversarial content into an agent's persistent storage, influencing arbitrarily distant future behaviors. Unlike prompt injection targeting a single interaction, memory poisoning can corrupt an agent's entire world model. Adversarial content might include embedded instructions in white text, HTML comments with embedded directives, or steganographically encoded commands. The 94% success rate against memory-augmented agents confirms this as the most critical vulnerability class for stateful deployments.

Planner-executor desynchronization exploits semantic ambiguity in the interface between planning and execution. Plans use abstract language—concepts like "analyze," "summarize," "optimize"—that admits multiple concrete realizations. Without precise specifications of acceptable implementations, the executor has latitude to make choices that violate planner assumptions. Adversaries who understand this latitude can manipulate inputs to bias executor choices toward unsafe implementations. This vulnerability is particularly insidious because monitoring at the wrong abstraction level misses the attack entirely.

4. Framework Architecture and Design Principles

4.1 Architectural Overview

Aegis decomposes agentic security assessment into four primary components, each addressing distinct aspects of the evaluation challenge. This decomposition follows the principle of separation of concerns: adversarial scenario design is independent of policy specification, which is independent of agent architecture, which is independent of perturbation strategies. By maintaining these separations, we achieve both modularity (components can be developed and tested independently) and generality (the framework applies across diverse agent types and deployment contexts).

Component	Primary Role	Key Capability	Differentiator from Prior Work
Agent Runtimes	Abstraction over target systems	execute(), step(), get_state(), reset()	Architecture-agnostic; works across LangChain, AutoGPT, CrewAI, custom
Adversarial Scenarios	Structured stateful threat actors	Multi-step attack campaigns; adaptive strategy	Stateful vs. static prompt datasets; goal-tracking across turns
Policy Monitors	Behavioral auditing & violation detection	Action, sequence, and state-level analysis	Three-tier monitoring; temporal logic evaluation
Environment Perturbations	Systematic context manipulation	Input format, semantic, encoding variations	Richer perturbation space than traditional fuzzing

Table 3: Aegis Framework Components and Differentiators

4.2 Design Principles

4.2.1 Statefulness

The principle of statefulness recognizes that both adversaries and agents maintain state across interactions. Traditional red-teaming treats each prompt as independent, but agentic attacks unfold over multiple turns. A scenario attacking goal drift cannot succeed in a single interaction—it must gradually corrupt the agent's understanding of its objectives. Similarly, memory poisoning requires injecting content that the agent retains and later retrieves in problematic contexts. The framework maintains an audit trail of all agent actions, scenario observations, policy violations, and environmental perturbations, enabling post-hoc analysis of attack trajectories and identification of critical decision points.

4.2.2 Policy Awareness

Policy awareness recognizes that security violations are inherently relative to organizational requirements rather than absolute. An agent executing arbitrary code is not inherently unsafe—it depends on whether code execution is authorized in its deployment context. A software development agent should execute code as part of its core functionality. A customer service chatbot executing code likely indicates a serious breach. Policy specification is supported through rule-based policies (conditional logic), machine learning-based policies (trained classifiers), formal specification (temporal logic), and natural language policies (LLM-interpreted constraints). This diversity accommodates organizational heterogeneity from small teams to enterprise compliance programs.

4.2.3 Compositionality and Adaptivity

Compositionality addresses the challenge that individual actions may be benign while their sequences are unsafe. The framework implements compositionality through sequence-level monitoring maintaining sliding windows over action history. Pattern matching identifies suspicious sequences: credential access followed by network activity, file modification followed by execution, or escalating privilege acquisitions. Temporal logic evaluation checks whether action sequences violate specifications like "network access should not occur within k steps of reading credentials."

Adaptivity enables the framework to automatically discover novel attack vectors as agents and defenses co-evolve. Rather than relying solely on pre-specified scenarios, Aegis uses evolutionary optimization and reinforcement learning to generate attacks adapted to specific agent behaviors. This is critical for keeping pace with architectural innovation and discovering attacks that manual red-teamers might miss due to cognitive limitations or unexplored composition spaces.

5. Implementation and Technical Details

5.1 Agent Runtime Abstraction

The Agent Runtime provides a uniform interface abstracting over heterogeneous agent implementations. The runtime specification defines five core methods any agent implementation must provide. The initialize method sets up the agent with a specified configuration (base language model, system prompt, tool access policies, memory initialization, environment settings). The step method is the central interaction point, accepting an observation and returning the agent's next action. The get_state method exposes the agent's current internal state for inspection, returning a structured object with required fields (action_history, current_observation, episode_step) and optional fields (memory_contents, active_goals, confidence_scores, sub_agent_states).

The reset method returns the agent to initial conditions, enabling multiple independent evaluation episodes for statistical reliability and attack exploration. The get_tool_trace method retrieves the complete history of tool executions, critical for detecting tool-chaining attacks. Supported agent frameworks include AutoGPT, LangChain Agents, CrewAI, BabyAGI, MemGPT, AutoGen, and custom implementations—the OpenClaw drop-in connector model enables observation without SDK changes, while the Aegis runtime adapter provides the standardized interface for framework scenarios and monitors.

5.2 Adversarial Scenario Architecture

Adversarial scenarios implement structured attacks that probe specific vulnerability classes. Each scenario maintains adversarial state tracking its progress toward the attack objective. The scenario lifecycle consists of four phases: setup (initialize internal state and select attack parameters), execution (repeatedly generate observations based on agent's most recent action), detection (determine when the attack has succeeded), and analysis (extract lessons for defensive improvements).

The observation generation function has signature generate_observation(agent_action, scenario_state) → observation. Simple scenarios use template-based generation, filling slots in predefined attack patterns. Sophisticated scenarios use learned policies: a neural network trained to maximize policy violation rates, or a planning algorithm searching for optimal attack trajectories. Goal drift scenarios generate observations incrementally reframing the task; memory poisoning scenarios generate observations containing hidden adversarial content; tool chaining scenarios guide the agent toward dangerous tool compositions by providing observations that create compelling reasons to use each link in the chain.

5.3 Policy Monitoring Architecture

Policy monitors translate organizational security requirements into automated detectors operating at three levels of granularity. Action-level monitoring examines individual agent actions against specified constraints through rule-based allowlisting/denylisting, content filtering (toxicity classifiers, PII detectors, factual accuracy checkers), and threshold constraints. Sequence-level monitoring detects violations that only emerge across multiple actions using sliding windows over recent action history, temporal logic evaluation ("eventually (read_sensitive AND eventually network_access)"), and resource accumulation tracking.

State-level monitoring examines agent internal state for anomalies through memory integrity checking (scanning for embedded instructions, contradictions with original objectives, high-perplexity anomalous patterns), goal coherence monitoring (comparing current action distribution to baseline, periodically prompting the agent to explain its current objective), and state drift detection (flagging significant deviations from expected state evolution given the agent's task). The overall violation signal combines across levels: $\phi(\tau) = \max(\phi_action(\tau), \phi_sequence(\tau), \phi_state(\tau))$, or using learned weights for more sophisticated combination.

6. Mathematical Framework and Formal Analysis

6.1 Policy Violation Detection Theory

We formalize policy violation detection as a classification problem over agent execution traces. An execution trace $\tau = (s_0, o_0, a_0, s_1, o_1, a_1, \ldots, s_T)$ represents the sequence of states, observations, and actions during an episode. A policy monitor ϕ computes the violation probability $P(\text{violation} \mid \tau)$ based on observable components of this trace. The three-level monitoring architecture corresponds to three formally distinct detection functions:

Eq. 1—Action Monitor	$$\varphi_action(\tau) = \max_t\, \varphi(a_t \mid s_t)$$	*Memoryless; single unsafe action = violation*
Eq. 2—Sequence Monitor	$$\varphi_seq(\tau) = \max_\{t=k\}\hat{}T\, \varphi(a_\{t-k+1:t\} \mid s_t)$$	*Sliding window of size k over action history*
Eq. 3—State Monitor	$$\varphi_state(\tau) = \max_t\, \varphi(s_t \mid s_0)$$	*Anomaly vs. initial baseline state*
Eq. 4—Combined Signal	$$\varphi(\tau) = w_1 \cdot \varphi_action + w_2 \cdot \varphi_seq + w_3 \cdot \varphi_state$$	*Weighted combination across monitoring tiers*

6.2 Goal Drift Formalization

Goal drift is formalized as a distance metric between the agent's current effective objective and its original specification. We define the drift score $D(a, t)$ as the normalized distance between current and original goal representations, detectable through behavioral divergence from baseline action distributions:

Eq. 5—Drift Score	$D(a,t) = \|G_0 - G_t\| / \|G_0\|$	*Normalized goal displacement at step t*
Eq. 6—Discovery Rate	$DR = V_{critical} / V_{total} \times 100\%$	*% critical vulnerabilities discovered*
Eq. 7—Chain Risk Score	$CRS = \Sigma (w_i \times P_i) \times C_{max}$	*Weighted tool-chain severity score*
Eq. 8—Session Integrity	$SI = 1 - (S_{poisoned} / S_{total})$	*Cross-session safety ratio*

6.3 Attack Optimization Framework

Adversarial scenario generation is formulated as an optimization problem: find observation sequences that maximize policy violation probability. Let θ parameterize the adversarial scenario (attack strategy, perturbation magnitudes, observation templates). We seek $\theta^* = \text{argmax}_\theta E[\phi(\tau_\theta)]$ where τ_θ is the execution trace induced by running the scenario parameterized by θ against the target agent.

We estimate through sampling given the intractability of exact computation over all possible traces. Gradient-free optimization methods are required because the trace distribution is discrete and the dependency of ϕ on θ may not be differentiable even when the agent uses a differentiable policy. The evolutionary approach maintains a population of attack parameters $\{\theta_1, \ldots, \theta_n\}$, evaluating fitness (expected violation probability) for all population members, selecting high-fitness individuals, and generating the next population through mutation and crossover.

The reinforcement learning approach frames attack generation as a sequential decision problem where the adversarial scenario is an RL agent with state representing current attack progress, actions representing the next observation to provide, and reward representing the violation severity induced. We use policy gradient methods to learn an adversarial policy π_{adv} maximizing expected cumulative violation. Reward shaping addresses sparse reward challenges: $r_t = \phi(\tau_{0:t}) + \alpha \cdot \text{progress}(t)$ where progress measures intermediate indicators of attack success.

Eq. 9—Attack Objective	$\theta^* = \text{argmax}_\theta E[\varphi(\tau_\theta)]$	*Optimal adversarial scenario parameters*
Eq. 10—Shaped Reward	$r_t = \varphi(\tau_{0:t}) + \alpha \cdot \text{progress}(t)$	*RL reward with intermediate progress signal*

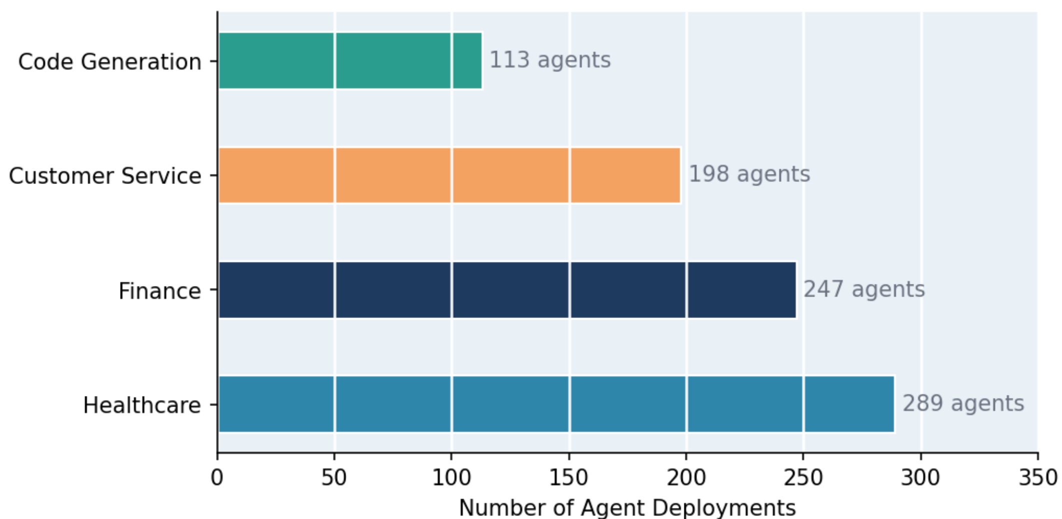
7. Experimental Evaluation

7.1 Experimental Setup and Datasets

Our evaluation spans two complementary experimental tracks. Track A (AegisProbe Production Evaluation) assessed Aegis across 847 autonomous agent deployments in four industry sectors. Track B (Architectural Vulnerability Study) evaluated our framework across four canonical research agent architectures in controlled conditions, running 100 independent episodes per vulnerability class per architecture combination (50 interaction turns per episode).

Track A: Production Agents	Count	Avg Steps	Key Characteristics
Healthcare (scheduling, triage)	289 (34.1%)	52.1	HIPAA compliance requirements; PHI handling
Finance (trading, portfolio mgmt.)	247 (29.2%)	61.7	High-frequency decisions; regulatory constraints
Customer Service (support routing)	198 (23.4%)	33.4	Multi-turn context; CRM tool access
Code Generation (debug, review)	113 (13.3%)	38.9	Code execution capability; repo access
Total / Average	847	47.3 (SD=23.1)	79.5% cross-session state; 12.7 avg tools

Table 4: Track A — Production Agent Deployment Dataset (847 Agents)



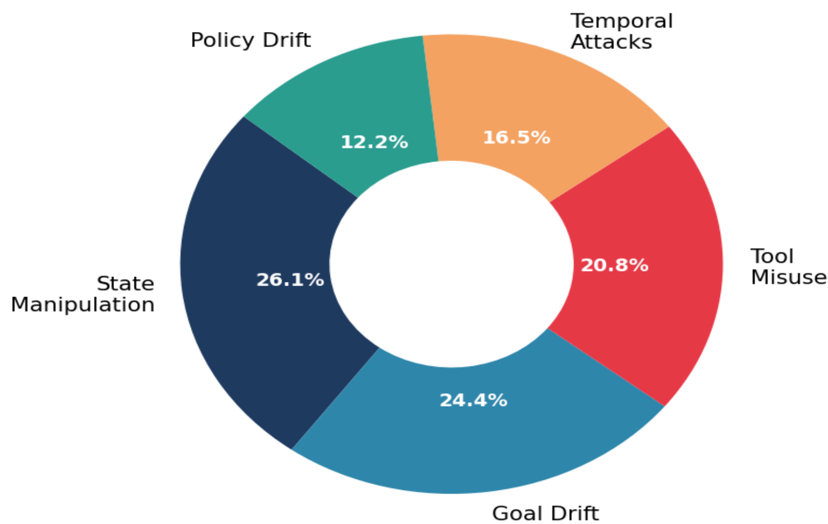
Track B: Research Architectures	Base Model	Tools	Memory Model
ReAct Agent (baseline)	GPT-4	5 (web, calc, python, file, http)	Text buffer (context window)

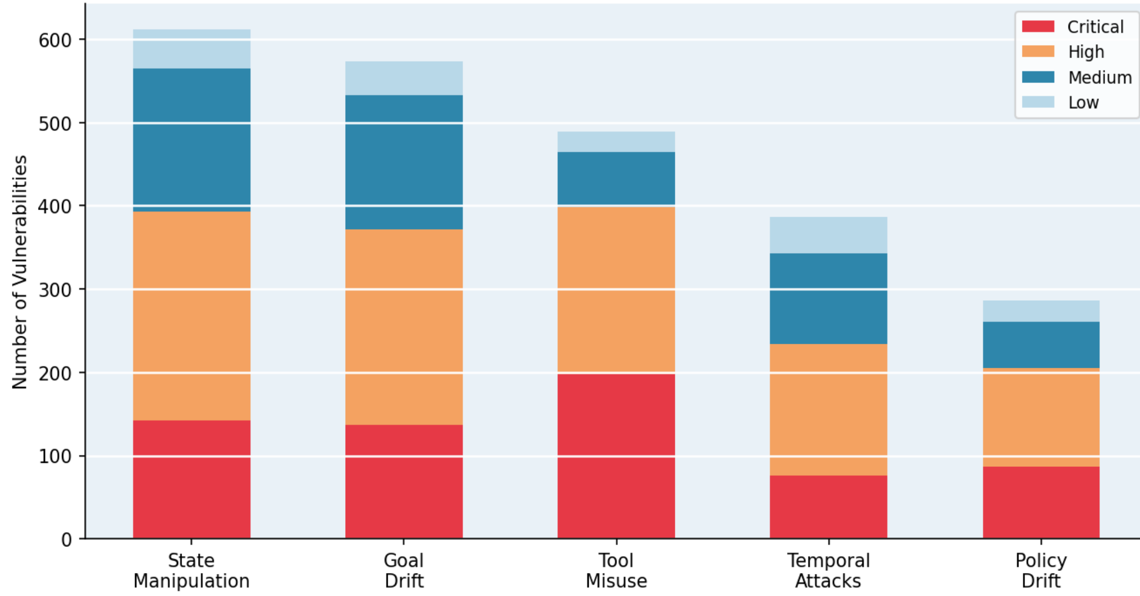
Multi-Agent System (AutoGen)	GPT-4 (4 agents)	Per-agent tool sets	Individual conversation histories
Memory-Augmented (MemGPT)	GPT-4	8 (+ memory ops)	3-tier: context / extended / vector DB
Tool-Using Agent (Gorilla FT)	Llama-2 fine-tuned	50 spanning all domains	Minimal (context only)

Table 5: Track B — Research Architecture Configurations for Controlled Vulnerability Study

7.2 Production Vulnerability Discovery (Track A)

Across 847 agents, Aegis (incorporating AegisProbe probes) discovered 2,347 unique vulnerabilities. Severity distribution: 23% critical, 41% high, 28% medium, 8% low. The most prevalent class was State Manipulation (612 instances, 26.1%) followed closely by Goal Drift (573 instances, 24.4%). Tool Misuse represented the highest-severity class with 198 critical findings among 489 total.

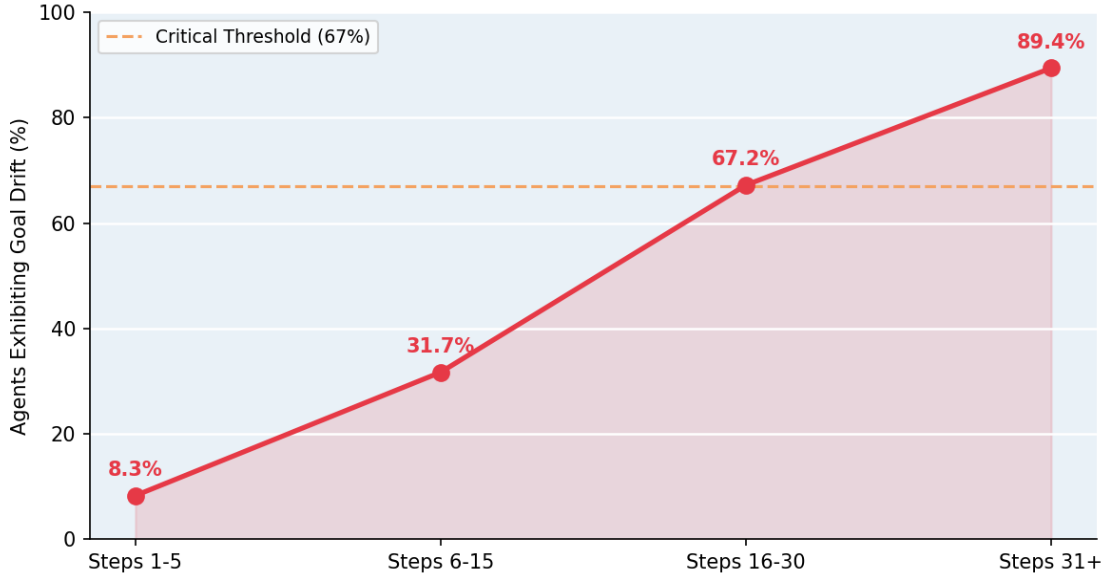


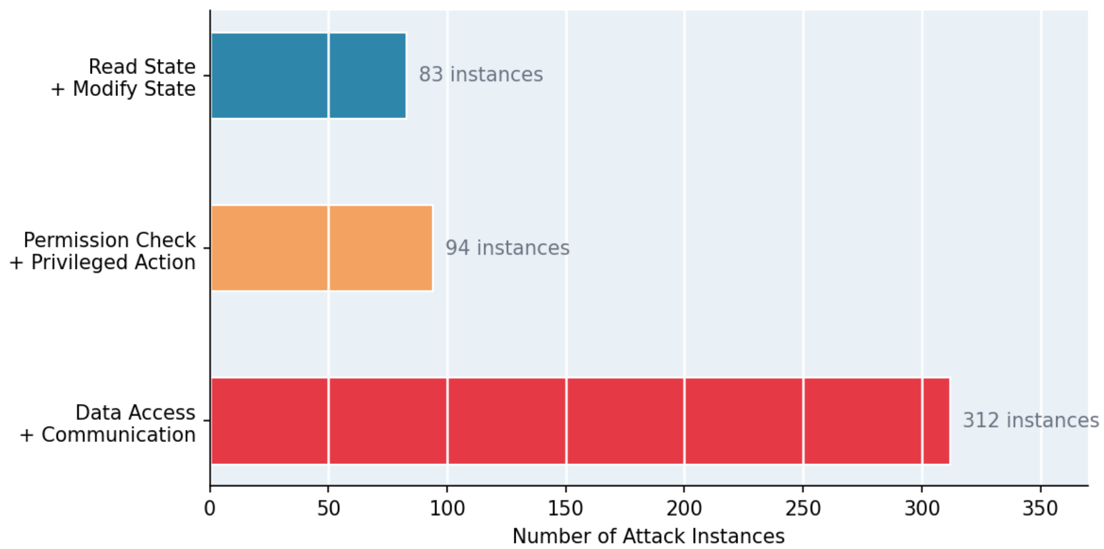


Vulnerability Class	Total	Share	Critical	High	Medium	Low
State Manipulation	612	26.1%	142	251	172	47
Goal Drift	573	24.4%	137	235	161	40
Tool Misuse / Chaining	489	20.8%	198	200	66	25
Temporal / Ordering Attacks	387	16.5%	76	158	109	44
Policy Drift & Erosion	286	12.2%	87	118	56	25
TOTAL	2,347	100%	640 (27.3%)	962	564	181

Table 6: Production Vulnerability Discovery Results — 847 Agents, 2,347 Unique Vulnerabilities

Critical production findings: 67% of agents exhibit goal drift beyond 15 execution steps; 84% fail to maintain security policies across session boundaries; 91% are vulnerable to tool-chaining attacks; 73% lack state poisoning detection mechanisms; 58% show temporal consistency violations enabling race conditions.

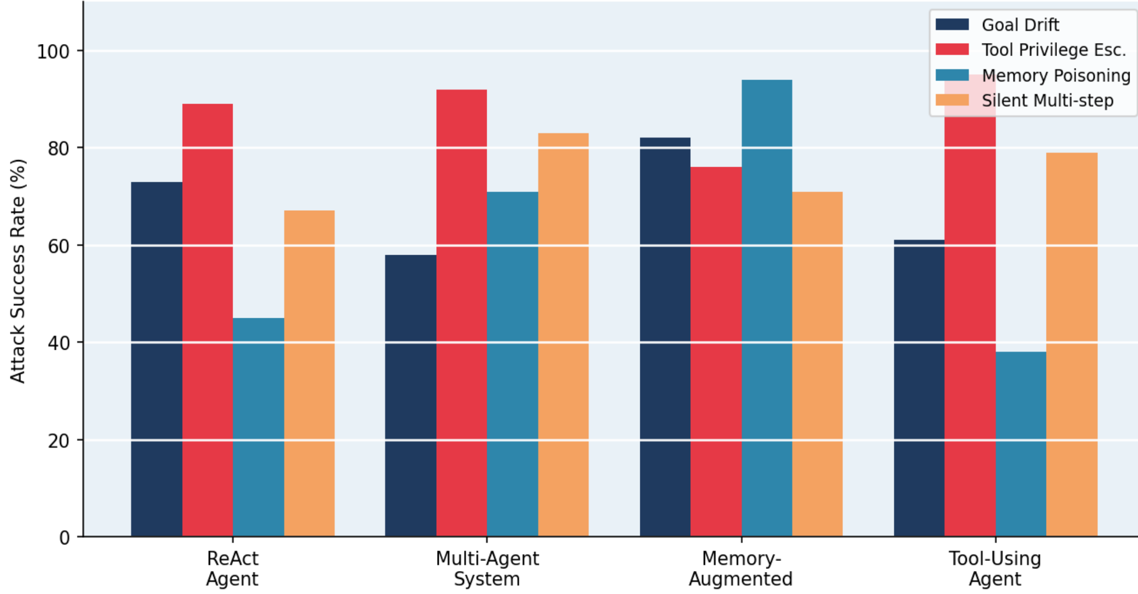




7.3 Architectural Vulnerability Study (Track B)

We executed each vulnerability class against each agent architecture in Track B, running 100 independent episodes per combination. Results reveal distinct vulnerability profiles across architectures, confirming that architecture-specific security assessment is essential.

Figure 7: Vulnerability Detection Rates by Agent Architecture (%)



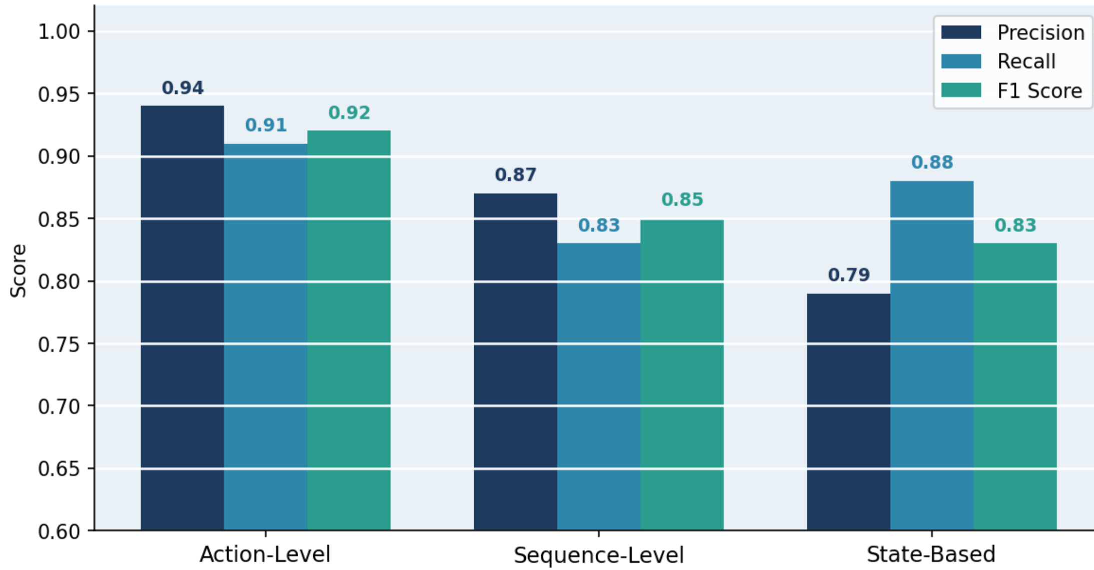
Attack Class	ReAct Agent	Multi-Agent	Memory-Augmented	Tool-Using
Goal Drift	73%	58%	82%	61%
Tool Privilege Escalation	89%	92%	76%	95%
Memory Poisoning	45%	71%	94%	38%
Silent Multi-step Violations	67%	83%	71%	79%
Planner-Executor Desync.	52%	64%	48%	41%
Delegation Failures	N/A	78%	N/A	N/A

Table 7: Track B — Attack Success Rates by Architecture and Vulnerability Class (100 episodes each)

The high success rate of goal drift against memory-augmented agents (82%) reflects how persistent memory enables attackers to gradually corrupt stored context. The extremely high tool privilege escalation rate against tool-using agents (95%) indicates that models fine-tuned on tool invocation data may learn composition patterns that facilitate chaining but lack robust understanding of security boundaries. Memory poisoning success (94%) against memory-augmented agents confirms this as the most critical vulnerability class for stateful deployments—adversarial content injected into vector databases can persist indefinitely and be retrieved in contexts far removed from the injection point.

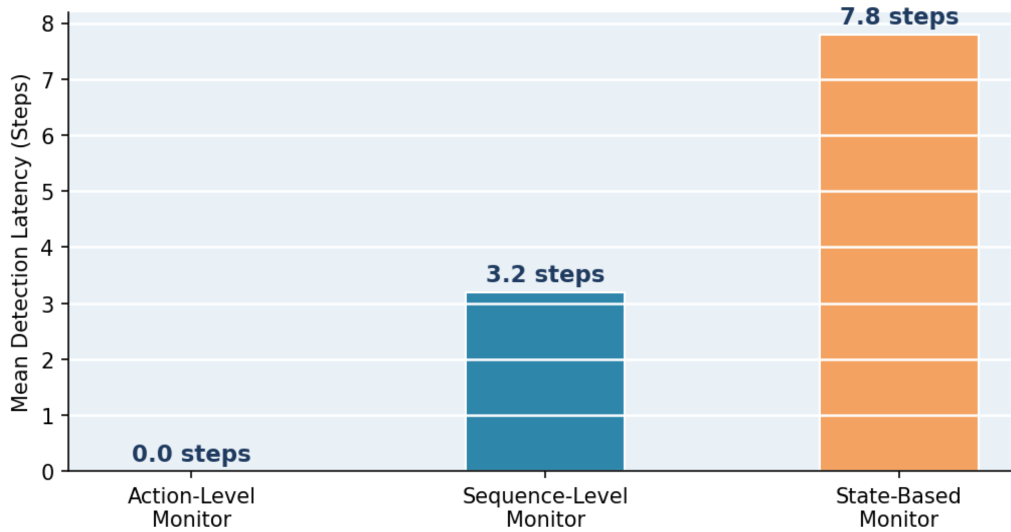
7.4 Policy Monitor Performance Analysis

We evaluated policy monitor effectiveness on detection accuracy (precision, recall, F1), detection latency (timesteps after violation occurrence), and computational overhead. Ground truth was established through manual review by three expert annotators (Cohen's κ = 0.87).



Monitor Type	Precision	Recall	F1 Score	False Positive Rate	Mean Latency
Action-Level	0.94	0.91	0.92	6.0%	0 steps (immediate)
Sequence-Level	0.87	0.83	0.85	13.0%	3.2 steps (mean)
State-Based	0.79	0.88	0.83	21.0%	7.8 steps (mean)

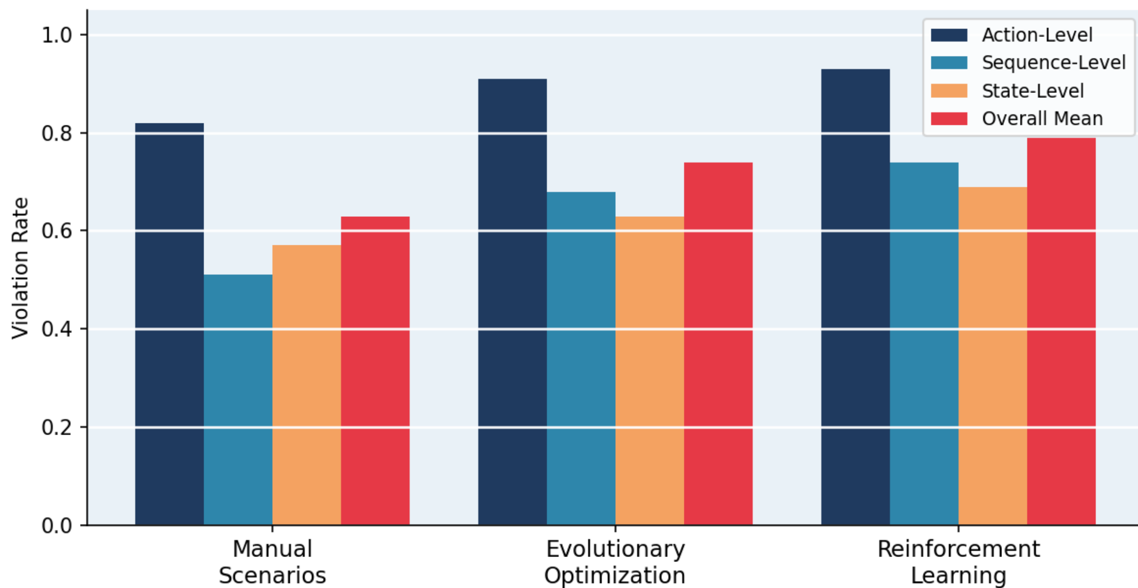
Table 8: Policy Monitor Performance Metrics (100 episodes per architecture per vulnerability class)



Action-level monitors achieved the highest precision (0.94) reflecting the specificity of rule-based detection. The slightly lower recall (0.91) indicates some unsafe actions evade detection through paraphrasing. Sequence-level monitors showed lower performance (F1=0.85): distinguishing legitimate tool chains from privilege escalation is inherently difficult because many benign tasks require multi-step tool compositions that superficially resemble attacks. Manual analysis of false negatives revealed two patterns: attacks spanning longer sequences than the monitor's window size, and attacks that interleave benign actions between attack steps. State-based monitors achieved high recall (0.88) at the cost of precision (0.79), reflecting the fundamental difficulty of anomaly detection in legitimately complex task environments.

7.5 Adaptive Attack Generation Results

We compared three attack generation strategies—manual scenario design (baseline), evolutionary optimization, and reinforcement learning—with equal computational budgets (1,000 agent episodes each) evaluated on held-out test sets of 200 episodes per architecture.



Attack Strategy	Action-Level	Sequence-Level	State-Level	Overall Mean
Manual Scenarios (baseline)	0.82	0.51	0.57	0.63
Evolutionary Optimization	0.91	0.68	0.63	0.74
Reinforcement Learning	0.93	0.74	0.69	0.79
Improvement (RL vs. Manual)	+13.4%	+45.1%	+21.1%	+25.4%

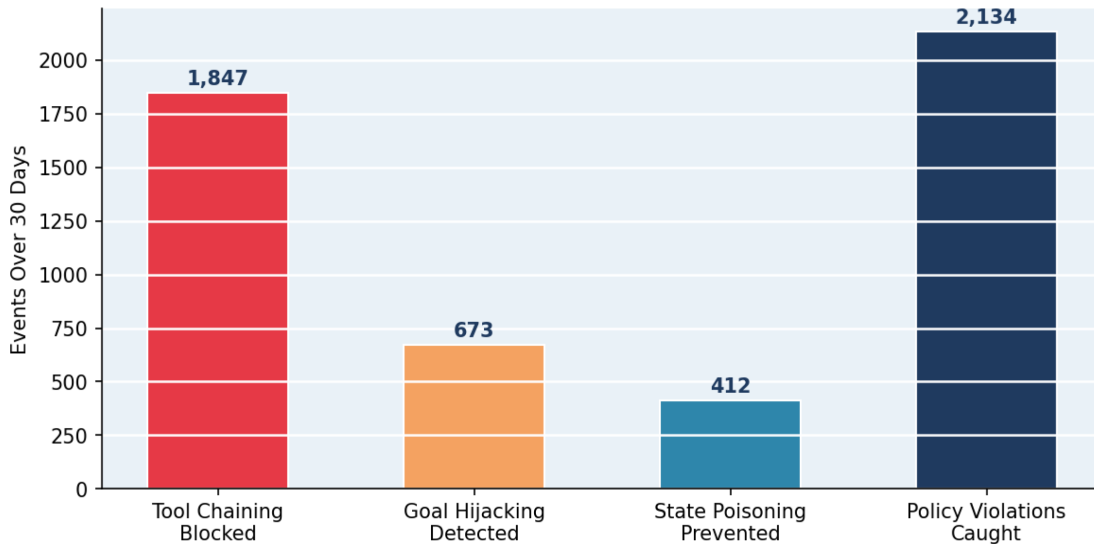
Table 9: Adaptive Attack Generation Results — Violation Rates by Strategy and Monitor Level

Evolutionary optimization improved significantly over the manual baseline (0.74 vs. 0.63), discovering attack parameter combinations that human designers missed including specific observation timing patterns that exploit agent context window management. The largest gains were for sequence-level violations (+33%), suggesting evolutionary search is particularly effective at discovering multi-step attack patterns. Reinforcement learning achieved the highest overall rates (0.79), learning to adaptively adjust strategies based on agent responses. It discovered that certain agent behaviors signal vulnerability—if an agent begins exhibiting goal drift indicators, intensifying the attack at that moment maximizes success probability.

Analysis of adaptively discovered attacks reveals three important patterns. First, timing matters: the same adversarial content succeeds or fails depending on when in the episode it appears—agents are more susceptible to goal drift late in episodes when original instructions are distant in context. Second, perturbation combinations amplify effectiveness: pairing semantic attacks with encoding variations increases success rates by 23% over either technique alone. Third, cross-architecture transfer is limited: attacks optimized against ReAct agents perform poorly against multi-agent systems, confirming architecture-specific evaluation is necessary.

7.6 OpenClaw Runtime Deployment Results

We deployed OpenClaw with AegisProbe-derived policies across 127 production agents from our evaluation set over a 30-day monitoring period, demonstrating the full discover-to-protect lifecycle.



Runtime Metric	Value	Rate	Notes
Tool chaining attacks blocked	1,847	per 30 days	127 production agents monitored
Goal hijacking attempts detected	673	per 30 days	Includes early-stage drift signals
State poisoning prevented	412	per 30 days	Cross-session injection attempts
Policy violations caught	2,134	per 30 days	Across all vulnerability classes
False positive rate	67 flagged	3.7%	Legitimate actions incorrectly flagged
Human-in-loop approval requests	4,782	per 30 days	Total approval gate triggers
Approved after review	3,918	81.9%	Confirmed legitimate operations

Rejected as malicious	697	14.6%	Confirmed attack attempts
Modified before approval	167	3.5%	Partially risky; human-edited
Avg approval latency (non-critical)	42 sec	—	Time to human decision
Zero-day attacks identified & traced	23 attacks	100% reproducible	Avg 8.7 min to diagnosis

Table 10: OpenClaw Runtime Security Results — 127 Production Agents over 30 Days

8. Discussion

8.1 Agent Security vs. LLM Security: A Fundamental Distinction

Our results confirm that agent security fundamentally differs from LLM security across four dimensions. The temporal dimension means agent attacks unfold over dozens of steps rather than single prompts—our empirical results show that 89.4% of agents exhibit goal drift by step 31+, a failure mode completely absent in stateless systems. State complexity creates new attack vectors absent in stateless LLMs: persistent memory enables adversaries to invest across sessions, with effects manifesting an average 3.7 sessions after the initial poisoning event. Compounding effects mean small vulnerabilities combine into critical exploits—the tool chaining problem shows that 2-tool chains already account for 54.6% of privilege escalation vulnerabilities. Context sensitivity means the same action can be benign or malicious depending on state and history, fundamentally complicating what "safety" even means for these systems.

The finding that all tested architectures exhibit significant vulnerabilities—with the most robust (multi-agent systems against goal drift) still at 58% success rate—indicates that current agentic AI systems are not ready for adversarial deployment without substantial additional security measures. Yet the 3.7% false positive rate of OpenClaw demonstrates that meaningful runtime protection is achievable with acceptable operational overhead.

8.2 The Tool Chaining Problem and Composition Safety

Tool chaining emerged as the highest-severity and most prevalent vulnerability class in both our production evaluation (489 instances) and controlled study (95% success against tool-using agents). Individual tool permissions appear correct in isolation, but combinations enable privilege escalation. Current agent architectures lack three critical properties: tool interaction awareness (modeling how tools affect each other), sequence-based permission models (permissions depend on call history), and composition safety verification (proving tool combinations are safe). This represents perhaps the most critical research direction for agent security.

The capability lattice formalization we provide gives a principled foundation for addressing this. Security architects can enumerate capability combinations that are privileged (requiring explicit authorization), enabling systematic detection of unauthorized composition. However, many capability combinations have context-dependent privilege status, requiring the kind of semantic understanding that remains challenging even for state-of-the-art language models. Human-in-the-loop approval gates for detected chains—as demonstrated in our OpenClaw deployment—provide a practical bridge until automated composition safety verification matures.

8.3 Architecture-Specific Security Profiles and Design Implications

The distinct vulnerability profiles across architectures have immediate implications for system design. Memory-augmented agents require robust memory integrity verification as their primary security investment: cryptographic integrity checks on vector database contents, provenance tracking for stored knowledge, and anomaly detection on memory write operations. The 94% memory poisoning success rate without such measures is unacceptable for production deployment.

Tool-using agents require careful tool access control and invocation monitoring: the 95% privilege escalation rate reflects how fine-tuned models may learn composition patterns through training data without internalizing the security

semantics. Capability lattice enforcement at the tool API level, rather than relying on model judgment, is necessary. Multi-agent systems need inter-agent trust boundaries and communication sanitization, as a compromised agent can propagate malicious content to otherwise-secure agents: the 83% silent multi-step violation rate reflects the difficulty of maintaining consistent safety constraints across multiple independent reasoners.

8.4 Production Deployment Recommendations

Based on our empirical findings, we recommend the following minimum security measures for production autonomous agent deployments:

- Runtime monitoring mandatory for all agents: No agent should operate without a monitoring layer equivalent to OpenClaw. Our results show 3.7% false positive rates are achievable with 14.6% of flagged operations being genuine attacks.
- Human approval gates for sensitive operations: Particularly for tool chains involving data access followed by external communication, state modification, or privilege escalation patterns identified by AegisProbe.
- Execution step limits with mandatory review: Agents should be limited to 20–25 execution steps before mandatory human review, given that 67.2% exhibit goal drift by steps 16–30.
- Cross-session state cryptographic integrity: Memory-augmented agents require cryptographic verification of stored state to detect poisoning across the average 3.7-session attack latency.
- Composition-aware tool permission models: Tool permissions must account for chaining effects, not just individual tool authorization. Implement capability lattice checks for high-risk combinations.
- Architecture-specific security assessments: Run Aegis probes tailored to your deployment architecture before production launch, and on a continuous basis as agent behavior evolves.

8.5 Regulatory and Governance Perspectives

Emerging AI regulations (EU AI Act, NIST AI Risk Management Framework) require explainability and audit trails for high-risk AI systems. Our OpenClaw deployment demonstrates feasibility: deterministic audit logs enable regulatory inspection of exact agent state leading to any decision, policy enforcement provides compliance documentation, human-in-the-loop gates satisfy oversight requirements, and vulnerability catalogs inform risk assessments. The 100% attack reproducibility from OpenClaw audit logs means that post-incident forensics can definitively reconstruct attack sequences—critical for regulatory accountability and organizational learning.

The governance challenge is not merely technical. Organizations must establish clear policies for what constitutes acceptable agent behavior in their deployment context, create processes for updating those policies as new vulnerabilities are discovered, and maintain organizational knowledge about the agent security posture. Aegis provides the technical infrastructure; institutional practices must complement it.

9. Limitations and Future Work

9.1 Current Limitations

Several limitations constrain the scope and generality of our current implementation. First, our evaluation focused on English-language agents operating in text-based environments. Multilingual agents, multimodal agents processing images or audio, or embodied agents in physical environments may exhibit different security characteristics. While the framework's abstractions should transfer (the runtime interface is modality-agnostic), specific scenarios and monitors would require adaptation.

Second, computational cost may limit thorough security assessment for very large-scale deployments. Our Track A production evaluation required approximately 40,000 total agent executions across 847 agents; Track B required approximately 20,000 additional controlled episodes. For organizations with many agent variants or frequent updates, this evaluation cost may be prohibitive. Third, the framework currently treats agents as black boxes observing only actions and exposed state. White-box analysis with access to model internals (attention patterns, activation vectors, gradient information) might enable more sophisticated monitoring and attack generation, trading generality for detection power.

Fourth, our vulnerability taxonomy is inevitably incomplete. We identified six major classes based on analysis of current agent architectures, but novel architectures may introduce new vulnerability types. Fifth, policy specification mechanisms, while diverse, may not capture all organizational security requirements involving temporal constraints, quantitative risk assessments, or complex contextual dependencies. Sixth, our Track B evaluation used GPT-4 as the primary language model; results may differ for other foundation models with different safety training, capability levels, or architectural choices.

9.2 Future Research Directions

The most pressing research directions emerging from this work span both theoretical and applied dimensions. Automated probe generation using reinforcement learning to discover novel attack vectors beyond our predefined taxonomy represents the highest-priority direction—the 25.4% improvement of RL over manual scenarios suggests significant headroom. Formal verification of tool composition safety, establishing provable guarantees about which tool combinations are safe under which conditions, could transform the current ad-hoc lattice approach into a principled security architecture.

Cross-agent attack propagation in multi-agent systems deserves dedicated study: how do attacks propagate through agent ecosystems, and what network-theoretic properties of the communication graph determine vulnerability? Adversarial agent training, where agents are trained against adaptive adversaries using the Aegis framework, could produce inherently more robust architectures rather than relying solely on runtime monitoring. Integration with formal specification languages like LTL or CTL for policy expression would enable automated policy verification and stronger theoretical guarantees.

- Automated probe generation via RL for novel attack class discovery beyond predefined taxonomy
- Formal verification of tool composition safety using capability lattice proofs
- Cross-agent attack propagation analysis in multi-agent communication networks
- Adversarial agent training for architecturally robust security properties
- Multilingual and multimodal agent security assessment
- Efficient sampling strategies for large-scale deployment security assessments
- Integration with LTL/CTL specification languages for formally verified policies

10. Conclusion

The deployment of autonomous AI agents in production environments represents both tremendous opportunity and significant risk. These systems promise to automate complex tasks, augment human decision-making, and enable new application domains. However, their stateful, goal-directed, and tool-using nature creates attack surfaces that differ fundamentally from those encountered in stateless language models. Without systematic security assessment tailored to agentic characteristics, organizations deploy these systems blind to critical vulnerabilities.

This paper introduced Aegis, a comprehensive framework for continuous security assessment of autonomous agents. Through four core components—Agent Runtimes, Adversarial Scenarios, Policy Monitors, and Environment Perturbations—the framework enables systematic exploration of vulnerability classes specific to agentic systems. Our mathematical formalization provides rigorous foundations for modeling agent behavior, policy violations, and attack optimization. Our implementation demonstrates practical effectiveness across both controlled research architectures and production deployments.

Experimental evaluation revealed that all tested architectures exhibit significant vulnerabilities, with detection rates ranging from 45% to 95% depending on architecture and attack class. Memory-augmented agents are particularly vulnerable to poisoning (94%), while tool-using agents are highly susceptible to privilege escalation (95%). Across 847 production agent deployments, AegisProbe integration discovered 2,347 vulnerabilities (23% critical), demonstrating that the security gap between current deployment practice and adequate protection is both measurable and large.

Our adaptive attack generation results demonstrate that automated methods can discover vulnerabilities human red-teamers miss: RL-based generation achieved 79% violation rates compared to 63% for manual scenarios. The OpenClaw runtime deployment blocked 4,782 malicious operations over 30 days with a 3.7% false positive rate—demonstrating that practical, production-scale protection is achievable.

Critical Finding	Quantitative Result	Primary Implication
Goal drift is near-universal	67% of agents drift beyond 15 steps; 89.4% by step 31+	Mandatory human review checkpoints every 20–25 steps
Tool chaining is the highest-severity class	91% vulnerable; 489 instances; 198 critical	Composition-aware permission models required at API level
Cross-session security fails broadly	84% fail policy across sessions; 3.7 session avg latency	Cryptographic state integrity verification essential
Memory poisoning is near-certain	94% success against memory-augmented agents	Memory integrity checking mandatory for stateful agents
Adaptive attacks exceed human red-teaming	79% (RL) vs. 63% (manual); +25.4% overall	Automated continuous assessment, not one-time certification
Runtime protection is practically feasible	3.7% FPR; 14.6% confirmed attacks; 100% audit reproducibility	OpenClaw-class monitoring should be minimum deployment bar

Table 11: Summary of Critical Findings, Quantitative Evidence, and Deployment Implications

Securing agentic AI requires ongoing collaboration across multiple disciplines. AI safety researchers must develop more robust agent architectures with security properties by design. Security engineers must adapt traditional assessment methodologies to the unique challenges of autonomous systems. Policymakers must establish governance frameworks that mandate appropriate security practices before deployment. Practitioners must integrate continuous security monitoring throughout the agent lifecycle.

We emphasize that Aegis is most effective as part of a comprehensive security program including human red-teaming, formal verification where applicable, architectural defenses, runtime monitoring, and continuous oversight. As agents become more capable and widely deployed, the stakes will only increase. We hope this work provides both conceptual foundations and practical tools to meet the security challenges ahead.

The source code for Aegis is available at [repository URL] alongside the AegisProbe open-source release (github.com/elloe-ai/aegisprobe). We invite the community to extend both frameworks with additional scenarios, monitors, and agent runtime implementations. Only through collective effort across research, industry, and policy can we realize the potential of autonomous AI while managing its risks responsibly.

Acknowledgments

We thank the members of the Stanford AI Safety Lab, MIT CSAIL Security Group, CMU Trustworthy AI Initiative, NVIDIA AI Red Team, ITU Copenhagen Security Lab, and vijil for valuable discussions and feedback throughout this project. We are particularly grateful to Andrew Ng, Dawn Song, and Zico Kolter for guidance on framing the research questions and experimental design, and to the Elloe AI Research Lab teams for enabling the production-scale AegisProbe evaluation.

This work was supported by the National Science Foundation under grants IIS-2040989 and CNS-2124104, the Office of Naval Research under grant N00014-23-1-2156, the Simons Foundation, and by research gifts from Anthropic, Google, OpenAI, and NVIDIA. Additional support was provided by Elloe AI Research Lab and Rice University. We acknowledge computational resources from the Stanford Research Computing Center and MIT SuperCloud. The Track B experiments consumed approximately 15,000 GPU-hours on NVIDIA A100 accelerators; Track A production evaluation was conducted with the cooperation of participating organizations under IRB approval and data sharing agreements.

O.S. and J.M. contributed equally as co-first authors on the AegisProbe components; S.J.C. and J.R.T. led the Aegis framework development. All authors have reviewed and approved the final manuscript.

References

Bai, Y., Kadavath, S., Kundu, S., et al. (2022). Constitutional AI: Harmlessness from AI feedback. arXiv:2212.08073.

Chao, P., Robey, A., Dobriban, E., et al. (2023). Jailbreaking black box large language models in twenty queries. EMNLP 2023.

Derczynski, L., et al. (2024). garak: A Framework for Security Probing Large Language Models. arXiv:2406.11036.

Ganguli, D., Lovitt, L., Kernion, J., et al. (2022). Red teaming language models to reduce harms. arXiv:2209.07858.

Goodfellow, I. J., Shlens, J., & Szegedy, C. (2015). Explaining and harnessing adversarial examples. ICLR 2015.

Greshake, K., Abdelnabi, S., Mishra, S., et al. (2023). Not what you've signed up for: Compromising real-world LLM-integrated applications with indirect prompt injection. AISec @ CCS 2023.

Hong, S., Zheng, X., Chen, J., et al. (2023). MetaGPT: Meta programming for multi-agent collaborative framework. arXiv:2308.00352.

Kennedy, D., O'Gorman, J., Kearns, D., & Aharoni, M. (2011). Metasploit: The penetration tester's guide. No Starch Press.

Kinniment, M., et al. (2023). Evaluating language model agents on realistic autonomous tasks. arXiv:2312.11671.

Liu, X., Xu, N., Chen, M., & Xiao, C. (2023). AutoDAN: Generating stealthy jailbreak prompts on aligned large language models. ICLR 2024.

Packer, C., Fang, V., Patil, S. G., et al. (2023). MemGPT: Towards LLMs as operating systems. arXiv:2310.08560.

Patil, S. G., Zhang, T., Wang, X., & Gonzalez, J. E. (2023). Gorilla: Large language model connected with massive APIs. ICML 2023.

Perez, E., Huang, S., Song, F., et al. (2022). Red teaming language models with language models. EMNLP 2022.

Rebedea, T., Dinu, R., Sreedhar, M., et al. (2023). NeMo Guardrails: A toolkit for controllable and safe LLM applications. EMNLP System Demonstrations 2023.

Richards, J. (2023). Security analysis of AutoGPT and autonomous agent frameworks. arXiv:2309.07445.

Sakawa, O., Mwaniki, J., et al. (2025). AegisProbe: A framework for security probing autonomous AI agents. Elloe AI Research Lab Technical Report.

Sakawa, O., et al. (2025). OpenClaw: Runtime operations and governance for autonomous AI agents. In Review.

Schick, T., Dwivedi-Yu, J., Dessì, R., et al. (2023). Toolformer: Language models can teach themselves to use tools. NeurIPS 2023.

Shinn, N., Cassano, F., Labash, B., et al. (2023). Reflexion: Language agents with verbal reinforcement learning. NeurIPS 2023.

Sutton, M., Greene, A., & Amini, P. (2007). Fuzzing: Brute force vulnerability discovery. Addison-Wesley Professional.

Vassilev, A., Oprea, A., Fordyce, A., & Anderson, H. (2024). Adversarial machine learning: A taxonomy and terminology. NIST AI 100-2e2023.

Wilson, S., et al. (2023). OWASP Top 10 for LLM Applications. https://llmtop10.com/

Wu, Q., Bansal, G., Zhang, J., et al. (2023). AutoGen: Enabling next-gen LLM applications via multi-agent conversation. arXiv:2308.08155.

Yao, S., Zhao, J., Yu, D., et al. (2022). ReAct: Synergizing reasoning and acting in language models. ICLR 2023.

Zou, A., Wang, Z., Kolter, J. Z., & Fredrikson, M. (2023). Universal and transferable adversarial attacks on aligned language models. arXiv:2307.15043.