SECURING AGENTIC AI

Aigos AI Security Blueprint Series

aigos

Securedbyaigos.com

Contents

Introduction

What is agentic AI security?

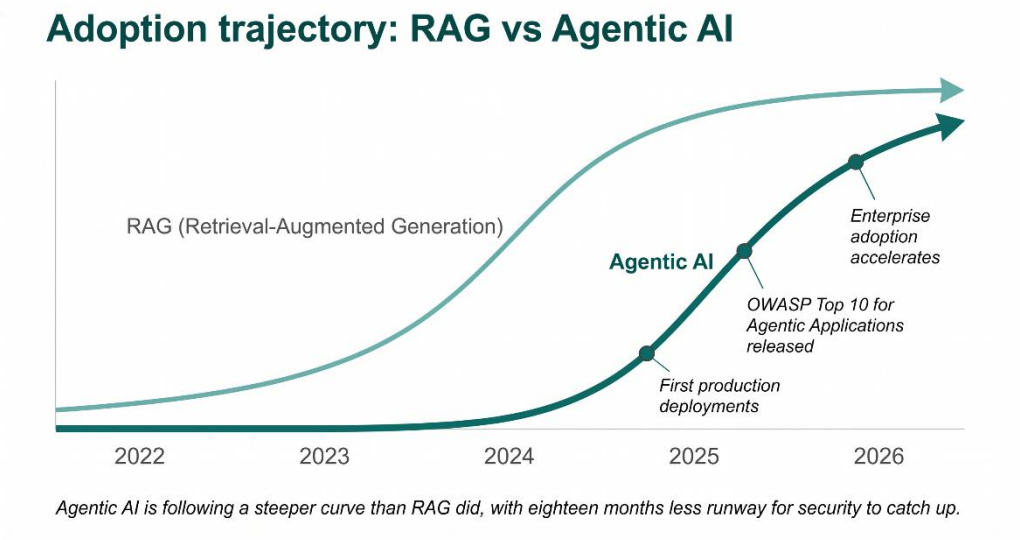
Agentic AI security refers to the practice of identifying, mitigating, and monitoring risks introduced by artificial intelligence systems that plan, decide, and execute multi-step actions on behalf of human operators. Where conversational AI generates text and traditional retrieval systems return information, agentic AI takes action: it runs commands, calls APIs, modifies files, sends messages, and orchestrates other systems with increasing autonomy. This shift from generation to execution introduces an entirely new attack surface that traditional application security controls were not designed to address.

Securing agentic AI is therefore not a single discipline. It draws on application security for input validation, on identity and access management for credential governance, on endpoint security for runtime monitoring, and on adversarial machine learning for the model-specific failure modes. The discipline that emerges combines elements of all four into something genuinely new.

Why this matters now

The parallel to retrieval-augmented generation (RAG) in 2023 and 2024 is instructive. RAG moved from research curiosity to enterprise deployment in roughly eighteen months, and the security conversations followed afterwards: how to handle vector embeddings, prevent prompt injection through retrieved content, and stop sensitive data from flowing into third-party model providers. Agentic AI is on a steeper trajectory. Frameworks that did not exist in production form eighteen months ago — Goose, OpenClaw, Hermes Agent, Claude Code, Claude Cowork — now operate inside development workflows, customer support pipelines, and internal automation across the Fortune 500.

The risk profile has changed accordingly. A chatbot that hallucinates returns a wrong answer. An agent that hallucinates can run destructive shell commands before anyone reads the response. The same model, given different scaffolding, becomes either a productivity tool or a privileged actor with the keys to production systems. The OWASP Top 10 for Agentic Applications, released in December 2025 and widely adopted in early 2026, validates what defenders have observed in production: agentic failures rarely look like bad output. They look like bad outcomes.



Agentic AI is following a steeper curve than RAG did, with eighteen months less runway for security to catch up.

Beyond traditional application security

This is where agentic security departs most sharply from familiar territory. Traditional application security is concerned with deterministic systems: a user submits an input, a known function processes it, and a predictable output emerges. The defender's job is to validate inputs, enforce authentication, and audit access. Agentic systems break every assumption in that model. The "input" is natural language and may contain hidden instructions. The "function" is a non-deterministic model that can chain tools in sequences the developer never anticipated. The "output" is a series of real-world actions taken under delegated authority, sometimes across multiple sessions.

Traditional perimeter security, endpoint detection, and even existing LLM guardrails were not designed for systems that autonomously chain actions across multiple services. A 2026 Dark Reading poll identified agentic AI as the number-one anticipated attack vector for 2026, outranking deepfakes, ransomware, and supply chain compromise. Yet only 34 percent of enterprises reported having AI-specific security controls in place. The gap is operational. The frameworks and patterns exist; the tooling to enforce them does not yet match the speed of agent adoption.

The Board, CTO and CISO challenge

Agentic security is a board-level, CTO-level, and CISO-level concern in three distinct organisational contexts.

For organisations building agentic products externally, security posture is now a procurement requirement. Enterprise buyers ask the same questions they ask of SaaS vendors: where does data go, what tools can the system invoke, how is access scoped, what happens when an action goes wrong. The vendor without good answers does not close.

For organisations adopting agentic AI internally, the calculus is different but the stakes are similar. A development team running Claude Code or Goose against a production codebase has effectively granted shell access to a non-human actor, with consequences ranging from accidental file deletion to credential leakage in the next API call. The same tools that improve developer velocity expand the blast radius of any single mistake.

For organisations on the receiving end of agentic traffic, even those not deploying agents themselves, public-facing systems now face autonomous traffic whose goals may have been hijacked, whose tools are being misused, or whose behaviour has drifted from its declared purpose while still appearing legitimate. This is the case OWASP frames as the dual concern: builders need to constrain agents from the inside, and defenders need to recognise agentic traffic from the outside.

The remainder of this document examines the agentic framework landscape, the OWASP risk taxonomy as it maps to operational decisions, and the architectural choices defenders face when designing controls for agents that they themselves do not build.

The Agentic Framework Landscape

Five agentic frameworks anchor the 2026 conversation, each occupying a distinct position between developer-tool and enterprise-platform, and between local-execution and cloud-orchestrated. A sixth, Claude Code, sits adjacent to the group as the dominant CLI agent in software engineering workflows. Understanding their differences matters because the architectural choices they make become the constraints that defenders inherit.

Goose

Backed by Block and now governed by the Linux Foundation through AAIF, Goose is the developer-focused framework most often described as the open-source counterpart to Claude Code. It runs as a local CLI agent, uses Model Context Protocol (MCP) extensions for tool access, and is written in Rust for performance. Its security model leans heavily on user discretion: the agent has shell access, writes files, and connects to development environments under the operator's user account. Recent additions include a ToolMonitor primitive and an egress logging inspector, indicating the maintainers are actively building the security infrastructure the framework currently depends on the user to provide.

OpenClaw

Distributed as an MIT-licensed open-source project with a chat-native interface (Telegram, WhatsApp, others), OpenClaw optimises for fast web automation via Playwright and is widely deployed for personal automation workflows. Its default open posture makes it powerful but also exposes a meaningful attack surface; OpenClaw was the subject of CVE-2026-25253, a CVSS 8.8 remote code execution disclosed earlier this year, and Cisco's AI security team publicly demonstrated a malicious skill exfiltrating data through the official skill registry. Red Hat engineers responded by packaging OpenClaw as a bootable Tank OS image that runs each agent inside an isolated Podman container — an enterprise security layer the project itself never shipped.

Hermes Agent

From Nous Research, Hermes Agent takes a different architectural bet: long-term automation with built-in learning loops that rewrite skill documents based on past failures. Latency runs roughly 30 percent slower than OpenClaw because of its larger context windows and SQLite FTS5 recall, but one-shot success rates on complex web tasks are higher. Its security posture includes command approval flows, dangerous-pattern blocking, and Docker container isolation by default — a more restrictive baseline than OpenClaw's, reflecting its positioning toward unattended server-side automation.

NemoClaw

NVIDIA's enterprise variant is the most aggressively secured of the five. The OpenShell runtime acts as an infrastructure-level sandbox enforcing four layers of isolation (network, filesystem, process, and inference) and a default-deny network policy where outbound requests to unapproved endpoints are blocked and routed to an operator's terminal for human-in-the-loop approval. This security comes at a cost: NemoClaw is the slowest of the five "by design", as the security checks intentionally throttle execution speed. For regulated industries — finance, healthcare, defence — that tradeoff is the point.

Claude Cowork

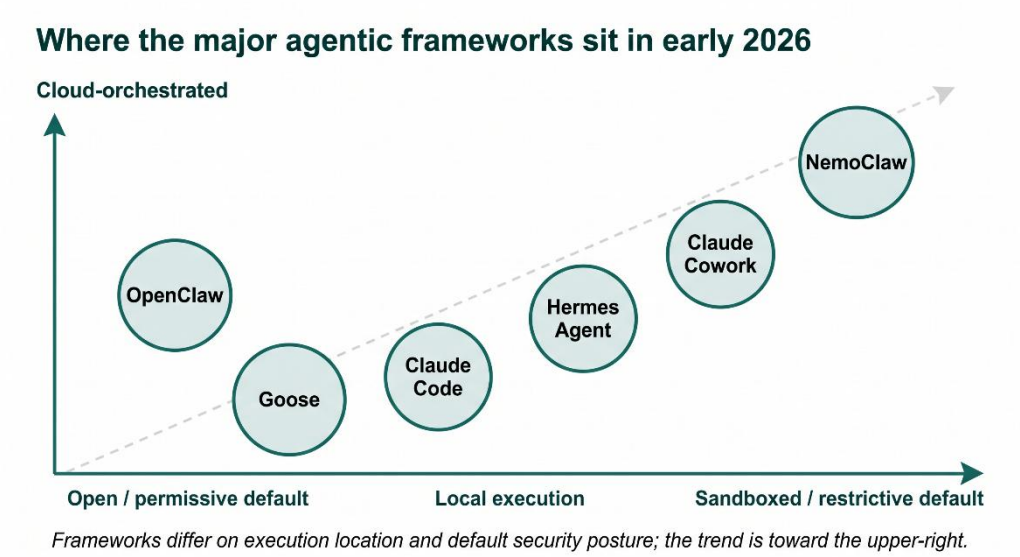
Anthropic's collaborative desktop assistant occupies the human-in-the-loop corner. It operates inside an isolated VM workspace alongside the user, navigates desktop applications visually, and uses a Sentry permission module with seven independent safety layers and granular allow/disallow lists. Anthropic recently introduced a Bring Your Own Cloud architecture that lets enterprises deploy the Cowork harness while routing model inference to their own AWS Bedrock, Google Cloud Vertex AI, or Azure AI Foundry tenants — a meaningful concession to data residency requirements that have been the primary obstacle to enterprise adoption.

Claude Code

Also from Anthropic, Claude Code sits adjacent to this group as the command-line agent for software development tasks. It is the most widely deployed of the agents that combine LLM reasoning with direct shell access in software engineering workflows, and its PreToolUse and PostToolUse hook system has become a de facto reference architecture for agent integration security across the broader landscape. The hook system supports both pre-execution gating (deny dangerous commands before they run) and post-execution transformation (modify tool outputs before they return to the model), which makes it structurally well-suited to the dual filtering approach discussed later in this document.

The frameworks differ on five axes that matter for security: where they execute (local versus cloud), what they can touch (CLI versus web versus desktop versus enterprise APIs), how they handle memory (ephemeral versus persistent, plaintext versus governed), what learning they perform (none versus implicit versus skill-rewriting), and what their default security posture assumes (open versus sandboxed versus default-deny). No single framework is correct for every deployment; the right choice depends on the threat model and the regulatory context.

What unifies them, and what makes the OWASP Top 10 a useful map across the differences, is the underlying loop. Each framework has an agent that reasons, a set of tools the agent can invoke, a memory or context store, and a decision boundary between "what the agent wants to do next" and "what actually happens". The risks identified in OWASP ASI 2026 manifest at consistent points in this loop regardless of which framework instantiates it.



Frameworks differ on execution location and default security posture; the trend is toward the upper-right.

The OWASP Top 10 for Agentic Applications: A CISO's Reading

The OWASP Top 10 for Agentic Applications 2026 was released in December 2025 after peer review by more than 100 industry experts. It is the first industry-standard taxonomy dedicated to autonomous AI agents, and it builds on but extends beyond the OWASP Top 10 for Large Language Models. The full list is summarised below, with the canonical example incident for each category included where one is publicly documented.

- **ASI01: Agent Goal Hijack —** attackers manipulate an agent's objectives through poisoned inputs (emails, PDFs, calendar invites, web content, RAG documents). The agent cannot reliably separate instructions from data, so a single malicious input can redirect it to perform harmful actions using its legitimate access. The EchoLeak incident (Microsoft 365 Copilot, 2025) is a documented example where a hidden email payload caused silent exfiltration of confidential content.
- **ASI02: Tool Misuse and Exploitation —** agents apply legitimate tools in unsafe or unintended ways. Examples include over-privileged tools that can write to production systems, poisoned tool descriptors in MCP servers, or shell tools that run unvalidated commands. The Amazon Q incident is the canonical real-world reference here.
- **ASI03: Identity and Privilege Abuse —** agents inherit user sessions, reuse secrets, or rely on implicit cross-agent trust, leading to privilege escalation and attribution gaps. An agent operating with the full authority of every key, token, and service account assigned to it becomes an aggregation point for non-human identities.
- **ASI04: Agentic Supply Chain Vulnerabilities —** malicious or compromised models, tools, plugins, MCP servers, or prompt templates introduce hidden instructions and backdoors at runtime. Unlike static supply chain risk, this is dynamic: components are discovered and integrated during execution. The GitHub MCP exploit is a recent example.
- **ASI05: Unexpected Code Execution —** agents generate or execute attacker-controlled code. Natural-language execution paths create new avenues for remote code execution that traditional application security tooling does not anticipate. The AutoGPT RCE class of vulnerabilities is the canonical reference.
- **ASI06: Memory and Context Poisoning —** persistent corruption of agent memory, retrieval-augmented generation stores, or contextual knowledge. Behaviour reshapes long after the initial interaction; the Gemini Memory Attack disclosed earlier this year demonstrated that a single poisoned interaction could persist across subsequent sessions.
- **ASI07: Insecure Inter-Agent Communication —** spoofed, manipulated, or intercepted messages between agents. Multi-agent workflows assume trust between participants; that assumption breaks when an attacker can impersonate one of them.
- **ASI08: Cascading Failures —** single-point faults propagate through multi-agent workflows at scale. One agent's hallucination becomes another agent's input, and the error compounds across the pipeline.
- **ASI09: Human-Agent Trust Exploitation —** agents appear confident and authoritative; humans trust their recommendations without independent verification. Attackers exploit this trust to induce credential sharing, risky changes, or financial transfers.
- **ASI10: Rogue Agents —** compromised or misaligned agents diverge from intended behaviour while continuing to appear legitimate. They may self-repeat actions, persist across sessions, or impersonate other agents. Detection is genuinely difficult once an agent has drifted from its intended purpose.

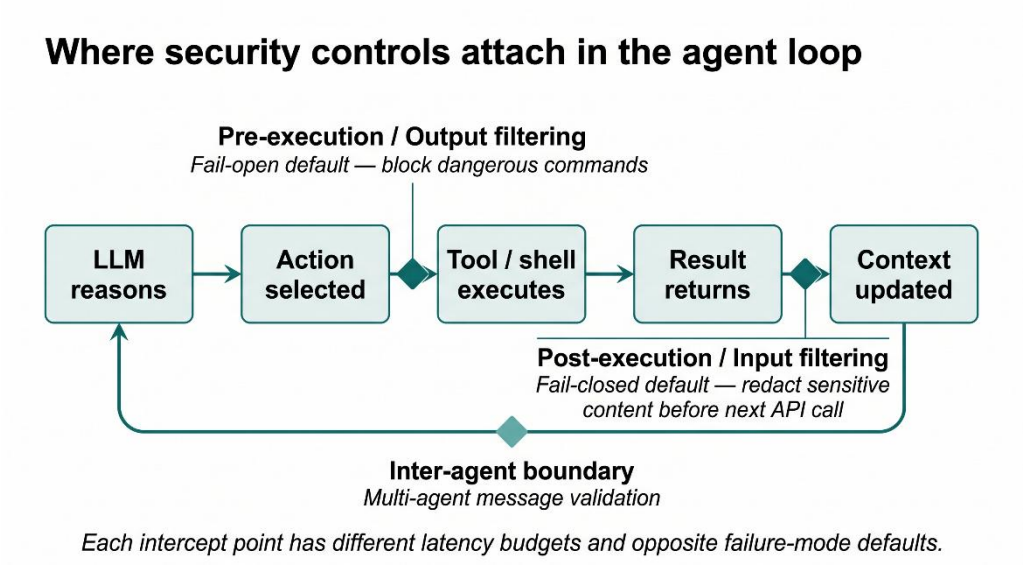
Reading the list as a whole, two observations stand out for security leaders.

The first is the principle of least agency, OWASP's framing-level guidance that agents should be granted the minimum autonomy required to perform their intended task. This is the agentic equivalent of least privilege, and the framework explicitly positions it alongside strong observability as the foundational defensive posture. The two are paired deliberately. Least agency without observability is blind constraint. Observability without least agency is surveillance of harm in progress. The point of combining them is to give agents the narrowest possible scope and to enforce that scope when behaviour drifts.

The second observation is that the highest-impact risks in the list — ASI01 (Goal Hijack), ASI02 (Tool Misuse), ASI05 (Code Execution) — manifest at the boundary between the model's reasoning and the system's action. The agent decides to do something; the system permits or blocks it. Whatever security control a defender wants to apply, that boundary is where it has to attach. We turn next to where exactly, in the agent loop, that attachment happens.

Mapping Risks to the Agent Loop: Where Controls Attach

Every agentic framework, regardless of its specific architecture, runs the same fundamental loop: the agent reasons, the agent decides on an action, the system executes the action, the result of the action returns to the agent's context, and the agent reasons again. The loop varies in detail. Goose's MCP-based tool calls differ from Claude Cowork's VM-based desktop interactions. But the structural points are consistent. Three of those points are critical for security.



Pre-execution: the output filtering boundary

The agent has decided on an action but has not yet executed it. The system has the candidate command, tool call, file path, or API request in hand, and the question is whether to permit it. This is the natural attachment point for output filtering, controls that examine what the agent is about to do and decide whether to allow it. ASI02 (Tool Misuse) and ASI05 (Code Execution) live here primarily.

Pre-execution checks have the strictest latency constraint. The agent is paused waiting for permission. Every millisecond of delay translates into perceived friction for the developer or end user. A control that adds two seconds of latency to every shell command will be disabled within a week. The defensive posture at this point is to fail open: if the safety check itself fails or is unavailable, the command runs. Failing closed at pre-execution would mean breaking the user's workflow whenever the safety system has a bad day, and the result is well-documented: the user disables the safety system.

Post-execution: the input filtering boundary

The action has run, the result has returned, and that result is about to be incorporated into the agent's context for the next reasoning step. For tools that read content (file reads, log queries, search results, API responses), this is the moment when sensitive content first crosses from the local system into the model's context, and from there, in nearly all current architectures, into an outbound API call to the model provider. This is the attachment point for input filtering. ASI01 (Goal Hijack via poisoned content) and ASI06 (Memory Poisoning) manifest at this boundary.

Post-execution checks face a different constraint. Failing open here is dangerous, because the failure mode is "raw tool output flows to the model unredacted", which is precisely the scenario the control exists to prevent. By the time the result enters the next outbound API call, any sensitive content it contained is already in the model provider's request body, in their logs, possibly in their abuse-review queue, and in any third-party observability tooling wired into the path. The egress has already happened on the read, not on the agent's downstream action. Post-execution controls therefore default to fail-closed, even though the latency budget is more forgiving. Redacting a fifty-megabyte log tail takes longer than classifying a single command, and the user generally tolerates the difference.

Inter-agent boundaries

When one agent passes a result to another, or when a sub-agent spawns and reports back, the boundary between agent A and agent B is where ASI07 (Insecure Inter-Agent Communication) and ASI08 (Cascading Failures) accumulate. Controls here look more like authentication and message validation than content filtering, and they are less mature in the current generation of frameworks. We discuss this in the Open Questions section toward the end of this document.

The asymmetry between pre-execution and post-execution failure modes matters. Output filtering and input filtering look superficially similar: both intercept content at a boundary, both apply some kind of policy decision. But their failure-mode requirements pull in opposite directions. A unified control plane that applies the same defaults to both is wrong by default. This is one of the design choices defenders frequently get wrong, and one that the next generation of agent security tooling will need to handle correctly.

Why Lexical and Static Approaches Fall Short

The instinct of a security engineer encountering agentic risk for the first time is reasonable: write a regex blocklist for the obvious dangerous commands. Block rm -rf, dd if=, chmod 777, the curl-pipe-bash patterns. Ship it. Done.

This approach fails for three reasons that matter, and a fourth that is more subtle.

The pattern surface is unbounded

Even restricting attention to shell commands that destroy data, the surface includes rm, find with -delete, dd, mkfs, shred, wipefs, truncate, scripted Python os.unlink, scripted Node fs.unlinkSync, database DROP TABLE, cloud aws s3 rm with --recursive, Kubernetes kubectl delete, and dozens more. Each has parameter variations that change behaviour. Writing the blocklist for one of these categories takes a week. Writing it for all of them takes a quarter. Maintaining it as new tools enter the agent's vocabulary takes a permanent staff role.

Encoding and indirection defeat regex

The simplest bypass is base64. The string "echo cm0gLXJmIC8K | base64 -d | sh" decodes to a recursive removal of the root filesystem, but the regex sees only the encoded form. Variable expansion, command substitution, aliases, and shell evaluation each defeat naïve string matching. Adding regexes for every encoding multiplies the maintenance cost without ever closing the gap.

Living-off-the-land attacks use legitimate tools

Searching for SSH keys with find / -name id_rsa -exec cat {} uses commands that any developer might run in any number of legitimate contexts. The surface form is innocent. The intent — exfiltration, persistence, credential harvest — is what matters, and intent does not appear on the command line.

False positives erode adoption

A safety system that blocks rm is workable on a developer's laptop only if it almost never blocks legitimate rm. The moment it does, the developer adds --dangerously-skip-permissions or its equivalent and never looks back. The safety property is gone. This is the central uninstall-risk argument that defines agentic security in practice. A noisy control that gets disabled provides zero protection. A quiet control that catches eighty percent of catastrophes provides eighty percent of the available protection.

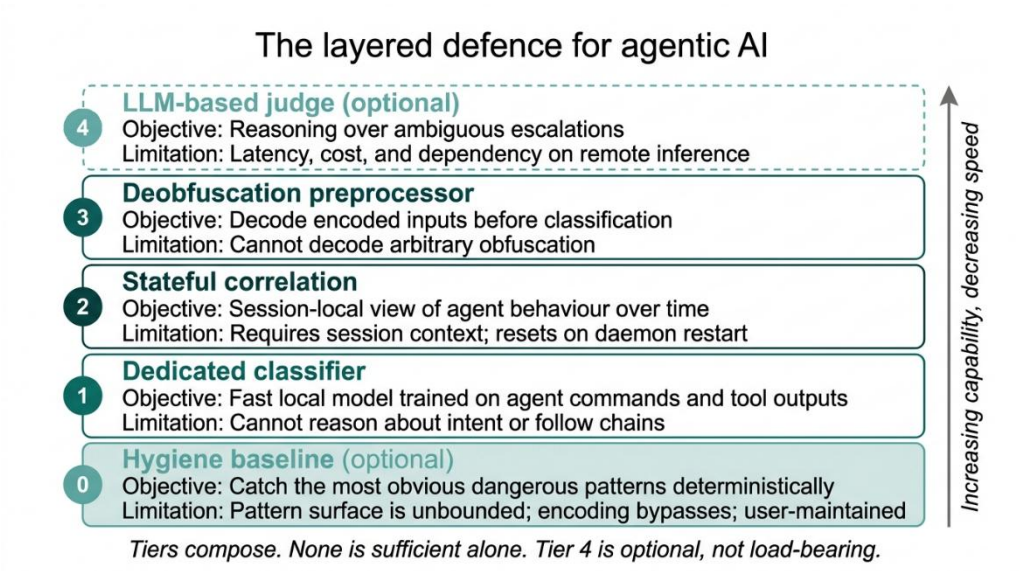
The implication for security architects is that the right baseline for output filtering is a purpose-built classifier trained on the surface of agent-generated commands, calibrated to the right precision-recall point, and capable of recognising semantic equivalents across encoding and obfuscation. This is not a theoretical aspiration. Small DistilBERT-class models trained on commands rather than general text run in single-digit milliseconds on commodity hardware and meet the latency budget for pre-execution attachment.

The same logic applies on the input filtering side, with one structural difference: secret detection is partially a deterministic pattern problem (AWS keys, GitHub tokens, JWTs all have predictable shapes) and partially a structural problem (config files, environment variable dumps, Kubernetes secret YAML). The first part can be handled by deterministic patterns with very high precision. The second part requires structural awareness — parsing the format, identifying sensitive fields by name regardless of value shape, and redacting accordingly. Mature redaction pipelines therefore combine high-precision regex for known credential formats with structured parsing for known sensitive file shapes, supplemented by probabilistic detection (entropy analysis, named-entity recognition) for the long tail.

Crucially, the regex layer in this combination is not the primary defence. It is a hygiene baseline that catches the highest-confidence patterns deterministically, leaving the harder cases to the classifier and the structural redactor. Selling regex blocklists alone as agent safety is selling something that does not work; combining them with semantic and structural controls is the architecture that does.

The Layered Defence

A single safety tier is insufficient regardless of how good the model behind it is. Three tiers, each with a different objective and different failure mode, compose into a defensible posture.



Tier 1: dedicated classifier

A purpose-built model trained specifically on agent-generated commands and tool outputs. Runs locally, in the millisecond range, with calibrated precision per risk category. Catches the broad case: known dangerous patterns, known sensitive content, known tool-misuse shapes. Designed for high recall on catastrophic outcomes and high precision on benign-looking commands. Limitations: cannot reason about intent, cannot decode arbitrary obfuscation, cannot follow multi-step chains.

Tier 2: stateful correlation

A lighter component that maintains a session-local view of what the agent has seen and done. Catches multi-step patterns the single-command classifier misses: read credentials in step one, attempt egress in step three. Also catches the post-read taint case. An agent that has just ingested a sensitive file should be evaluated against a tighter threshold for any subsequent action with egress potential. This is where ASI06 (Memory Poisoning) and ASI08 (Cascading Failures) get traction. Single-point classification cannot see the pattern, but session-level state can.

Tier 3: deobfuscation preprocessor

A component that runs before the classifier and transforms candidate commands into their semantically equivalent decoded forms. Decodes base64 substrings, expands shell variables (read-only, no side effects), resolves aliases, accounts for PATH-priority anomalies (a ~/bin/make that shadows the system make). The classifier then runs on the transformed input rather than the surface form. This is the honest answer to "the model can be fooled by encoding", which is to do the decoding before the model sees the input.

A separate question is whether the architecture should include an LLM-based judge as a fourth tier — a larger model that handles ambiguous cases the classifier escalates to. This has been argued both ways. The case in favour is that

LLM judges can reason about intent and follow multi-step chains in ways a small classifier cannot. The case against is threefold: latency (LLM judges typically run in seconds, not milliseconds), cost (per-call pricing accumulates rapidly under autonomous workloads), and dependency (the judge needs to be reachable, which contradicts the local-first principle for any organisation with strict data residency requirements).

The pragmatic position is that LLM escalation is a tool the deployer may wire up if their threat model and operational constraints justify it, but it is not load-bearing in the baseline architecture. The first three tiers — classifier, stateful correlation, deobfuscation preprocessor — handle the broad case. Organisations that want a fourth tier can integrate one of their choosing, retaining full control over which model serves it, where it runs, and what content it sees. This separation of concerns is important because the dedicated classifier and the LLM judge have fundamentally different objectives. The classifier is a fast, calibrated yes/no on a single observation. The judge is a reasoning step that may follow a chain of evidence. They complement each other; neither replaces the other.

The Local-First Imperative

Three independent constraints push agentic safety controls toward local execution.

Latency

Pre-execution checks block the agent. Even at hundred-millisecond latencies, the user experience degrades noticeably. At second-scale latencies, the agent becomes unusable for interactive workflows. Network round-trips to a cloud-hosted safety service are too slow for this attachment point, even when the network is fast. The classifier has to run on the same machine as the agent.

Privacy

Agent commands and tool outputs frequently contain the most sensitive content the agent ever sees: credentials, internal codepaths, confidential queries, customer data. Sending that content to a third-party safety service for evaluation creates exactly the data exposure the security control is supposed to prevent. The same content the customer would refuse to send to OpenAI for inference, they would refuse to send to a safety vendor for classification.

Availability

A safety control that requires network reachability to function is a safety control that breaks when the network breaks, when the safety vendor has an outage, when the customer is on a flight, or when corporate firewalls block egress. Each failure mode either disables protection (fail-open) or breaks the agent (fail-closed). A local-first architecture removes this dependency entirely.

These three constraints together imply a specific architectural shape. The safety control runs as a local daemon, exposes a loopback API for integration, and operates without network dependencies for its core function. Updates flow through the user's package manager, not a phone-home channel. Telemetry, when present at all, should be opt-in, narrow, and inspectable by the user before transmission, and should be entirely absent from any deployment claiming to support regulated or air-gapped environments.

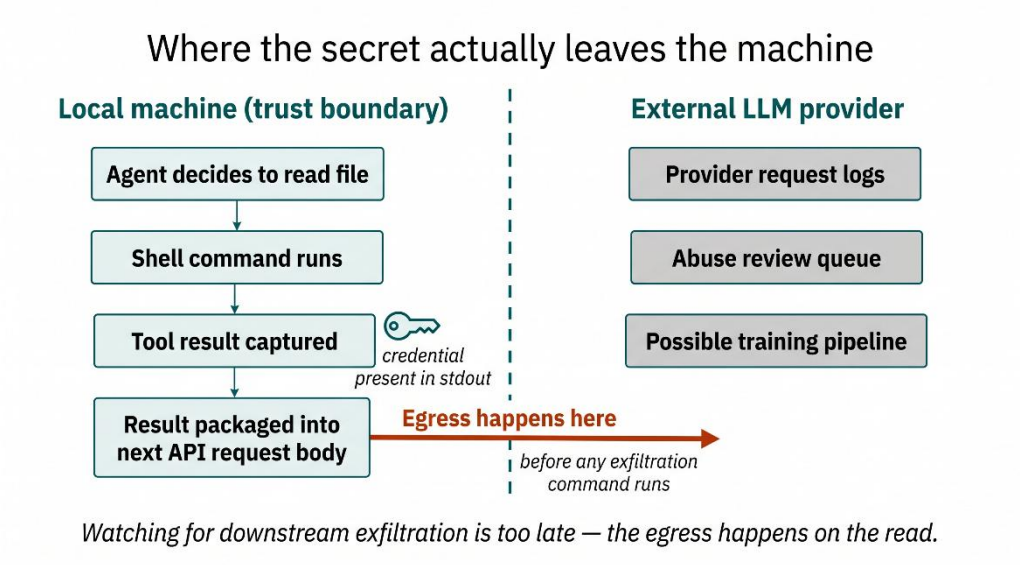
This architectural posture has commercial implications. The vendor offering local-first agent safety cannot operate the kind of cloud-hosted dashboard their SaaS competitors offer. Fleet visibility, audit log centralisation, threshold tuning at scale — these all have to be customer-hosted reference implementations rather than vendor-operated services. The vendor sells the model and the control plane; the customer runs the operations. For organisations buying agent safety tooling, the question to ask procurement is direct: does this control require network connectivity to your infrastructure to function, and what content crosses your boundary as part of normal operation? The answers reveal whether the vendor has internalised the constraints or just implemented them on a feature flag.

The Two Sides: Output and Input Filtering

Most existing conversations about agentic security focus on output filtering, which means preventing the agent from issuing dangerous commands. This is necessary but not sufficient. The under-discussed and equally important side is input filtering: controlling what content from tool results enters the agent's context and, critically, what crosses the network boundary into the model provider's API.

Consider a concrete case. An agent is asked to debug an authentication problem. It runs a command to read the AWS configuration file. The output, including the literal AWS secret access key, returns to the agent's runtime. The agent's harness packages that output as a tool result. The harness then constructs the next API call to the model provider — Anthropic, OpenAI, Google — including the tool result in the request body. At the moment that API call is sent, the credential has left the user's machine. It is now in the provider's logs, possibly in their abuse-review queue, and depending on the customer's settings, possibly in their training data pipeline.

The egress happened on the read. Watching for downstream exfiltration via a curl or scp command misses the actual exposure entirely. The agent did not need to attempt egress. The harness already did it as a routine part of operation.



Watching for downstream exfiltration is too late — the egress happens on the read.

The right control is post-execution redaction at the local boundary, before the result is packaged into the next outbound API call. This is the structural fix and it is achievable today through hooks that the supported integration patterns expose: Claude Code's PostToolUse hook, OpenClaw's emerging outputGuard interface, and Goose's ToolMonitor primitive with post-tool filtering. Each lets a local component inspect the tool output, redact sensitive content, and return a modified version that the harness substitutes into the request body.

What gets redacted, and how

The redaction itself has to be more sophisticated than a regex blocklist. High-confidence credential patterns (AWS keys, GitHub tokens, SSH private key blocks, JWTs) handle one part of the surface. Structural awareness of known sensitive file formats — .env files, kubeconfig YAML, Kubernetes Secret resources, AWS credential files — handles a second part by keying on field names rather than value shapes. Probabilistic detection (named-entity recognition for PII, entropy analysis for unknown high-value strings) handles the long tail. None of these is sufficient on its own; layered together they catch the broad case.

Equally important is what the agent receives in place of the redacted content. A simple delete-and-substitute approach loses information the agent legitimately needs: it now sees a malformed config file with random gaps. The right pattern is to substitute typed redaction tokens — a placeholder that signals "an AWS access key was here" rather than just empty space — and append a short footer to the tool output naming the redactions that occurred. The agent then knows it did not get the literal value and can either work without it or surface the gap to the user. This pattern preserves the agent's ability to reason about structure while denying it the specific values that should not leave the machine.

What input filtering cannot do

Input filtering at the integration-hook layer cannot intercept reads that happen inside non-shell processes the agent spawns. An agent that runs a Python one-liner to open a sensitive file and post its contents to an external URL performs the read and the exfiltration inside the Python interpreter. The shell never sees the file content. The harness never sees the file content. No hook fires. The only defence at this point is OS-level: making the file unreadable to the agent's process tree through Linux Landlock or AppArmor, or macOS Endpoint Security framework, or per-session sandboxing with explicit unmask grants.

This OS-level sandboxing is the frontier. It closes the gap that hook-based input filtering cannot close, but it requires significant per-platform engineering (Linux profiles differ across distributions, macOS has its own framework, Windows is yet again different) and per-organisation configuration (which paths to mask, which to allow, what unmask flow to provide for legitimate access). It is firmly an enterprise-grade capability today and will remain so for some time.

The honest summary of input filtering is that hook-based redaction handles the broad case and is the right baseline for any deployment, while OS-level sandboxing handles the in-process bypass and is the right answer for organisations whose threat model includes determined attackers operating through agent harnesses. Neither alone is sufficient for the highest-risk deployments; together they form the credible layered defence.

Open Questions and Ongoing Industry Work

Several aspects of agentic security remain genuinely unsettled in early 2026, and security leaders should expect the consensus to continue evolving. We highlight five.

The right attachment point for inter-agent communication

ASI07 and ASI08 collectively address risks that arise when agents talk to each other. Solutions range from message authentication (cryptographic signatures on agent-to-agent calls) to behavioural watchdogs (a separate agent that monitors peer actions for drift) to delegation depth limits (refusing to chain beyond N hops). None of these has converged into a default architecture across frameworks, and the multi-agent space is where the most novel security work will likely happen over the next eighteen months.

Memory governance in persistent agents

Hermes Agent and similar self-improving frameworks rewrite their own skill documents based on past failures. This creates a feedback loop where a single poisoned interaction can permanently alter agent behaviour, which is ASI06 in its sharpest form. The right response is unclear: aggressive memory rotation (lose the learning), human-in-the-loop review of memory updates (lose the autonomy), or memory provenance tracking with rollback (the most promising path, but operationally complex). Organisations adopting persistent-memory agents should treat memory as a privileged data store with the same controls applied to it that they apply to a configuration database.

Supply chain integrity for MCP and tool descriptors

ASI04 highlights that the components agents discover at runtime — MCP servers, tool descriptors, model endpoints — are themselves attack surfaces. The Model Context Protocol does not currently include strong authentication of tool providers; an agent connecting to a community MCP server has no built-in way to verify that the server's tool descriptions match what the server actually does. This is an industry-wide problem more than an individual-vendor problem, and we expect to see signed MCP tool manifests and a registry of verified providers emerge over the next year. The Cisco-discovered malicious skill incident on the OpenClaw skill registry is a current example of this attack class in production.

Coordination with provider-side safety systems

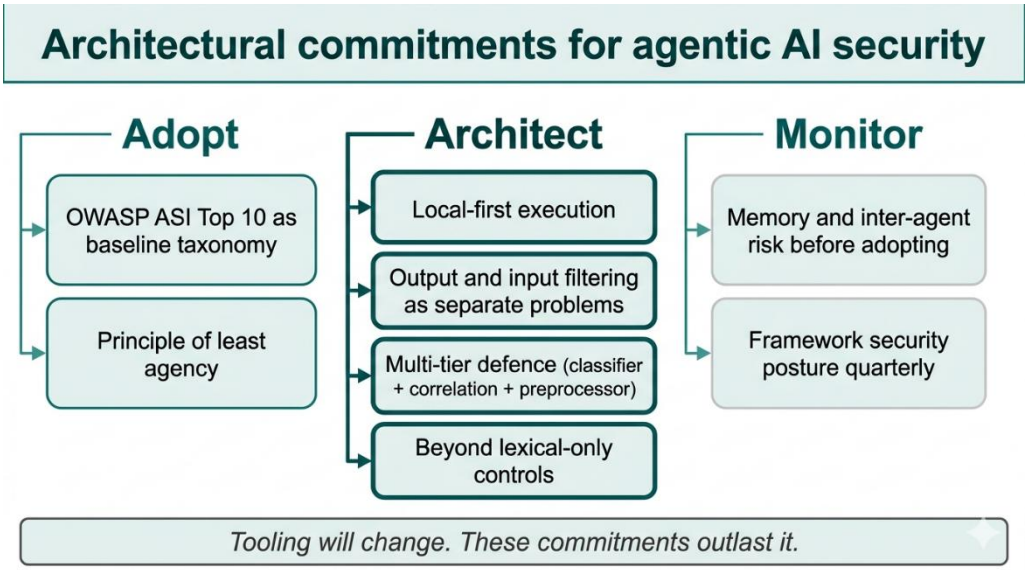
Anthropic, OpenAI, and Google all run safety filters on inbound API content and outbound responses. These are out of scope for the local agentic safety control to influence, but they sometimes interact in unexpected ways. A tool result that is benign in context can trip a provider's safety filter when sent as part of an agent prompt, breaking the workflow in ways the user cannot debug. Coordination between local agent safety and provider-level safety is an open conversation, and one where standards bodies like OWASP could play a constructive role.

Sandboxing as a default expectation

NemoClaw ships with default-deny network policy and four-layer isolation. Hermes Agent ships with Docker container isolation. Tank OS packages OpenClaw inside Podman containers. The trend is unambiguous: production-grade agentic frameworks will treat sandboxing as the baseline, not the premium feature. Goose, Claude Code, Claude Cowork, and

the broader developer-tool category will likely converge on similar defaults over the next year, either through native implementations or through wrapper distributions like Tank OS. CISOs should expect this baseline shift and plan their procurement and architectural reviews accordingly.

Recommendations for Security Leaders



The recommendations below are pitched at the architectural level rather than the product level. Specific tooling choices change quickly. The architectural commitments matter far longer.

- **Adopt the OWASP ASI Top 10 as the baseline taxonomy.** It is peer-reviewed, vendor-neutral, and industry-accepted. Internal risk registers, vendor security questionnaires, and agent procurement criteria should reference the ASI categories explicitly. This is the same role SOC-2 and ISO 27001 play for traditional infosec — a common language that makes cross-organisation conversations possible.
- **Apply the principle of least agency from the start.** New agent deployments should ship with the narrowest tool access that satisfies the use case, scoped credentials with short TTLs, and explicit denials for any action whose blast radius exceeds the value of automating it. This is harder than it sounds. Agentic frameworks default to permissive configurations because that maximises the perceived capability, and overriding the defaults requires deliberate organisational policy.
- **Design controls for the local-first constraint.** Any safety system that requires sending agent commands or tool outputs to a vendor for evaluation creates a data-residency problem that will surface during procurement at every meaningful enterprise. Build for, or buy for, local execution. The vendor that cannot operate without sending your content elsewhere will not pass your CISO's review.
- **Treat output and input filtering as separate problems.** They have different failure modes (open versus closed), different latency budgets, and different attachment points in the agent loop. A unified policy that applies the same defaults to both is wrong by default. Each integration point — pre-execution gating, post-execution redaction, sandboxing — has its own correct configuration.

- **Avoid lexical-only defences.** Regex blocklists for dangerous commands have a place as a tier-zero hygiene layer, and organisations may write their own as a baseline. They should not be the primary defence, and they should not be sold or marketed as such. The combination of a purpose-built classifier with structural redaction handles the broad case far better than any pattern list, regardless of how exhaustive the pattern list is.
- **Build for multi-tier defence.** A single safety component is insufficient. The minimum architecture combines a fast local classifier (output side), a structural redactor (input side), and increasingly, OS-level sandboxing for the in-process bypass case. LLM-based judges add value in narrow cases and at meaningful latency cost; they are not load-bearing in the baseline.
- **Plan for memory and inter-agent risk before adopting them.** Persistent-memory agents and multi-agent workflows are higher-risk deployments than stateless single-agent tools. The OWASP guidance treats them as advanced cases for good reason. Organisations should not adopt them without explicit memory governance and inter-agent authentication strategies in place.
- **Monitor the framework-security gap.** Most agentic frameworks ship with a security posture meaningfully weaker than the 2026 risk landscape requires. The gap is narrowing — Goose's ToolMonitor, OpenClaw's Tank OS distribution, NemoClaw's OpenShell, Cowork's Sentry — but it is not closed. CISOs should track which frameworks their organisation depends on and what each one has shipped or announced in the last six months.

Conclusion

Agentic AI security has reached the operational stage that retrieval-augmented generation reached in late 2023: real production deployments, real incidents, and a recognisable taxonomy of risk. The OWASP Top 10 for Agentic Applications gives security leaders a common reference point. The framework landscape is sorting itself into developer-tool, enterprise-platform, and human-in-the-loop categories with distinct security postures. The architectural choices defenders face are sharpening into a small number of consequential decisions.

The defining constraints of this generation of controls are local execution, layered defence, and honest acknowledgement of what each tier can and cannot do. A safety system that runs on the user's machine, classifies fast, redacts structurally, and degrades gracefully when components fail is the shape of the answer. The vendors and frameworks that build toward that shape will be the ones whose tools survive in production. The ones that do not will be replaced by the ones that do.

This document is published as part of the Aigos AI Security Blueprint Series and follows the open, work-in-progress framing of earlier blueprints in the series. It is intended as a starting point for conversations between CTOs, CISOs, security architects, and the AI engineering teams now working through these decisions for the first time. As the field continues to mature, we expect the architectural recommendations to evolve. The risk taxonomy itself, however, is unlikely to shrink.

V1.02 (29 Apr 2026)

Aigos — Securing AI Foundation

Securedbyaigos.com