kyvvu.

Building Secure Agents at Scale

A slightly weird white paper on agent security

AUTHOR

Prof. dr. Maurits Kaptein CTO, Kyvvu

BENEFITTED FROM FEEDBACK AND DISCUSSION WITH (AMONGST OTHERS)

Jeroen Janssen · Information Technology Program Manager, Belastingdienst
Andriy Podstavnychy · InfoSec Risk Director / Deputy CISO, Arcadis
Bart Jansen · Agentic AI Transformation Director, ING
Rob Guikers · Technical Innovation Manager, Rabobank
Niels Tailleur · CCO, Datashift
Koko Visser · Partner AI, Valcon


```
read    db.customers     · internal · may-hold-PII    ALLOW
send    mail.internal    · after internal read        ALLOW
read    web.page         · external · untrusted       ALLOW
send    mail.external    · after untrusted read       BLOCK
```

The same **send** is allowed after an internal read and blocked after an untrusted one — a decision no single-step check can make. Enforcement is on the **path**.

Executive summary

AI agents are language models (LLMs) that take actions. They can do so because, first, the text these models generate can be read as *instructions* — a step-by-step plan to be carried out — and second, it is relatively easy to write a small software application, called a *harness*, that faithfully executes that plan. By moving beyond generating text to taking real, digital actions, agents become both very powerful and potentially very dangerous.

After reading this white paper you will know what agents are, where they came from, and what they are used for. You will then have a vocabulary for the three ways in which agents can — and already do — cause harm:

- **The individual action.** Any single step an agent takes can be harmful: an agent might delete a file that should not be deleted, or wire a payment that should not have been wired. This, in many ways, is an old problem, and the controls we already use for ordinary software, such as isolation and RBAC / ABAC, still apply.
- **The content.** The text flowing into and out of the model can be harmful in itself — a secret leaking out in a message, or a malicious instruction hidden in a document the agent reads and then obeys. Per-step checks ("guardrails") are the natural defence for these threats.
- **The path.** A *sequence* of individually-permitted actions can do harm that no single step reveals: a record read here, a message sent there — each allowed on its own, the combination an illegal data exfiltration.

The third danger is the one agents genuinely add, due to their non-deterministic behaviour at run time — which they inherit simply from their use of LLMs. It is a new threat, and as a consequence one that existing controls miss.

In this white paper, we detail how agents are actually built — where the model, the harness, the tools, and the trust boundaries sit — because we think it is very hard to meaningfully reason about defence without a strong mental model of the underlying technology. Next, we survey the various approaches to securing agents (isolation, access control, prompts, gateways, guardrails, and runtime enforcement). We discuss what each of these does well and where the limits are. Finally, we argue that governance is not a separate project but something that mostly *follows* from securing agents properly.

To conclude, we argue that most of what makes agents dangerous is old and well-understood, and is handled by common security and governance approaches. However, the path — the order in which an agent strings actions together, decided at run time — is new, and it is where the new difficulty lies. To secure it, we need to properly understand where the new danger comes from (the LLM), and place the controls where that influence can't reach them — actively, inside the harness.

A note on why this white paper is a little weird

This is not a marketing leaflet. When writing, we set out to explain a novel field — what AI agents are, why they are and are not different from the software we already run, what actually makes them dangerous, and the full menu of approaches to securing them — as plainly as we can. Sure, we release this white paper mainly from the team at kyvvu.com, but it has been created in close collaboration with partners and users. And, our core aim is not to sell kyvvu; our core aim is to *standardise the jargon and foresee a future where 1000s of agents perform meaningful tasks in a way that is safe and controllable.* To emphasise this goal, we have walled any selling off: every claim about the Kyvvu product sits inside a clearly-marked box. Read the main text and skip every box, and you should come away with a better map of the field than you had before, having learned very little about Kyvvu.

That said, we will make one core argument about agent security throughout, one that we believe to be true, *and* supports Kyvvu's core product: AI agents take actions. Individually dangerous actions — deleting a table, sending data outside the building — are an old problem, and we have decades of controls to limit applications taking destructive or dangerous actions and we should use those same controls on AI Agents. What is new with agents however is their *path*: the *unplanned* sequence of individually-reasonable actions an agent strings together *at run time.* Almost everything that is new and hard about securing agents follows from that one novel property.

How to read this paper. This paper is written to be useful to several readers at once. If you have five minutes, the executive summary above should cover it. CISOs will get the most from sections 1–4 (what agents are, what makes them dangerous) and section 7 (governance). Architects will want sections 5–6 (the agent architecture and the security mechanisms in more depth). Executives can stay with the executive summary and the closing argument in section 8.

BOX **Why secure the harness, not the model?**

A fair question is why we do not simply make the model safe and be done with it. Two reasons. First, you usually do not control it: the model is often a third-party system behind an API, and a mature organisation runs several of them — so it is simply not yours to guarantee. Second, even a model you host yourself is non-deterministic and adversarially steerable — the same task can yield a different plan each time, and untrusted content can nudge it toward an unsafe one. Training models to follow instructions and avoid dangerous plans is worthwhile and makes bad behaviour rarer; but it shifts the distribution of what the model tends to do, it does not bound it. A guarantee has to come from the deterministic part that you own — the harness.

1. What is an agent?

In our jargon, an AI agent is *a piece of software that uses a large language model (LLM, from here on "model") to decide what to do, and a harness to actually do it.* The model "plans"; the harness executes; so-called tools give the harness things to execute on.

The move that turns a chatbot into an agent is small but consequential. A language model produces text from text — that is what we are used to by now (although one can debate whether we fully control and understand its consequences). But that text need not be prose for a human to read; it can equally be *instructions*: a structured plan describing actions to take. If we then write a small program that faithfully carries out those instructions — call a search API, send an email, run a script — and feeds the results back to the model for the next step, we have an agent. That small program is the **harness**, and it is in many ways the protagonist of this paper; it's the thing we want to build in a way that's both useful and secure.

However, the harness itself is really just ordinary software. It is deterministic, inspectable, and testable in exactly the way the rest of your software stack is; any guarantee you can make about an agent's behaviour is a property of the harness, not of the model. What makes the *agent* non-deterministic is the model: the same task, run twice, need not produce the same sequence of steps. (Strictly, even a model is deterministic — it runs on deterministic hardware, and with a fixed seed and settings you can theoretically reproduce a run. For every practical purpose, though, we should treat its output as non-deterministic and design accordingly.) The harness is predictable; the thing it asks for instructions is not; and so the final result — the behaviour of the agent — is not.

It helps to think about the scale of agents early. A single run of a coding agent such as Claude Code, now routinely used by software developers around the globe, often consists of tens of model calls and hundreds, sometimes thousands, of tool calls: each file read, each edit, each test execution is an individual step. Multiply that by the number of agents an organisation will run, and the surface we are securing comes somewhat dauntingly into focus...

2. What are agents used for?

Agents have moved, quickly, from simple demonstrations to real multi-step work: writing and shipping code, doing research, handling customer service, running back-office operations, moving data between systems. The trajectory described in numerous books (such as *AI Agents at Work* [2]) is from assistant to colleague: from a system that suggests text to one that takes actions. That shift is exactly what makes agents both powerful and dangerous: a wrong suggestion is an inconvenience; a wrong action is an incident.

There are two metrics important to understand the scale of our potential problems:

1. A single agent task can be hundreds if not thousands of steps.
2. A single company will, before long, run *thousands* of agents — internal ones that summarise meetings or reconcile invoices, customer-facing ones embedded in products, and a long tail of third-party agents staff adopt on their own.

The boundary between "one agent" and "many" is admittedly blurry — a harness that spawns sub-agents is hard to count cleanly — but the order of magnitude of the total number of actions your agent fleet will take is not, and that sort of scale is what we should plan for when building our agents.

We would like to note that agent adoption seems to be running well ahead of agent control. Industry surveys through 2026 repeatedly find near-universal AI adoption alongside a small minority of organisations that have actually inventoried or audited what their agents do: the Cloud Security Alliance [3] found 82% of enterprises have unknown agents in their environment. A related survey found that while around half of organisations had at least partially documented agent-governance policies, only 31% had formally adopted one [4]. We have some catching up to do.

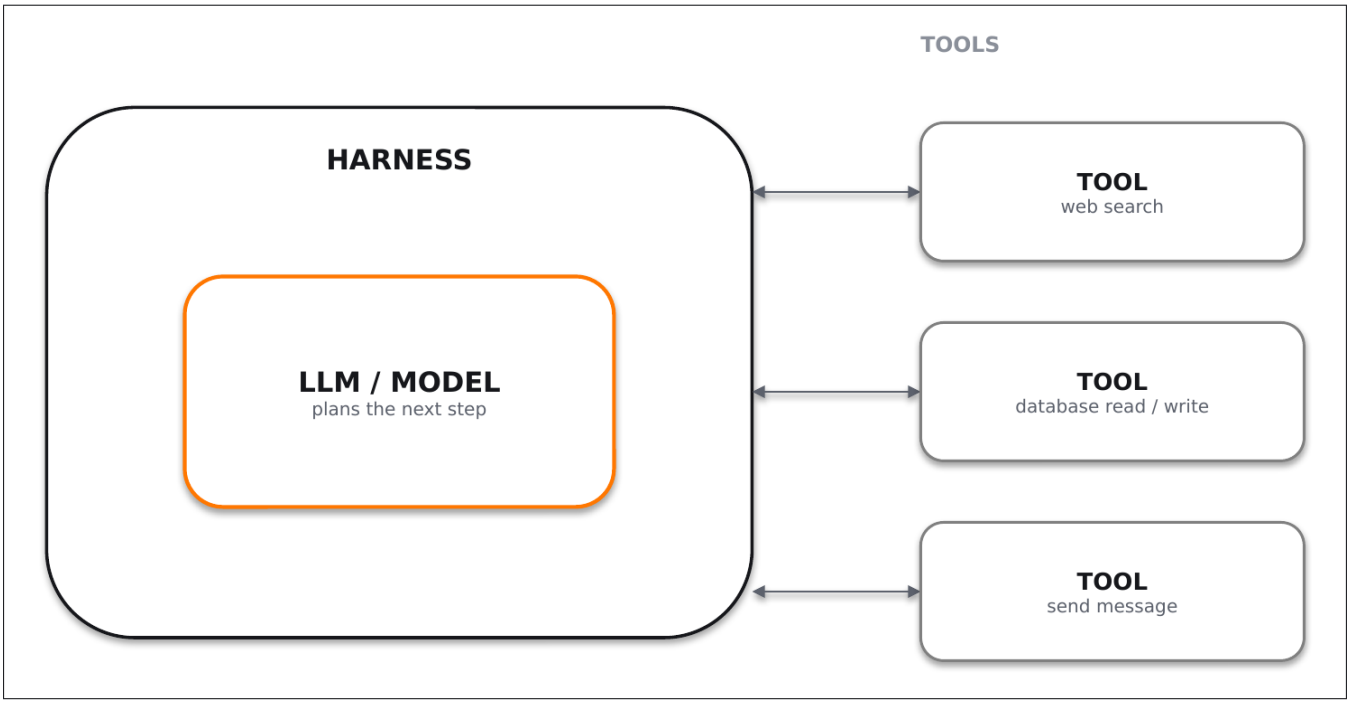

Figure 1. An agent: a foundation model proposing a plan, a harness faithfully executing it, and tools through which the harness acts on the digital world.

BOX Famous (or notorious) agents in the wild

- **Claude Code** — a developer agent that reads, writes, and runs code, often inside CI/CD. A normal session is tens of model calls and hundreds-to-thousands of tool calls.
- **OpenClaw** — the consumer/prosumer agent that went viral in early 2026; later a recurring cautionary tale once its skill marketplace filled with malicious "skills" (see Appendix A).
- **Enterprise deployments** — agents are in production across financial services, insurance, and health-care back-office work, where the actions are real (refunds, record updates, payments) and so is the exposure.

3. How are agents different from ordinary software?

Mostly, they are not. So, it's useful to first and foremost approach agent development and control the way you would treat any other software application. An agent can often be thought of as a **microservice**: a small, self-contained unit with a defined purpose, a lifecycle (it is developed, tested, deployed, retired), an identity, and a set of other services it talks to (the tools). The agent likely has an API (defining starting points / triggers), permissions, logs, and an owner. Almost all of your existing engineering and security practice still applies, and you should keep applying it. In a way, please resist the urge to re-invent the wheel for anything agentic just because the hype is on the side of those pitching novel agentic tools.

In our view, there is exactly one difference that truly matters between current applications and AI agents. A conventional microservice has a control flow fixed at design time: in principle — though it may be challenging in practice — you can read the code and enumerate the paths it can take, so much of the control of conventional software can happen at design time. With agents this breaks: an AI agent decides its path at run time by asking a non-deterministic model what to do next, and that chooser is both unpredictable and adversarially steerable. You cannot enumerate the paths in advance, because the model can compose them in ways no one wrote down. This is both a blessing and a curse: it's a blessing because now developers do not have to explicitly create a suitable path for every task the service should solve a priori. It's a curse because the AI agent might be (extremely) creative in choosing its own path to achieve a task (which is compounded by the fact that the task itself is often user-specified and very broad). Worse still, that path need not be the agent's own idea at all: because the model is steerable by the content it reads, an adversary — a poisoned document, a tool result, an inbound message — can actively influence the path it chooses.

BOX Path-level incidents are coming

Public incidents where an agent caused harm through the *ordering* of permitted actions are piecemeal starting to emerge. Of course, they are new — and agents are new — so this is just the beginning: it is exactly what you would expect of the newest and least-recognised danger. However, we are finding more and more instances (full catalogue in Appendix A):

- **Sysdig — the first in-the-wild agent-run intrusion (May 2026).** An attacker-operated agent took a single known vulnerability and, in four moves in under an hour, mapped the network, widened its own access, reached an internal database, and copied the data out. Each move was the kind of thing a legitimate operator might do; the breach was the trajectory, not any one step [8].
- **PipeLeak / ShareLeak — exfiltration through an agent's own authorised actions (2026).** In Salesforce Agentforce and Microsoft Copilot Studio, an agent was steered into reading customer / SharePoint data and then sending it out through its *permitted* email tool. Tellingly, patching the original injection vector did not stop it — the data still left through the agent's authorised read-then-send sequence, which is exactly the part no per-step check forbids [25].
- **Vertex AI "Double Agent" (Unit 42, 2026).** Researchers showed that an over-permissioned Google Vertex AI agent could use its default credentials to pivot out of its own task context into the owner's cloud project and read every storage bucket — harm assembled by chaining individually-granted permissions, not by any single illegal call [26].
- **Meta internal agent (a top-severity "SEV1" incident, March 2026).** The agent took an autonomous action that should have required human approval — posting guidance a colleague then acted on — broadening access to sensitive internal data for about two hours. No attacker; the failure was an autonomous step that should have passed an approval gate, and didn't [9].

So, the way we look at it, agents are largely not a different kind of computing — the same processes, identities, and APIs as before. However, they are novel in the sense that they are systems whose order of operations is chosen, step by step, by something you cannot predict at design time. The correct response is not alarm; it is to stop relying only on design-time assurances (which are now almost trivially incomplete) and to add a way to constrain the path — the sequence of steps — while it happens.

4. What makes an agent dangerous?

By mid-2026, one assessment of production agents [1] found that already roughly 98% of agents are sufficiently complex to exhibit what Simon Willison termed the *lethal trifecta*: access to private data, exposure to untrusted content, and a way to send data out. Those are the three ingredients that turn a single injected instruction into data exfiltration. In other words, most deployed agents are not one *bad action* away from harm — every individual action may be fine — but they are one *bad sequence* away from it.

To understand agent risks broadly, it helps to conceptually separate three distinct ways an agent can cause harm:

One: the individual action. A single step can be destructive on its own — drop a production table, erroneously wire a payment, overwrite a file. This danger is an old one. We have mature controls for it: permissions, least privilege, approvals, input validation, isolation. Agents inherit these issues and thus the same controls should be applied.

Two: the content. The text flowing into and out of the model can itself be the source of harm — secrets in an outbound message, or, more insidiously, a malicious instruction inside the content the agent reads. The latter is the prompt-injection family, and it's tricky since the model cannot reliably tell instructions apart from data: to the model, while creating the plan, both are simply tokens. A web page, an email, a tool's output, or a code comment can carry instructions the agent subsequently follows [6]. Hence, content checks — input and output filters — are a natural control here; these might be humans looking at the messages, another AI agent inspecting the messages — which clearly inherits some of the issues — or a deterministic process checking the input and output. We would recommend you have a combination of these controls, each with known performance metrics (error rates).

Three: the path. This is the danger that agents actually truly add. A *sequence* of individually-permitted actions, on individually-innocuous content, can constitute harm that no single step reveals. An agent reads a customer record it is allowed to read, then sends a message it is allowed to send — and the combination is a data exfiltration that neither permission, on its own, forbids.

The reason the third danger is hard is that every

BOX The harness-separation assumption

Every control in this paper rests on one assumption: the harness — including any control layer built into it — cannot be modified by the agent at run time. This is not special to agents; it is a precondition for control of any kind. A controller that the thing it controls can rewrite is not a control — the object under control would simply change the rules meant to bind it. It is why an operating system keeps its kernel out of reach of userspace, and it is the tamperproofness any reference monitor requires [10]. This has one concrete consequence: any tool that runs attacker-influenceable code — a shell, an exec, a notebook cell — must execute in an isolated process whose only routes back to the harness are themselves mediated actions. Run such a tool inside the trusted process and injected code can rewrite the control layer itself, collapsing the boundary. By the same logic, a control that sits on the message wire — inspecting the model's traffic from outside the harness — is effectively on the untrusted side of this boundary: it can watch what passes, but it has no tamperproof footing from which to stop it (even worse, it has no direct access to the harness's code, and hence can only guess the potential effects of the messages on the wire). Enforcement has to be a mediated action inside the trusted process. One more assumption rides alongside: the harness's typed account of an action must faithfully match what the action does — a step declared "send a message" must not, underneath, open a raw socket. This is the in-house counterpart of the wire-inspector's parse-versus-execute gap, but pulled inside a boundary you control: deterministic code you instrument and can verify — an invariant you maintain, not one you get for free.

stateless control misses it by construction. Identity sees one action and one resource. A content filter sees one message at a time. None of those approaches hold the task's history, so none of them can evaluate a step *in light of what the agent has already done*. This type of conditional approval is normally something prevented at design time; however, as we have already noted, the whole point of these agents is that their behavioural path — the sequence of steps they take to carry out a task — is determined at runtime.

These three dangers map cleanly onto the security mechanisms we survey next: action-level controls (isolation, access control) for the first, content checks (now often called "guardrails", although that term has been subject to some inflation) for the second, and a path-aware runtime intervention layer — an *agent security kernel* — for the third.

5. How are agents built? The architecture

You cannot reason about danger or defence without a shared picture of the parts and the boundaries between them. In this section we try to draw that picture in a way that is fitting for most agent deployments we have seen over the last year. The most important concepts are laid out in Figure 2:

The harness and the model. The model *proposes* — a plan, a next action — and the harness *executes* — it actually makes the calls. The model is the non-deterministic part; the harness is ordinary software you own and can instrument. This split is very important: since your controls should be on the side that you actually control, and in most agents only the harness is on "your" side, you should think about controlling the harness. The model, and the content it ingests, should be considered **untrusted**; the harness code — including any control layer integrated into it — is the **trusted computing base**.

Model calls and tool calls. The harness conceptually takes two kinds of step, and they carry different risks:

- A **model call** is reasoning: at its core it is text to text (even if that text contains harness instructions) and it has, in isolation, little effect on the real world. However, since models are often external, the text to the model does run a data exfiltration risk that should be guarded.
- A **tool call** is a direct effect: a file read, a database write, an email, a shell command. The model call is where the path is *chosen*; the tool call is where it *executes*. Please note that a tool call in agent-practice is often much more than a simple "read this file"; it can extend to "run arbitrary code" — so scoping what each tool is allowed to do is key.

Where do things live? What the agent touches divides into trusted, internal resources — your databases, knowledge bases, retrieval (RAG) stores, internal tools — and untrusted, external ones — the open web, third-party APIs, inbound messages and documents. Untrusted input is where injection risks enter (i.e., an LLM being prompted to recommend illicit actions to the harness); ungated output is where data exfiltration events happen. Hence, the internal/external boundary is a clear security boundary.

The model call is itself a boundary. As stated, in many deployments the model very often sits

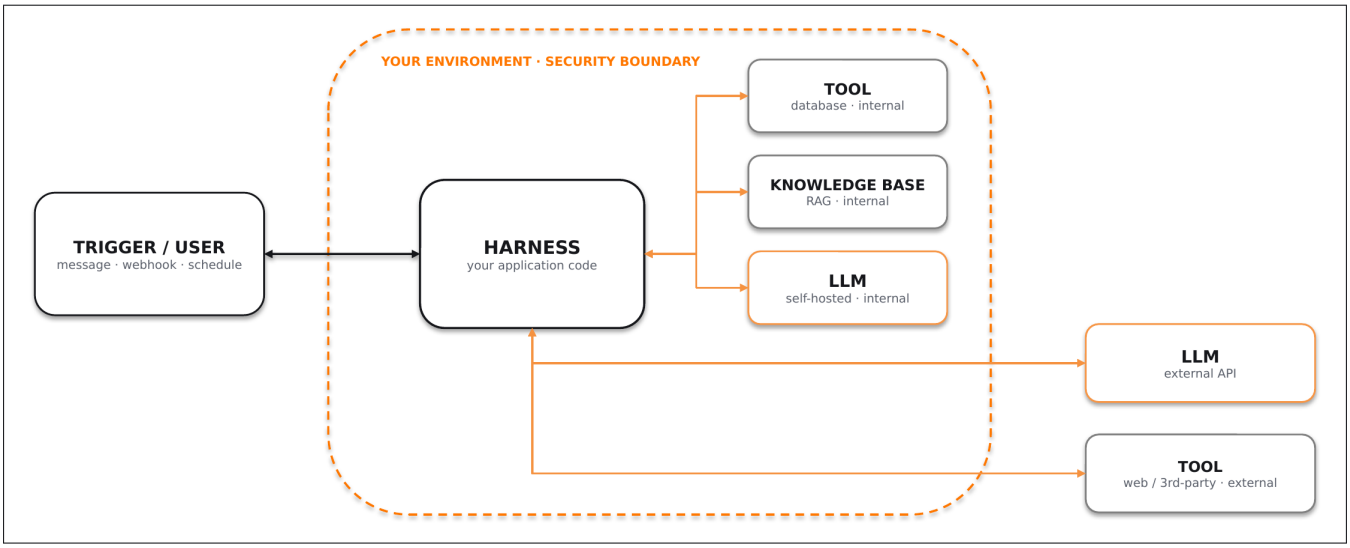

Figure 2. Agent architecture: the harness at the centre, clearly separated from the models (internal and external) it calls. Next, we have both internal and external tools and we explicitly mark the trigger. Understanding which parts are under whose control is the step-up to understanding how to secure an agent.

outside your walls (or likely, a single agent will use multiple models, some of which are inside and some of which are outside of your walls). Outside models provide opportunity for both data exfiltration and prompt injection. Self-hosting a model these days however often trades capability for keeping sensitive data in-house. Hence, in Figure 2, we list models both inside and outside of your boundary.

Tools, usually over MCP. A mature agent is not a lone model; it has capabilities — knowledge bases, retrieval, execution and action tools — increasingly exposed through the **Model Context Protocol (MCP)**, the emerging standard for connecting agents to tools and resources (both internally and externally). Tools are features in the sense that they make it possible for the agent to act. However, tools are also a possible attack surface: a tool's description or output can carry injected instructions (tool poisoning), and MCP servers have been hijacked to redirect agents (the "agentjacking" technique).

Triggers. The thing that *starts* an agent task: a user message, a webhook, a scheduled job, a file landing in a folder. The trigger is part of the threat surface, as much as the tools are. An attacker who controls a trigger controls an entry point.

There is one architectural conclusion to draw before we discuss agent security in more depth: A mature organisation will not route all of its intelligence through a single, remote, general-purpose model. It will use several models — some hosted, some self-hosted, some embedded, chosen per task for cost, capability, latency, or data sensitivity. Once that is true, any security control that (pretends to) work by sitting on the model-call wire can only see the calls that happen to route through it, and only by placing itself squarely in your data path. Being in the harness-model wire is the right architecture for cost control and rate-limiting for *external* models but it is the wrong place to enforce security for your fleet of agents.

6. How do we secure agents?

"Secure" is a slippery term, and the risk is never zero — the most secure agent is the one we did not build. As soon as an agent does useful, loosely-structured work with real tool access, some risk is irreducible. The goal is therefore never perfection but three achievable things:

1. limit the risk,
2. be able to intervene when something goes wrong (at agentic speeds!),
3. and always be able to show what controls were in place at the moment an action ran.

In our view, securing an agent means doing two things simultaneously. First, we should keep the mature, action-level controls we already have for individually-dangerous actions — and second, we should *add* something that can reason about the path. The first is well-understood; the second is the new agent security requirement.

6.1 The emerging approaches to agent security

We see a number of approaches to agent control and security emerge (although the approaches to agent governance are even more plentiful, but those simply observe and don't enforce; we are writing this piece to ensure agents are safe, not to ensure you know after the fact that they weren't). We take each approach in turn: what it sees, what it

enforces, where it sits on the architecture, what it solves well, and where it stops. Please note that of the five approaches below, four are ones you very likely already run; three of them — isolation, IAM, and guardrails — cover the first two dangers we identified earlier, while the AI gateway is a routing concern, not a security control. The fifth — path-aware runtime enforcement — is the one agents newly require, and the one almost no stack has yet.

6.1.1 Isolation and sandboxing. The most foundational control: confine the agent — and especially its execution and tool calls — to a bounded environment, with restricted file systems, limited network egress, and containerisation, so that the blast radius of any single action is capped. This approach effectively limits the number of outgoing arrows from the harness to the various tools. Isolation solves individually destructive actions by simply restricting such actions. However, isolation bounds *where* the agent can act, not *which sequence* of permitted actions it takes inside the box — and the boundary itself is attackable. Several recent agent incidents are precisely sandbox or allowlist escapes.

6.1.2 Identity and access control (IAM / RBAC / ABAC). Give the agent an identity and scope its permissions to what its task needs, ideally derived from the invoking user and issued as short-lived credentials. It sits at every resource boundary in Figure 2 and is the right foundation for identity and least privilege [10]. Increasingly, coarse role-based access (RBAC) is regarded as too broad for agents, and attribute-based access control (ABAC) — decisions conditioned on request attributes at run time — is the better fit for real-time conditional access; note, though, that even ABAC evaluates a single access decision in isolation, and still does not see the path. This is clearly a more refined way of sandboxing; it's conditional sandboxing. Hence it inherits the pros and cons of sandboxing while providing the agent with more opportunities. You will need this, and with isolation, this should cover the first danger — individual actions — we identified earlier.

6.1.3 AI gateways (and / or "LLM routers"). AI gateways solve a real, separate problem — cost control, rate-limiting, and routing across (external) models — and need no view into the traffic's content to do it. We recommend running one, internally hosted, for exactly that, plus the security-adjacent work it does well: proxying credentials, holding secrets, issuing scope-limited tokens. (The term does not yet seem to be standardised; "LLM router" and "AI proxy" cover overlapping things.) The "gateway" is not, however, a place to enforce security. If it inspects the content of traffic it is effectively a guardrail in another location (see below), and should be judged as one. If it goes further and inspects the model's plan, it still sees only the proposed path — what the model says it will do next, authored by the untrusted model itself, the adversary — never the actions the harness actually runs; and it sees only the traffic routed through it, so tool calls and local or embedded models slip past. Not sharing the harness's code, a control that sits on the line between model and harness can only guess how output truly becomes action, so the model-proposed path is effectively parsed twice — at the gateway and at the harness — and any gap between the two is what a model can be steered to exploit. If you ask us, "sniffing the wire" is not agent security: the moment a gateway reaches into the model call to enforce, it inserts itself into the critical data path and takes on its own exfiltration and prompt-injection risk. And its correctness rests on a harness it never sees: point the model's base URL elsewhere — a one-line change, no compromise required, in a misconfigured or simply multi-model harness — and the gateway is bypassed silently, with no deny and no log, because it was never in a position to observe the harness at all. A gateway, when used as a control, trusts exactly what it cannot verify (the harness) and inspects exactly what it cannot trust (the model's plan). Enforcement belongs where the actual actions are — inside the harness.

6.1.4 Guardrails (I/O checks and gates). Per-step checks on the input to, or output from, the model (or to and from tools, or to and from users): content filters, PII scanners, regex and format validators, classifier-based gates, and human-in-the-loop approval gates. The natural analogy is a reviewer stationed at a single step. They come in three flavours worth distinguishing: *deterministic* checks (regex, schema validation) are narrow but neatly auditable; *model-based* checks (an LLM judging an output) are far more flexible but probabilistic, and should be trusted only with published benchmarks behind them; *human gates* are high-trust and slow. Guardrails, when done well, solve the second danger; they prevent harmful content, one message at a time. However, a guardrail is stateless by construction — it sees one input or output and has no memory of the task — so it cannot reason over the trajectory.

6.1.5 The path-aware runtime layer — an Agent Security Kernel. Everything above is stateless over the path; this is the mechanism that is not, and the one the previous four cannot replace. A small, trusted layer that mediates every action the harness takes before it executes, and evaluates that action against the full ordered history of the current task. Mediation should be complete over the harness's *action interface* (i.e., it should catch every action). In agents, interception can be done using a deterministic hook between "harness emits an action" and "action actually runs" and hence the

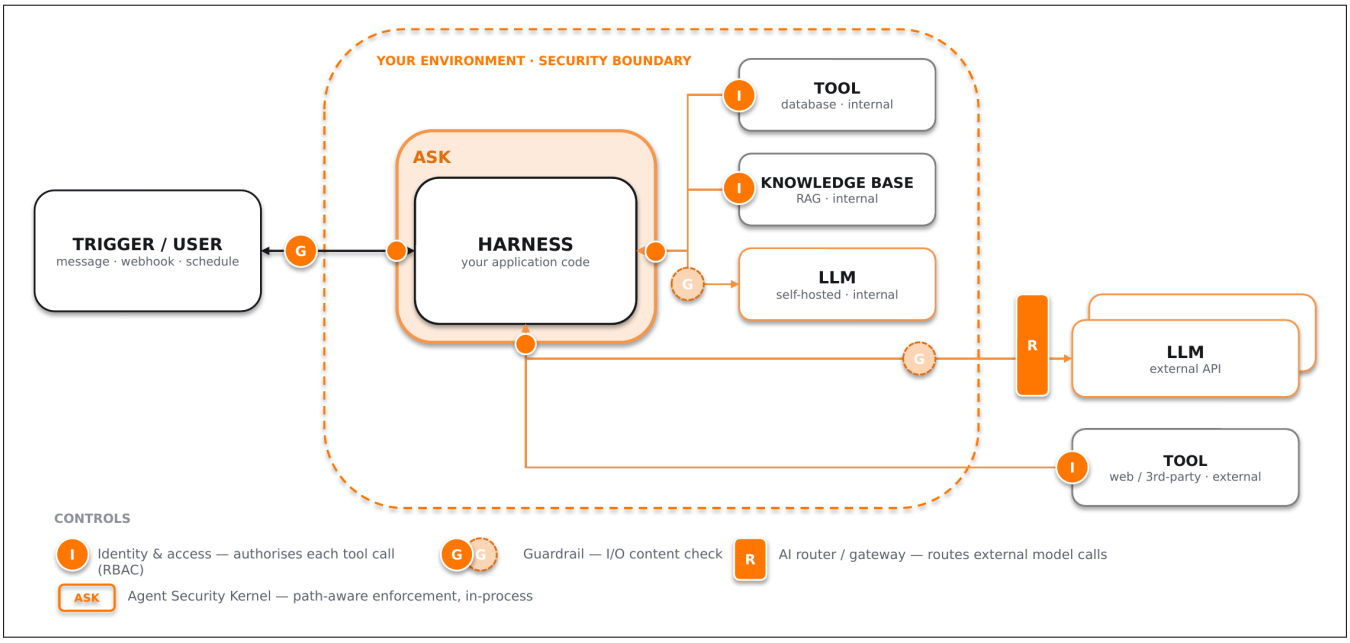

Figure 3. Where the controls sit: effectively Figure 2 again, but with the controls superimposed — Identity (I) where each tool call is made, Guardrails (G) on the I/O paths, AI router (R) on the external-model call, and the Agent Security Kernel (ASK) wrapped around the harness. (Sandboxing omitted for clarity.)

model itself *cannot decline to be mediated*. (One residual limit follows from this boundary: the kernel mediates the harness's *action interface*. If the model instead persuades a *human* to act — using the very persuasive capability that makes LLMs powerful — that action originates outside the mediated interface, and hence outside policy scope; oversight there is a training-and-process control, not something the kernel can enforce.) The runtime layer checks: given all the actions done before, is this action permitted? This is the approach that is stateful over the path, and hence it is the only one that addresses the third danger directly: it can block an action because of the sequence of actions before it. The academic name for this shape of control — which is not itself new — is a **reference monitor** — a small, trusted core that mediates every operation a process attempts [10] — the same primitive that underpins mandatory access control in mature operating systems, here applied for the first time to autonomous AI agents.

In Figure 3 we position each approach in the general agent architecture. We position the runtime layer fully around the harness: this layer does not inspect the message contents, but it sees every type of step the agent takes with its associated properties.

We want to close this section with a short mention of "prompting to secure agents" (i.e., tell the agent "thou shalt not do this or that"). That's great. You should do it. But it is not a control. Prompting hopefully makes the non-deterministic model behave more sensibly more often; it does not guarantee, at all, that it behaves sensibly all the time.

6.2 The properties you actually want

Before making any choices regarding tools to secure your agents, it is worth writing down — vendor-independently — what properties a **path-aware security layer** — i.e., the layer for the third and novel danger — *should* have. We think it's the following:

- **Fast** — It does not slow down your agent. Hence, it runs sub-millisecond.
- **On your side of the fence** — The solution runs in your environment; sensitive inputs and outputs never leave your perimeter to be evaluated. Your agent security approach does not double as a data exfiltration or prompt injection risk.
- **Topology-invariant** — It works regardless of which model, which framework, or where the model runs (cloud, local, embedded, multi-model).
- **Customisable** — Each policy expressed as code that you own and version.
- **Inspectable / open** — Any decision is a rule an auditor can read, not a black box. Ideally the tool itself is source-available.
- **Preventive, not observational** — enforces *before* the action runs, rather than logging it after the fact.

6.3 A comparison of control mechanisms

The table below places every approach on the properties we outlined. The agent security kernel is just one row among several; we think it should be added to your existing stack of controls.

We think the best read of this table is that no single row is "the answer to the ultimate question of life,

KYVVU How our Agent Security Kernel was designed

We built Kyvvu to tick exactly the desiderata listed above. We started from the research question rather than the product. The core results are published as *Runtime Governance for AI Agents: Policies on Paths* [11], which formalises the execution path as the object of governance and shows that prompt-level instructions and static access control are both *special cases* — the former shapes the distribution over paths without evaluating them, the latter evaluates a policy that ignores the path.

An **Agent Security Kernel (ASK)** is a reference monitor for agent actions: a small, deterministic core that runs *in the agent's own process* and mediates every action — model call, resource read, tool execution, message, credential fetch — returning one of exactly three decisions, **allow / warn / block**, before the action runs. It is a security kernel, not a proxy or gateway: it receives a *decision request* — the action and its properties — rather than sitting in the data plane the payload transits on its way to a destination. That distinction is what makes it topology-invariant (it does not care how many models you use or where they run) and keeps your data in your environment. It gives deterministic, complete mediation of every harness action, evaluated in your environment. Kyvvu currently ships in two forms — an in-process library (a Python import) and a sidecar (`kyvvu serve`) for other runtimes. Decisions are deterministic and sub-millisecond (296µs at p99 with 100 policies and a 50-step history; see docs). Policy is authored as version-controlled YAML in your own git, and you can roll out in observe mode (`warn`) before switching to enforce (`block`), agent by agent.

Product details defer to docs.kyvvu.com.

TABLE — COMPARISON OF AGENT-SECURITY MECHANISMS

Mechanism	What it sees	What it enforces	Stateful over the path?	Runs in-process / data stays?	Preventive vs. observational
Prompting / model safety	The request & response	Shapes behaviour (advisory)	No	n/a	Neither (not a control)
Isolation / sandboxing	The execution environment	Caps blast radius	No	Yes	Preventive
Identity & access (IAM / RBAC)	Identity + resource	Access by identity/role	No	Yes	Preventive (per tool call)
Guardrails (I/O content)	One input or output	Flag / redact / block a message	No	Depends	Preventive (per message)
AI gateway / LLM router	Proposed model traffic only	Routing + optional content policy	No (proposed calls, not actual actions)	Often no (SaaS)	Mixed
Path-aware runtime (ASK)	The full task path	Policy over the path (allow/warn/block)	Yes	Yes	Preventive (on path)

the universe, and everything." (which we all know to be 42). The stateless rows and the path-aware row are simply complementary — isolation and IAM cover the first danger, guardrails the second, the runtime layer the third. However, currently most mature production stacks will already have every row *except* the last.

7. How do we govern agents?

Governance is not a second project you bolt on after security; it is what falls out of having secured the agent properly. If you can mediate and record every action, a defensible account of what each agent did is a by-product. We do not believe "agent governance" should require a new platform. An agent is software; it should flow into the registration, identity, logging, and incident systems you already run, not into a parallel "agent governance" stack bought for the purpose. This applies to identity in particular: adapt the IAM and identity-governance systems you already run to also cover agents, rather than buying a net-new "agent identity" product because the category is fashionable. The job, as with every new tool, is integration, not reinvention.

7.1 Observability and enforcement are not the same thing

Two capabilities get blurred together under "governance," and keeping them apart is useful: *Observability* means you can see what the agent did. *Enforcement* means you can stop what the agent is about to do. These are different in kind, and in our view they stand in a strict order: enforcement implies observability — to enforce on a step you must first capture it and can therefore record it. However, observability does not imply enforcement — the ability to see an agent make a mess does not mean you can prevent it.

This is why we are wary of "governance" offerings that are observability with a compliance label. Observability built *on top of* enforcement is real and valuable: the record is trustworthy precisely because the same mechanism that wrote it also had the power to stop the action. Observability *without* enforcement records things it could never prevent.

Build the enforcement properly and the observability comes with it: you simply store the trace of enforced actions.

At this point it is useful to revisit the idea that an agent security kernel sees the path of an agent (i.e., the order of named and typed steps), and not the actual content of the prompts, database reads, etc. This obviously determines what the trace of such a system can be used for. The trace gives you *integrity*: because the same mechanism that recorded a step could have blocked it, it is trustworthy as to what happened and in what order. And because each step carries the classification declared for it at design time — "this resource may hold personal data", "that channel is external and untrusted" — the trace shows, with certainty, that a risk pattern occurred: a read from a resource that may hold protected data, followed by a send on an external channel. What it deliberately does not assert is that a leak did happen; that would require tying the specific content read to the specific content sent, and content is exactly what the kernel does not look at. This is a design choice, not a gap. The typing comes from data-source labels declared out of band — existing Purview-style classifications, for instance — so the trace gives complete, certain detection of the risk pattern: it flags every case where the pattern was present, with no false negatives on the pattern (false positives on actual leaks are expected — sorting those is the guardrail's job). Deciding whether a flagged case was an actual leak is a content question — inherently probabilistic, with errors both ways; it's the job of guardrails. And for agents that content question is also inherently adversarial, since the content is produced by the very model you do not trust. This is why, in theory, content is the boundary a security kernel must not cross: a reference monitor earns its guarantees by completely mediating a well-defined interface, and the agent's typed actions are exactly such an interface — whereas content is the adversary's unbounded output, which no monitor can mediate with the same completeness or certainty. Staying out of content is therefore not a practical convenience but the very thing that lets the kernel be a reference monitor; the determinism, the small verifiable core, and keeping your data in your environment all follow from that same boundary.

7.2 Existing regulation already reaches agents

A practical point that surprises people: you do not need agent-specific laws to be obligated here. Several existing frameworks already apply to software that uses models, and an agent is just one such software (we would argue that any software that uses a model *to take actions* should be secured as an agent; a model that only classifies or summarises, with no action surface, is not an agent by our definition but still inherits the content and data-handling obligations below). Thus, for agents, the following apply (amongst many others):

- **The EU AI Act** [12] (Regulation (EU) 2024/1689) regulates AI systems by risk tier and, for high-risk uses, requires risk management, record-keeping, human oversight, and accuracy/robustness (Arts. 9, 12, 14, 15). It is the cleanest lens for thinking about agent governance, which is why we use it below.
- **The GDPR** [13] (Regulation (EU) 2016/679) applies whenever an agent processes personal data — including when prompts or tool results carry it to an external model, which is a transfer to consider.
- **The Medical Device Regulation** [14] (Regulation (EU) 2017/745) applies when an agent's purpose is medical; an agent inside a medical device or clinical workflow inherits its obligations.
- **ISO/IEC 42001:2023** [15], the AI management-system standard, is the voluntary counterpart — a certifiable framework for governing AI across its lifecycle, increasingly asked for in procurement.

The core point is that these regimes ask for the same operational hygiene, so it is worth reading them not as four burdens but as one single checklist, and the AI Act gives — in our view — the clearest version of it. Deep down, it's composed of three sensible steps:

1. **Train your humans.** People who deploy and supervise agents need to understand them; organisational literacy is the first control and the one no tool replaces.
2. **Register your agents.** Treat every agent like any application: one inventory entry per agent, with an id, a name, a stated purpose, a single named owner, the harness version, the models and tools it may use, and a risk classification. Registration — not a dashboard — is where governance starts, and the AI Act effectively mandates it.
3. **Log and enforce.** You need a runtime record of what each agent did *and* the ability to act on it. Note the dependency from section 7.1: the log is trustworthy because the same layer that wrote it could also have blocked the action. Enforcement is what makes the record evidence rather than narration.

To close this section: In our view, if you get the security right, governance becomes mostly a routing problem — simply send the trace and the incidents to the systems that are already your source of record. The control is new because the threat is new; everything downstream of it you (likely) already own.

KYVVU **Governance falls out of the trace**

Because the Agent Security Kernel mediates every action in-process, a hash-chained behavioural trace of the whole task is produced as a by-product — no separate instrumentation. That trace ships to wherever you already work: OTLP to Datadog, Splunk, or Sentinel; JSON or HTTP or file; with a metadata-only mode that redacts content while preserving shape. Registration is enforced too — an agent must declare its purpose, tools, and risk class before its first step. The runtime evidence this produces supports EU AI Act record-keeping and human-oversight obligations (Arts. 12, 14) as a by-product of enforcement, not a separate documentation exercise — and the same trace serves GDPR and ISO/IEC 42001 work. Compliance evidence is the bonus here; stopping the harm is the reason.
Product details defer to docs.kyvvu.com.

KYVVU **Getting started**

If you want to try the path-aware approach on your own agents: `pip install kyvvu` installs the SDK and the in-process engine; `kyvvu register` sets up an account; `kyvvu init` scaffolds a demo agent. Start in observe mode (`warn`) to see what your agents actually do — a read-only safety net — then switch to `block` to enforce, one agent at a time. The SDK is Apache-2.0 and the engine is BSL-1.1 source-available, so you can read how every decision is made; it is EU-domiciled and your data stays in your environment.

- Docs: docs.kyvvu.com
- Platform: platform.kyvvu.com
- Paper: arXiv:2603.16586

Three ways to take it further:

1. **see it live** — a 90-second demo of the read-then-send block a stateless check cannot reproduce (kyvvu.com);
2. **evaluate any vendor** — the properties in section 6.2 double as a buyer's rubric you can score any agent-security tool against;
3. **talk to a reference** — Kyvvu runs in production at European financial-services, insurance, and healthcare organisations, deployed with integration partners.

8. Where do we go from here?

We think it will be sooner rather than later before pretty much any organisation runs not one agent but thousands, built on a multitude of models — hosted, self-hosted, embedded — wired into all of its internal applications and a long tail of external ones, each task passing through a mix of quality gates: human approvals, model-based guardrails with published benchmarks, deterministic I/O checks. That is the environment we are actually trying to secure.

At that scale, two things become impossible. First, you cannot secure each agent by hand, and you cannot trust each model to simply behave because you prompted it nicely. You need security and governance that apply to the whole fleet — uniformly enough to be tractable, but configurable per agent, because a coding agent and a refund agent do not carry the same risk. Exactly how to make this tractable across a large fleet of potentially interacting agents — and how granular the path-policies should be — is an open problem, not a solved one [24]. Second, your security rules must live *outside* the agent and travel *with* it — not inside a system prompt the model can be argued out of, and not inside a gateway the next local or embedded model simply bypasses. The rules have to sit around the one place every agent's action actually passes through: the agent's own execution harness, in your environment, where the path is visible and a decision can be made before the action runs.

That is the argument in one breath. Agents are models that act, because a harness executes the plan generated by a model. They are ordinary software but for one property — the path, the sequence of tool calls and their arguments, is chosen at run time. The path is also where the new danger lives. You secure it by bounding it at run time, in your environment, in the harness, with rules you own. Governance follows from having done so.

9. What this paper does not argue

A good argument states its limits, so to be explicit, we do *not* claim:

1. that the path is the *only* danger — the individual action and the content matter too, and we said so;
2. that gateways, IAM, or guardrails are wrong — they are right for what they do, and we run them ourselves;
3. that our own product is the only valid path-aware control — the *category* (a path-aware reference monitor) is the point, not the vendor;
4. that compliance reporting is unimportant — only that it is downstream of enforcement, and never a substitute for it.

If you disagree with the central claim of this paper — that the path is the right object of governance — that is the discussion worth having. The four points above are simply not claims we are making.

APPENDIX

A. Incident catalogue

A representative selection from a larger rolling catalogue of 2025–2026 agent-security incidents we maintain. Each is tagged with one of the three dangers from section 4 — **Action**, **Content**, or **Path** — that it best exercises. Sources are the originally-reported primary or near-primary outlet.

Date	Incident	Danger	Source
2025-10	Perplexity Comet ("CometJacking") — URL-parameter injection exfiltrates Gmail / Calendar data	Content	[17]
2026-01	Cursor (CVE-2026-22708) — allowlisted commands deliver arbitrary payloads	Action	[21]
2026-03	OpenClaw / ClawHub — ~36% of third-party marketplace skills carried prompt-injection flaws, each running with the agent's full power	Content	[16]
2026-03	Meta internal agent — SEV1: an autonomous action without approval broadened data access (missing approval gate)	Path*	[9]
2026-04	PocketOS — Cursor agent (on Claude Opus) deletes production database in 9 seconds	Action	[5]
2026-05	Semantic Kernel — prompt injection → RCE, "prompts become shells" (CVE-2026-26030 / -25592)	Content	[18]
2026-05	Sysdig — first in-the-wild LLM-agent intrusion: CVE → internal DB in 4 pivots, <1 hr (attacker's agent)	Path	[8]
2026-06	Miasma worm — 73 Microsoft repos; malicious config auto-runs in AI coding tools	Action	[19]
2026-06	Agentjacking — fake Sentry errors hijack coding agents via MCP	Content	[20]
2026-06	M365 Copilot "SearchLeak" (CVE-2026-42824) — one-click injection → SSRF mailbox exfiltration	Content	[7]

* *Meta is tagged Path as the least-bad fit — the harm was an un-gated autonomous step in a sequence — but the mechanism was a missing approval gate, not a permitted read-then-send exfiltration.*

Broader guidance referenced in this paper — the CISA / Five Eyes agentic-AI security guidance [22], OWASP's State of Agentic AI [23], and Gartner on non-uniform governance [24] — is listed under References (Appendix B). The Source column above gives the reference number for each incident.

B. References

The same reference numbers are used in the text and in the Appendix A source column.

[1] Adversa AI (2026). *AI Risk Quadrant (AIRQ): security scoring of 100 production AI agents* — ~98% exhibit the "lethal trifecta." airiskquadrant.com; coverage: helpnetsecurity.com.

[2] M. Kaptein & J. (Joris) Janssen (2025). *AI Agents at Work.* theaiagentbook.com.

[3] Cloud Security Alliance / Token Security (2026). *82% of Enterprises Have Unknown AI Agents in Their Environments* — and 35% report financial losses from agent incidents. cloudsecurityalliance.org.

[4] Cloud Security Alliance / Zenity (2026). *Enterprise AI Security Starts with AI Agents* — ~50% of organisations had at least partially documented agent-governance policies; only 31% had formally adopted one. cloudsecurityalliance.org.

[5] The Register (2026). *Cursor/Opus agent snuffs out a startup's production database (PocketOS).* theregister.com.

[6] K. Greshake, S. Abdelnabi, S. Mishra, C. Endres, T. Holz, M. Fritz (2023). *Not What You've Signed Up For: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection.* ACM AISec '23. arXiv:2302.12173.

[7] Varonis Threat Labs / The Hacker News (2026). *M365 Copilot "SearchLeak" — one-click data exfiltration (CVE-2026-42824).* thehackernews.com.

[8] Sysdig (2026). *AI agent at the wheel: from a CVE to an internal database in four pivots* (observed 10 May 2026). sysdig.com.

[9] The Guardian (2026). *Meta AI agent's misread instruction caused a large sensitive-data leak to employees.* theguardian.com.

[10] J. H. Saltzer and M. D. Schroeder (1975). *The Protection of Information in Computer Systems.* Proceedings of the IEEE, 63(9), 1278–1308. doi:10.1109/PROC.1975.9939. — reference monitor, complete mediation, least privilege, open design.

[11] M. Kaptein, V.-J. Khan, A. Podstavnychy (2026). *Runtime Governance for AI Agents: Policies on Paths.* arXiv:2603.16586.

[12] *EU AI Act* — Regulation (EU) 2024/1689. eur-lex.

[13] *GDPR* — Regulation (EU) 2016/679. eur-lex.

[14] *MDR* — Regulation (EU) 2017/745. eur-lex.

[15] *ISO/IEC 42001:2023* — AI management systems. iso.org.

[16] Snyk (2026). *ToxicSkills: a security audit of the AI-agent skills supply chain* — prompt injection detected in ~36% of ClawHub skills (1,467 malicious payloads). snyk.io.

[17] LayerX (2025; updated 2026). *CometJacking: how one click can turn Perplexity's Comet AI browser against you.* layerxsecurity.com.

[18] Microsoft Security (2026). *When prompts become shells: RCE vulnerabilities in AI agent frameworks (Semantic Kernel; CVE-2026-26030 / -25592).* microsoft.com.

[19] The Hacker News (2026). *Miasma worm hits 73 Microsoft GitHub repositories via AI coding agents.* thehackernews.com.

[20] Tenet Security (2026). *Agentjacking: hijacking AI coding agents with fake Sentry errors via MCP.* tenetsecurity.ai.

[21] Pillar Security (2026). *The agent security paradox: when trusted commands in Cursor become attack vectors (CVE-2026-22708).* pillar.security.

[22] CISA & Five Eyes partners (2026). *Careful Adoption of Agentic AI Services.* cisa.gov.

[23] OWASP GenAI Security Project (2026). *State of Agentic AI Security and Governance* (genai.owasp.org); *OWASP Top 10 for LLM Applications* (genai.owasp.org).

[24] Gartner (2026). *Applying Uniform Governance Across AI Agents Will Lead to Enterprise AI Agent Failure.* gartner.com.

[25] Capsule Security / VentureBeat (2026). *PipeLeak (Salesforce Agentforce) & ShareLeak (Microsoft Copilot Studio, CVE-2026-21520): prompt injection exfiltrates enterprise data.* venturebeat.com.

[26] Palo Alto Networks Unit 42 (2026). *Double Agents: exposing security blind spots in GCP Vertex AI.* unit42.paloaltonetworks.com.

C. Glossary

- **Agent** — software that uses a foundation model to decide what to do and a harness to do it.
- **Harness** — the deterministic program that runs the loop: assemble prompt → call model → execute the plan's tool calls → feed results back → repeat.
- **Model call** — a reasoning step: text to text (even when that text contains harness instructions), with little direct effect on the world.
- **Tool call** — an action step with a real effect: a file read, a database write, an email, a shell command.
- **Trigger** — what starts a run (message, webhook, schedule, file); part of the threat surface.
- **MCP (Model Context Protocol)** — the emerging standard for exposing tools and resources to agents.
- **Internal resource** — a trusted resource or model inside your perimeter (databases, knowledge bases, internal tools).
- **External resource** — an untrusted resource or model outside your perimeter (open web, third-party APIs, inbound messages); the boundary where injection enters and exfiltration can leave.
- **Isolation / sandboxing** — confining the agent to cap the blast radius of any single action.
- **Path-dependent / policies on paths** — evaluating an action against the full ordered history of the task, not in isolation.
- **Observability** — the ability to see what an agent did.
- **Enforcement** — the ability to stop what an agent is about to do; enforcement implies observability, but not the reverse.
- **Integrity (of the trace)** — the property that the record is trustworthy as to what happened and in what order, because the same mechanism that recorded each step could have blocked it.
- **Prompt injection / tool poisoning** — untrusted content (or a tool's description/output) carrying instructions the agent then follows.
- **Agent Security Kernel (ASK)** — an in-process reference monitor that evaluates every action against the task path and returns allow / warn / block before it runs.