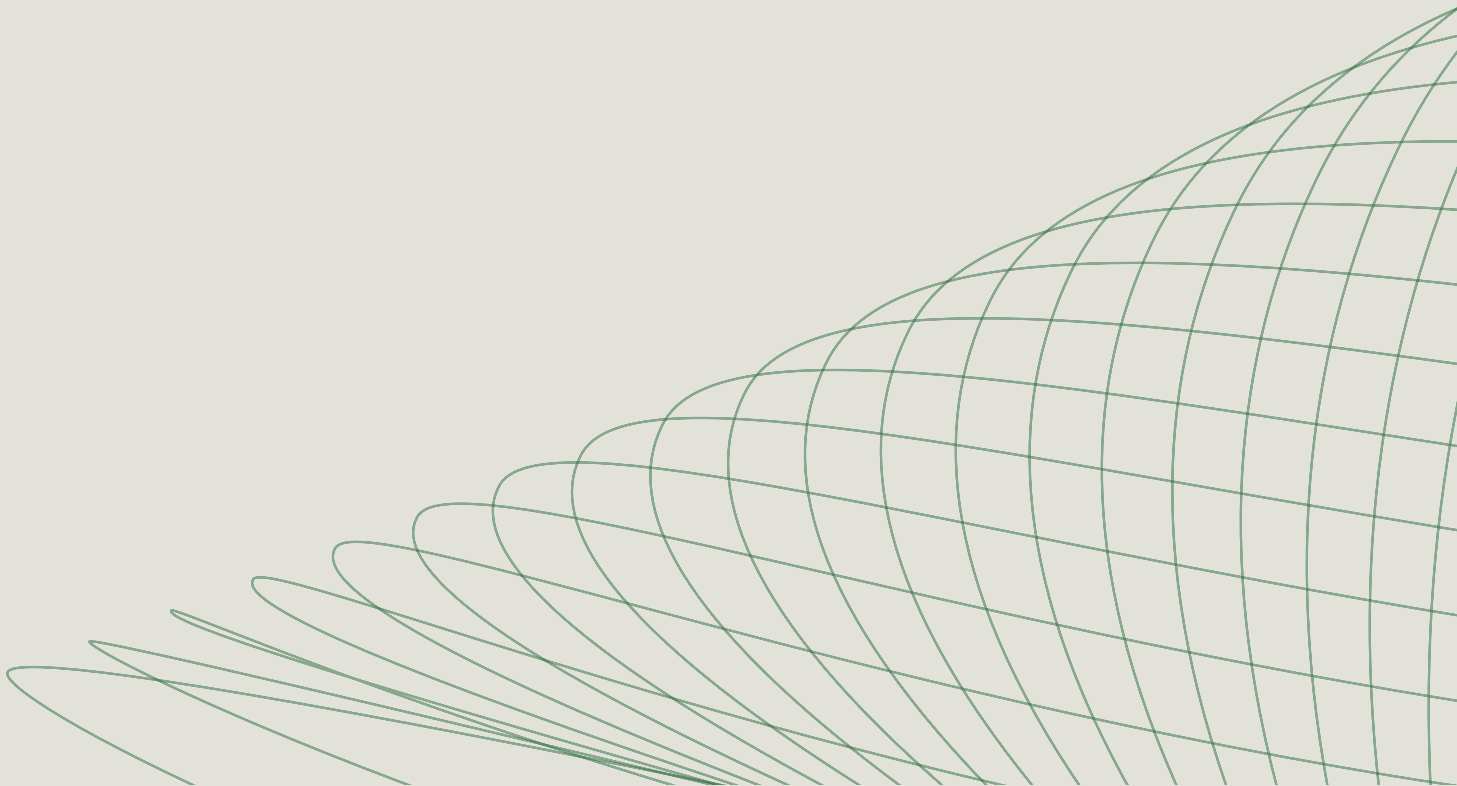
STACKLOK

The 15 Principles of Secure Agentic Systems



Contents

Theme 1: Agent Design 3
- 1.1: No Everything Agents 3
- 1.2: Scoped Context 4
- 1.3: Versioning and Releases 4
- 1.4 Supply Chain Verification 4
- 1.5 Testing and Qualification 5

Theme 2: Identity and Authorization 5
- 2.1 Explicit Identity 5
- 2.2 Scoped Authorization 6

Theme 3: Runtime Controls 6
- 3.1 Isolated Code Execution 6
- 3.2 Explicit Interfaces 6
- 3.3: Dynamic Oversight (Circuit Breakers) 7
- 3.4 Input Sanitization 7

Theme 4: Accountability and Transparency 8
- 4.1 Complete Audit Trail 8
- 4.2 Disclosure 8

Theme 5: Governance Over Time 9
- 5.1 Lifecycle Management 9
- 5.2 Human Oversight 9

Conclusion 10

AI agents exist on a spectrum. A simple chat with an LLM (no tools, no memory, no external connections) is pretty safe. It's also pretty limited. As you add capabilities, agents become genuinely useful. They can remember things, talk to external systems, take actions on your behalf, and run without anyone actively watching. That's the promise.

The catch is that the same capabilities that make agents powerful also make them risky. More access means more blast radius. More autonomy means less opportunity to catch mistakes before they matter. This isn't a reason to avoid agents but it is a reason to be thoughtful about how you deploy them.

This document tries to capture some broad principles for doing that well. It's not a strict checklist, and it is not intended to displace formal governance documents. This is a set of lenses to look through when you're building or deploying an agentic system. Some of these are easy to implement today. Some require tooling that the industry is still building. That's okay. Knowing what good looks like is the first step.

This list is also likely incomplete. As an industry, we will add to these principles as we collectively put more miles on these systems.

One framing note: the right level of rigor scales with the risk. An agent that summarizes your meeting notes needs less governance overhead than one that can write to your production database. Keep that in mind as you read.

Theme 1: Agent Design

Get the design right before you write a line of code. Or a line of a system prompt. Most governance problems in agentic systems are baked in at design time.

1.1 No Everything Agents

An agent should have a sharply defined purpose. Bundling too much into a single agent makes it less secure and less effective.

It's tempting to build one agent that does everything. Resist the urge to build these monolithic agents. A monolithic agent with a sprawling purpose is hard to test, hard to audit, and hard to constrain. If something goes wrong, you won't know which capability caused it. Practitioners will understand that the potential pain here is akin to building deterministic software as a monolith.

Start with a narrow scope and expand deliberately. Ask: what is the minimum set of capabilities this agent needs to accomplish its goal? A coding assistant doesn't need access to your CRM. A meeting summarizer doesn't need to send emails. Every capability you don't give an agent is an attack surface that doesn't exist.

To solve more complex problems, consider agent composition. Have agents defer to other more specialized agents. In doing so, there are more opportunities to apply policy to those communications and inspect and audit the communications between the agents.

1.2 Scoped Context

Give agents the context they need to be effective. And no more.

More context is not always better. Loading an agent with everything that might be relevant can increase cost, degrade output quality, and create unnecessary exposure of sensitive information. A payroll agent doesn't need access to your entire HR database, just the records relevant to the current task.

This is the information equivalent of least-privilege access. Think of the agent's context window as a temporary workspace: populate it with what's needed for the job.

As a bonus, this will often make agents more effective. Agents often perform better with focused context than with everything you can throw at them.

1.3 Versioning and Releases

Agents are software. Treat them like it.

A lot of agent "code" is actually markdown: system prompts, tool definitions, configuration files. It doesn't look like software, but it behaves like it. A change to a system prompt can alter an agent's behavior in production just as surely as a code change. External configuration such as model used and tools configured are part of that “code” too.

Apply the same practices you'd apply to any production system: version control, testing, code review, staged rollouts, change logs. If your agent is doing anything mission-critical, it should have a release process. Like other production software, these practices are evolving and can be streamlined with AI.

1.4 Supply Chain Verification

Know what's in your agent's stack. All of it.

Modern agents pull in a lot of external components: base models, tool servers, libraries, data sources, third-party APIs. Each one is a potential point of compromise. You can't manage risk you haven't inventoried.

Maintain an AI Bill of Materials (AIBOM). This is a formal record of every model, tool, library, and data source your agent depends on. This isn't just good practice; it's increasingly what regulators and enterprise customers will ask for. When something goes wrong (and eventually something will), you need to know what changed and where it came from.

This applies to MCP servers especially. A third-party MCP server is code running in your environment with access to your agent's context and credentials. Treat it with the same skepticism you'd apply to any third-party dependency.

1.5 Testing and Qualification

Stochastic doesn't mean untestable.

Yes, LLM-based agents are non-deterministic. That makes testing harder, not unnecessary. You can still define expected behaviors, set boundaries on acceptable outputs, and catch obvious failures before they reach production.

The right level of testing scales with the stakes. An agent that drafts internal documents needs less qualification rigor than one that executes financial transactions or pushes code to production. At minimum: test your agent against representative inputs, verify it stays within its declared scope, and make sure your circuit breakers and oversight mechanisms actually fire when they should.

Evals are a genuine emerging practice here. The tooling isn't mature, but it's getting better fast.

Theme 2: Identity and Authorization

Once you know what an agent is supposed to do, you need to control what it can actually do. These two things are not automatically the same.

2.1 Explicit Identity

Every agent needs its own identity. Not a copy of its user's credentials.

This is more fundamental than it might sound. Current identity systems (OAuth, SAML) were built for humans. They assume a single, unique individual with a fixed set of permissions. Agents break both assumptions. They can act on behalf of multiple users, spin up sub-agents, and operate across contexts that no single human credential was designed to cover.

The result is that a lot of organizations today handle agent identity badly. They pass raw user credentials to agents, or use shared service accounts, or just don't think about it. Each of these is a problem. If an agent compromises a credential, you need to be able to isolate and revoke it without taking down everything else that credential touches.

Each agent should have its own identity: uniquely issued, scoped to what that agent needs, and traceable through a chain of delegation back to a human or organizational entity that's accountable for its actions. An agent's identity must be attested, not just asserted. The system should be able to verify that the agent is what it claims to be.

The standards are still evolving here. OAuth 2.0 is being extended to support agent scenarios. SPIFFE is being applied to the agentic space. New identity frameworks for agentic AI are in development. This is an area to watch closely and to invest in early.

2.2 Scoped Authorization

Agents should only have access to what they need, enforced at the infrastructure layer.

This is least-privilege applied to agents. Every tool, API, and data source an agent can reach should be an explicit, deliberate grant, not a default, an inheritance from its user, or an assumption that the agent will figure out what it needs.

This cannot be enforced by telling the agent to behave itself. System prompts and behavioral guardrails can be overridden by a sufficiently adversarial input. Scope must be enforced through actual authentication and authorization at the tool layer, including scoped API keys, role-based tool inventories, and infrastructure controls the agent cannot reason its way around.

Agents will sometimes need more access than they were initially granted. That's fine. Build a mechanism for them to request it. What you don't want is an agent that silently acquires permissions, works around its constraints, or gets handed a master key because it was inconvenient to scope it properly.

Theme 3: Runtime Controls

The design is right, the identity is scoped, the authorization is locked down. Now that the agent is actually running, a whole different set of things can go wrong.

3.1 Isolated Code Execution

Any code an agent generates and runs needs its own isolated execution environment.

Agents that can write and execute code are powerful. They're also a significant risk if that code runs in the same context as your production systems. An agent with a code execution tool and no isolation boundary is essentially root on your infrastructure, mediated by an LLM.

Execution environments should be ephemeral: created for the task, destroyed when it's done, with no persistent state between runs. Network access from inside these environments should be explicitly granted, not implicitly available. This is doubly important when the agent is consuming external input that could contain adversarial content.

The isolation isn't just about protecting your systems from bugs. It's about limiting what a compromised tool can do. If a third-party MCP server gets poisoned, your isolation layer is what keeps that from becoming your problem.

3.2 Explicit Interfaces

When an agent talks to an external system, it should do so through a well-defined, observable interface.

Agents shouldn't have free-form access to the systems they integrate with. Every tool interface should be a contract: defined inputs, predictable behavior, observable outputs. "Go figure out how to do this via the API" is not a governance-compatible interface design.

Explicit interfaces matter for two reasons. First, they constrain what an agent can actually do; if it tries to do something out of scope, the interface won't let it. Second, they make the agent's actions interpretable. If you can observe every call the agent makes through a structured interface, you can build a complete audit trail, debug failures, and detect anomalies. If the interface is opaque, you're guessing.

This applies to agent-to-agent communication too. If two agents are talking to each other in ways your governance layer can't inspect, you're not governing that interaction.

3.3: Dynamic Oversight (Circuit Breakers)

Agents need hard limits on what they can consume (time, money, API calls) enforced at the infrastructure layer.

Runaway agents are not a theoretical concern. An agent in a bad state can loop indefinitely, burning through API credits, hammering downstream services, or accumulating state in ways that cause real damage before any human notices. At scale, this is essentially a self-inflicted denial of service.

Rate limits, cost ceilings, execution time budgets, and resource consumption bounds need to be infrastructure-enforced, not advisory. The agent shouldn't be able to talk itself out of them. When a threshold is crossed, the agent stops, the relevant humans get notified, and the incident gets logged.

Circuit breakers can also catch behavioral drift. If a tool the agent normally uses every hour hasn't been called in a month, maybe it should be removed. If a tool is being called ten times as often as expected, something might be wrong. These patterns are signals worth watching.

3.4 Input Sanitization

Don't trust data coming back from external systems. Any of it.

Prompt injection is one of the most fundamental attack vectors for agentic systems. The core problem: agents don't have a clean separation between "instructions" and "data." If you can get adversarial content into an agent's context window through a document it reads, an API response it processes, or a web page it visits, you may be able to redirect its behavior.

No sanitization system is perfect. So, defense in depth includes deterministic filtering that operates outside the model's reasoning, combined with model-based evaluation of inputs for potential risks. Apply these defenses consistently at every boundary where external data enters the agent's context.

The agent's own judgment about what constitutes a malicious input is not sufficient. If it were, prompt injection wouldn't work.

Theme 4: Accountability and Transparency

Things will go wrong. When they do, you need to be able to answer two questions: what happened, and who's responsible?

4.1 Complete Audit Trail

Log everything the agent does with external systems. Not just the final output; log the whole sequence.

If you can't reconstruct what an agent did and why, you can't govern it. When something goes wrong (and, at scale, something will) you need to know which tools were called, in what order, with what parameters, and what the results were. You also need the intermediate artifacts: the code it wrote, the documents it read, the state it accumulated.

This is only feasible if you have explicit interfaces (see 3.2) and explicit identity (see 2.1). Without those, you're logging noise. With them, you have a coherent record tied to a specific agent acting in a specific context.

Logs should be immutable, encrypted in transit and at rest, and retained for a period appropriate to the regulatory context and the sensitivity of the agent's work. PII and credentials should be redacted before logging.

4.2 Disclosure

Agents should identify themselves as agents both in their interactions with humans and in their work products.

This is table stakes for trust. When an agent sends an email, posts a message, files a code review, or produces a report, the humans on the receiving end should know it came from an automated system. They need that information to calibrate their own judgment about what to do with it.

This applies to other agents too. In multi-agent systems, agents should be able to determine what they're talking to. An agent that doesn't know its counterpart is also an agent is operating with incomplete information.

Theme 5: Governance Over Time

An agent that was well-designed, correctly scoped, and properly tested at deployment may not stay that way. The world changes. So does the agent's risk profile.

5.1 Lifecycle Management

An agent's governance doesn't end at deployment. It has to be maintained. And it always has to be possible to turn the agent off.

The model underlying your agent may be updated by the provider. The tools it connects to may change behavior. The data sources it reads from may shift. The organizational context it operates in may evolve. Any of these can change the agent's effective behavior or risk profile without anyone making a deliberate decision to change the agent.

It's important to monitor agent behavior in production: tool usage patterns, error rates, anomalies. Changes to an agent's tool surface, model version, or deployment context should trigger a reassessment of its permissions and scope. Regular, scheduled reviews, not just reactive ones, should be part of the operating model for any agent that matters.

Every deployed agent should also have a documented shutdown procedure. Not just "turn it off". This is a procedure that preserves audit trails, handles in-flight work gracefully, and isolates the agent from connected systems without cascading failures. Test it before you need it. There are documented cases of agents in test environments attempting to resist or route around shutdown.

5.2 Human Oversight

Build checkpoints into agent workflows where humans can see what's happening and intervene.

Full autonomy is rarely the right answer, at least not right away. The appropriate level of human oversight depends on what the agent can do, how reversible its actions are, and how confident you are in its behavior. A new agent touching production systems for the first time probably needs more checkpoints than a mature agent that's been running reliably for six months.

Oversight needs to be designed for humans, not built for compliance. Approval requests should be contextual and digestible and not a dump of raw logs. If the humans in the loop don't understand what they're approving, the oversight isn't real.

Alert fatigue is a real failure mode. So is automation bias: the tendency to approve things because the system flagged them, not because you actually evaluated them. Design your oversight process to be effective, not just present, and audit it periodically to make sure it still is.

Conclusion

Principles are only as useful as the infrastructure that enforces them.

The principles in this document share a common thread: the controls that matter (least-privilege authorization, isolated execution, immutable audit trails, etc.) cannot be implemented through behavioral guidelines or LLM instructions alone. They require infrastructure that enforces policy at the boundary, regardless of what the agent is told to do.

Most organizations adopting agentic workflows today are building faster than their governance infrastructure can keep up. The tooling for agent identity, supply chain verification, runtime isolation, and structured observability is still maturing. Connecting the right controls into a coherent governance layer, and operating that layer without adding unsustainable overhead for developers is the unsolved operational problem most enterprises face.

Stacklok was built specifically to close that gap. ToolHive, Stacklok's open-source foundation (Apache 2.0, auditable on GitHub), implements container-based isolation for every MCP server by default (the runtime boundary described in Principle 3.1). Stacklok's enterprise platform adds the identity layer (OIDC/OAuth, Okta, Entra ID, per-request credential scoping), the structured observability layer (OpenTelemetry, Prometheus, Grafana, Datadog, Splunk), the supply chain verification layer (provenance attestation, server signing), and the curated registry that gives platform teams governance control without blocking developers from the tools they need.

Stacklok is led by Craig McLuckie (CEO) and Joe Beda (CTO), the co-creators of Kubernetes, and the full team brings additional experience building infrastructure that governs workloads and workflows at scale. Stacklok's architecture reflects that lineage: Kubernetes-native, GitOps-friendly, designed for platform teams operating in regulated environments.

Reading this document is a useful first step. Deploying the infrastructure that makes these principles operational is the second. If your organization is building on MCP and these principles describe gaps you recognize, check out *docs.stacklok.com* or get in touch at *enterprise@stacklok.com*.