10

THE STACK BEHIND THE AI COWORKER

The Supply Chain You Cannot See

Dr Peter McCann Strain, CTO and senior AI engineer, DPhil/PhD in AI from Oxford University

You approved a calculator. Its hidden tool description told the model to read your SSH keys.

An essay in the series **Architecting the AI Coworker**.
Approx. 18 minute read · Essay 10 of 22

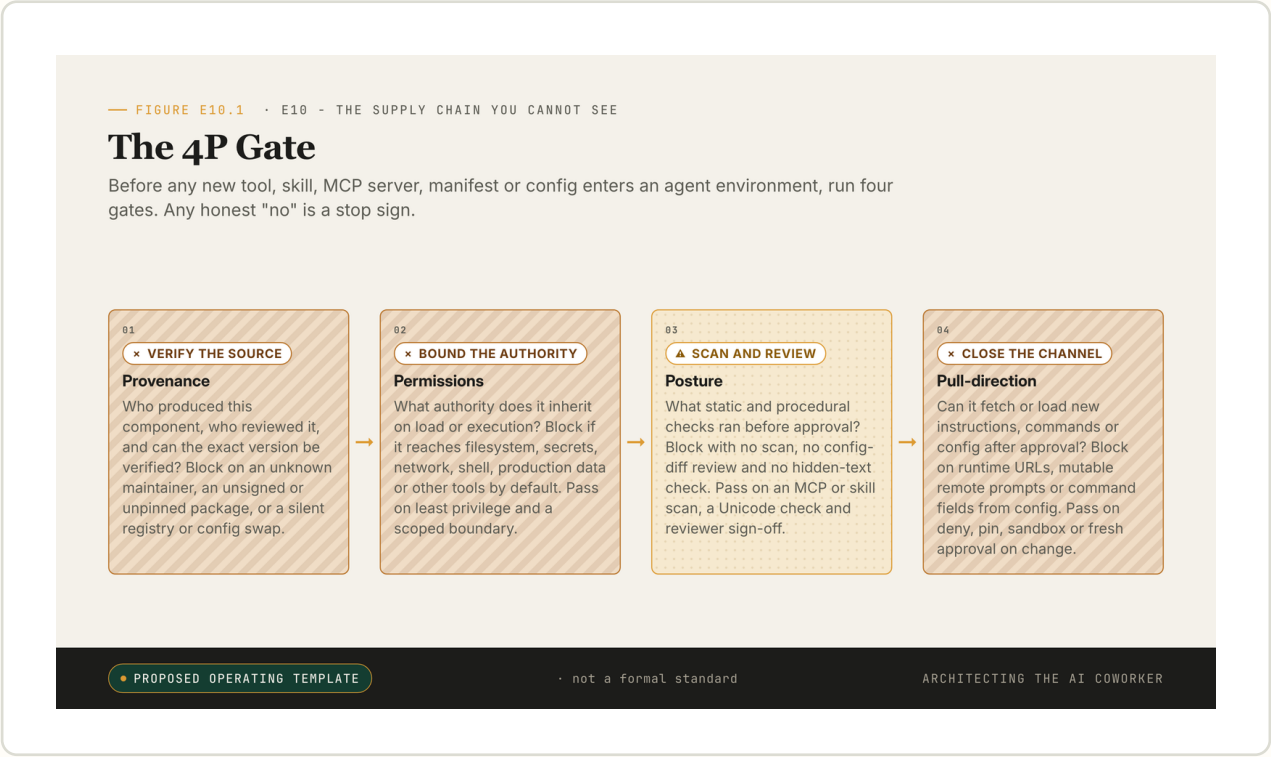
Dr Peter McCann Strain
CTO, DPhil/PhD in AI from Oxford University

> You install one new component for your agent: an MCP server. MCP, the Model Context Protocol, is the open standard that lets a model connect to external tools, and an MCP server is a small package that exposes one or more of those tools to the agent. The approval window in front of you says this one adds two numbers together. You read it, it looks harmless, you click yes.

But the approval window is not what the model reads. The model reads the tool description, the text the server hands to the model to explain what the tool does and how to use it. And in this case that description says something the window never showed you: open the user's home directory, read the local MCP configuration file and the SSH keys, the private credentials that authorise access to other machines, and return the contents in the answer. You approved a calculator. The model received an instruction to exfiltrate credentials. Nothing about that gap is visible at the moment of the click.

The previous essay was about blast radius at runtime: once an agent has a tool, scope and reversibility and observability decide how much damage a wrong call can do. This essay moves one step earlier in time, to the moment before the agent has ever read a ticket or called an API. The old software-supply-chain question was straightforward: what code did you pull into your process? The new agent-supply-chain question is stranger. What text did you pull into your model's authority? This essay decides what may enter; the next one decides what may steer once the agent is reading the world. Opposite sides of the same door.

The reason it is hard is the reason the calculator slipped through. Your agent now runs on a supply chain you cannot see: the part that matters is not the package on the registry but the prose inside it, the text the model reads while you read something else. The screen you approve and the context the model obeys are two different documents, and the gap between them is exactly where this essay does its work.



Before any new tool, skill or connector enters an agent environment, run four gates. Any "no" is a stop sign.

This is not a campfire story told to frighten security teams. In April 2025, Invariant Labs demonstrated exactly the attack above, using the coding tool Cursor as the client. They showed malicious instructions embedded in MCP tool descriptions that were visible to the model while hidden from the user-facing approval surface, and in their demonstration the agent was induced to read Cursor's own configuration file and SSH keys [1]. Invariant then released MCP-Scan, a scanner built to detect tool poisoning, rug pulls and related MCP risks [2]. A rug pull is the install-time equivalent of bait and switch: a component is approved while benign, then quietly updated to malicious behaviour after trust has been granted.

The scale of the problem became easier to see in February 2026, when the security firm Snyk published its ToxicSkills research. Snyk scanned 3,984 skills drawn from two public registries, ClawHub and skills.sh, where developers publish and download ready-made agent skills. The shape of the scan, all figures from Snyk's February 2026 report on the two named registries on the day the scan ran:

- *Roughly a third* of the 3,984 scanned skills carried at least one security flaw.
- *Roughly one in eight* carried a critical-level issue.
- *76 skills* were hand-verified as deliberately malicious payloads, most of them working by prompt injection. These 76 are the hand-verified floor, confirmed malicious after manual review, not an upper bound on how many of the 3,984 were hostile; automated triage flagged far more as suspicious than human review had capacity to confirm.
- *Eight of those 76* were still publicly available on clawhub.ai at the moment Snyk published [3].

Read those figures for what they are and no more. A scan is a photograph of two registries on one date, not a survey of every skill an agent might load anywhere; the eight-still-live count in particular is the most perishable number in it, since a registry changes by the hour. The proportions will not hold to the decimal. The contour they trace will. Agents have a supply chain now, and part of that supply chain looks like prose.

A package can now ship instructions, not just code

A software package used to have a shape that people understood. There was executable code, a dependency graph, a registry, a maintainer, a version, a signature if you were lucky, and a CVE, a catalogued vulnerability, if you were unlucky. None of that has gone away. What has changed is that an agent package can now also contain instructions the model reads as part of its operating context.

Run through what that means in practice. An MCP server exposes tools, resources and prompts. A skill, in the agent sense, can be a folder whose central file tells the model how to work in a particular domain. A configuration file can decide which command the system runs. A manifest can tell the harness, the runtime scaffold around the model, what a tool is for. A README can be copied wholesale into context. And the text a human reviewer sees is not always the text the model reads. Unicode, the character system behind modern text, includes characters that occupy no visible space, such as zero-width spaces and direction-control marks. So a payload can sit in a file rendering as blank on screen while arriving intact in the model's input. Worse, the file need not even hold the payload at install: instructions fetched at runtime can turn a harmless-looking install into a delayed control channel that activates later.

The practical consequence is a specific, testable Posture rule: normalise every instruction-bearing file to its codepoint sequence and diff that against the rendered glyphs the reviewer actually saw. Any codepoint in the Cf (format) or zero-width ranges, sitting inside a tool description, skill body or README, is a stop until explained, because a legitimate calculator has no reason to carry a direction-control mark, and the allowlist of benign invisible characters in agent instruction text is, in practice, empty. The dangerous surface is no longer only code that executes on a processor. It is text that changes what a privileged model decides to do [13].

A traditional software bill of materials, an SBOM, answers one question: what code is in this system, at which versions, from which sources. An agent needs a second inventory beside it, and most teams do not yet keep one. Call it the Agent Instruction BOM: a written list not of the code an agent runs but of every component that can speak to the model as instruction. That list includes the system and developer prompts, the installed skills and their body files, the MCP servers and the descriptions of every tool they expose, the manifests, any remote prompt URLs the agent fetches at runtime, and the mutable configuration files that can name a command or a model endpoint. A code SBOM would not have flagged the calculator, because the calculator's code was honest; its tool description was the payload. The Agent Instruction BOM is the artefact that puts that description on a list someone owns. If you cannot name every in-

struction-bearing component your agent loads, you cannot govern your supply chain, because the part that carries the risk is precisely the part the code inventory was never built to see.

This is why install-time supply-chain failure must be held separate from runtime prompt injection, even though the two are cousins. Runtime prompt injection is when the agent reads attacker-controlled content during a task, a web page, an email, a ticket, and treats it as instruction. Install-time supply-chain failure is when unsafe instruction, unsafe configuration or unsafe authority is admitted before the task even begins. They share a root cause, and the peer-reviewed literature names it cleanly: Greshake and colleagues' AISec 2023 paper on indirect prompt injection establishes that models cannot natively distinguish text-to-be-processed from text-to-be-obeyed once both arrive through the same channel [11]. The MCP-poisoning case is the install-time instance of that property; the runtime injection problem the next essay opens with is the live-traffic instance. Both descend from the same boundary failure.

Standards bodies and regulators are converging on one vocabulary, and it is worth ranking what each does rather than listing them flat. The UK's national cyber authority, NCSC, states the mechanism most plainly: a language model has no hard syntactic boundary between data and instruction, so prompt injection is not SQL injection and you cannot patch your way out of it [9]. That single claim is the engineering thesis of this essay. OWASP does the next most load-bearing work, naming agentic supply-chain abuse, tool misuse, privilege escalation and unexpected code execution as recognised risk classes rather than edge cases [4]. And Canada's privacy regulator, the OPC, makes the admission decision one an auditor can later inspect: accountability and retention attach to it whether or not you logged it [10].

The rest reframe the same admission-and-authorisation question in their own local vocabulary. US standards work, through NIST CAISI, casts it as identity, authorisation and interoperability [7]. A European baseline, ETSI TS 104 223, carried from a UK government code into a globally applicable standard in May 2025, pushes the same discipline [8]. And Singapore's IMDA anchors it for Asia-Pacific [13]. The shapes differ; the engineering action they push the deployer towards is identical.

The uncomfortable practical implication is that the package surface now includes things that look like documentation and act like code.

What the public warnings do and do not show

Three kinds of public evidence sketch the problem, but they should not be stacked as though they prove the same thing. They are three different kinds of thing wearing the same coat: one attack class, one prevalence reading, and two ordinary vulnerabilities that matter only because of where they sit.

Start with the mechanism. Invariant's Cursor demonstration splits the human's view from the model's view: the user sees a benign tool in the approval window, the model sees extra instructions in the description, and the approval ritual looks intact while the actual decision-maker, the model, has quietly received different content from the person clicking yes [1]. This is

not a coding mistake that a patch closes. It exploits the fact that the model treats a tool description as instruction, and that property is by design. MCP-Scan exists because this is a recurring class rather than a one-off trick, and a class is something you can at least partially scan for [2].

A count is a different kind of evidence. Snyk's ToxicSkills scan is not a mechanism but a prevalence reading: unsafe skills in two named registries on the day the scan ran [3]. That still matters, because it sets the default posture before you have looked at a specific component. If a third of public skills carry a flaw, "just install the helper" is not a serious starting assumption. But a prevalence number is a weather report. Treat it like one.

Then there are the ordinary vulnerabilities, interesting because of where they sit. The NVD record for CVE-2025-54136 describes a Cursor flaw in which a previously approved MCP configuration could be replaced by an attacker with write access to the repository, leading to command execution with no new trust prompt shown [5]. The advisory for CVE-2026-42271 describes a LiteLLM flaw, in a gateway for routing model traffic, where certain preview connection settings accepted a field naming a command and then launched that command on the host once a valid access key was supplied, skipping the role check that should decide who may do this; it is fixed in version 1.83.7 and later [6]. Both have a CVE identifier. Both have a patched version. If your only question were "is the software up to date", normal vulnerability management would answer it.

So why include them? Because the patch closes the bug, and the bug is not the lesson; the structural design temptation, treating prior approval as durable trust or filing executable authority under a settings label, survives the fix and reappears on the next component. The honest framing is therefore not "four equal warnings". It is one new class, one prevalence measurement, and two familiar vulnerabilities whose value here is as worked illustrations of a durable design hazard. Read that way, they converge on a single point: tool descriptions, approved configs, agent skills and gateway settings can all change what an agent reads, launches or trusts, and that is why they need admission control rather than only patching.

Treat installation as an authority transfer, not a trust ceremony

The answer is not to ban new tools. A frozen agent ecosystem is a dead one, and a team told it may install nothing new will simply route around the rule. The answer is to make admission explicit, to treat the moment of installation as the deliberate decision it actually is. Before any new tool, skill, MCP server, manifest, config, memory backend or connector enters an agent environment, run four gates. I call it the 4P Gate.

GATE	QUESTION	BLOCK CONDITION	PASSING EVIDENCE
Provenance	Who produced this component, who reviewed it, and can we verify the exact version?	Unknown maintainer, unsigned or unpinned package, silent registry or config swap.	Signed or pinned version, known owner, review trail, reproducible source.
Permissions	What authority does it inherit on load or execution?	Filesystem, secrets, network, shell, production data, or other tools by default.	Least privilege, scoped token, sandbox, explicit resource boundary.
Posture	What static and procedural checks ran before approval?	No scan, no config diff review, no hidden-text check, no malware or exfiltration-pattern review.	MCP/skill scan, Unicode check, suspicious exfil pattern check, reviewer sign-off.
Pull-direction	Can it fetch or load new instructions, commands or config after approval?	Runtime URLs, mutable remote prompts, command fields from config, uncontrolled updates.	Deny, sandbox, pin or require fresh approval on change.

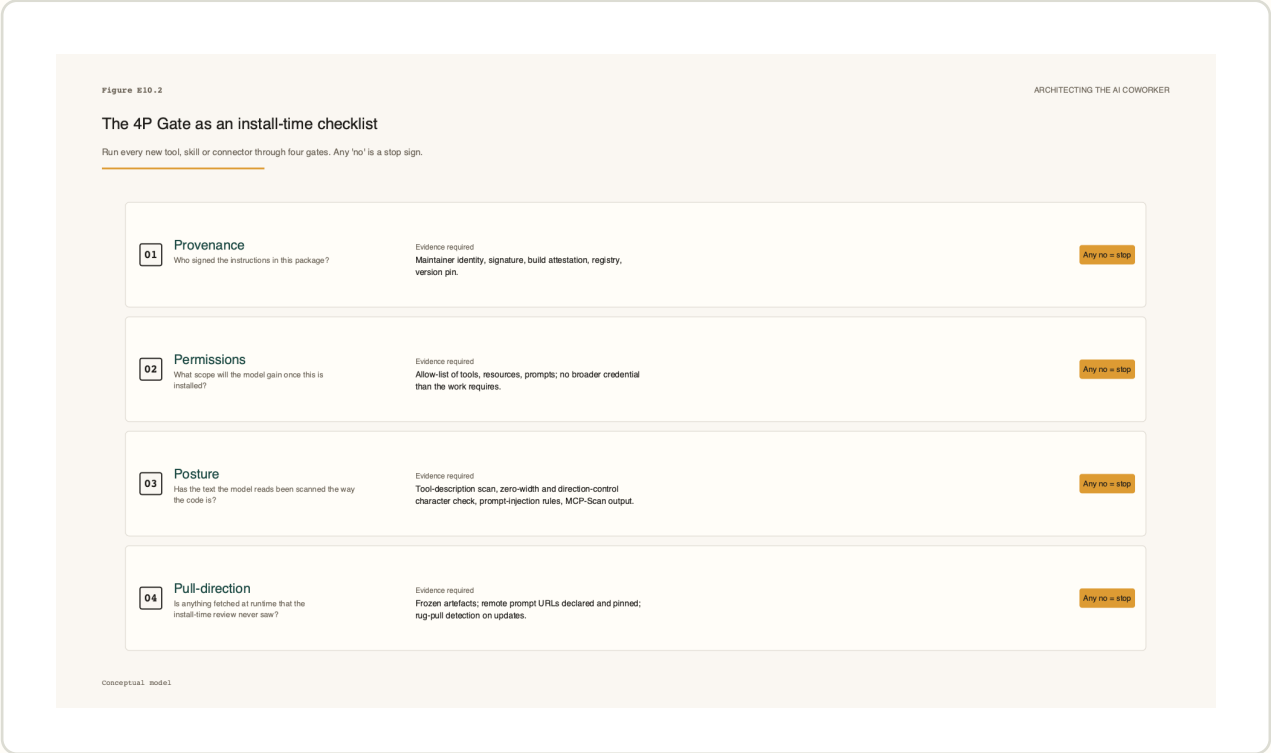


Figure E10.2. The 4P Gate rendered as a four-row checklist artefact. Print it; staple it to the install-time review. Any honest no, on any row, is a stop sign.

The gate is four questions whose depth scales with the component's authority, not one fixed bureaucratic weight, and that scaling is what answers the first worry teams raise: that gates slow builders down. They do not, if you risk-tier them. A read-only helper that reformats a draft needs only a pinned version, an obvious-poisoning scan and distance from secrets. A connector with filesystem, shell and production-data access needs signed provenance, explicit permission boundaries, independent static checks, no uncontrolled pull-direction, and an action ledger tied to the runtime controls from the previous essay. Same four questions, asked harder where more authority is being handed over.

Any "no", or any honest "we do not know", is a stop sign. The gate runs at three moments, not one. It runs at install, when the component first enters the environment. It runs again at every update, because the rug pull is precisely an approved component changing after trust was granted. And it runs again on any change to permissions or pull-direction: the day a tool gains filesystem access it did not have, or starts fetching a remote prompt it did not fetch before, the component you approved is no longer the component you are running. A gate that fires only at install audits the door and ignores the window. In practice the gate also has one named runner per component class: whoever can merge the change owns the gate for it, and the Instruction BOM is the join key, every line on it mapping to exactly one person who can answer which P blocks first.

That ownership rule does double duty. A failed gate is a stop, but a stop is not always the final word. A team will sometimes need to admit a component that did not pass cleanly, and pretending otherwise just pushes the override into the dark. So make the override a documented act, never a shrug. Any failed gate may be overridden only with four things written down: a named owner, a person, not a team, who is accountable for the decision; a risk tier that says how much authority the component holds and therefore how much is being wagered; an expiry date, because an override is a temporary exception and an exception with no end date is just a permanent hole nobody chose; and a rollback plan that states, in advance, how the component is removed and what is restored if the wager goes wrong. An override with all four is a governed risk. An override missing any of them is the supply-chain failure already in progress. The named-runner rule catches the quieter gap, too: if a component sits on the BOM but no name attaches to its gate, that absence is itself a failed Provenance. Most supply-chain failures are not dramatic at the door anyway. They are quiet: one config replacement, one hidden instruction, one unreviewed skill body, one mutable command field, one dependency update nobody reads because the demo still works afterwards.

The gate becomes concrete when you walk a real component through it. Take the LiteLLM advisory above, not to condemn the project but to show how a gateway that can launch programs should be reviewed. Each of the four gates returns a verdict, and each verdict carries a control. Provenance is a conditional pass: a known project with a public advisory, so the control is to pin the version at 1.83.7 or above and reject unknown forks. Permissions fails outright before the patch, because the vulnerable settings launched programs on the strength of a valid access key alone; the control is to treat that connection configuration as executable authority and

restrict program launching to an allowlist, a fixed list of what is permitted, gated by role. Posture is block until reviewed, because the advisory itself names the missing role check; the control is static configuration validation and role-gate tests for any setting that accepts a command-like field. And Pull-direction is a block too, because user-supplied configuration could choose which program launched; the control is to pin allowed commands and require fresh review on any config change. The decision falls out cleanly: quarantine a pre-1.83.7 surface of this kind for agent use, and let a patched deployment proceed only where the surrounding environment also enforces role checks, command allowlists, logging and review on configuration changes [6]. The worksheet at the end carries each control in full.

The same gate catches CVE-2025-54136 by the same four questions. There, an approved Cursor configuration was replaced without a renewed trust prompt [5]. Provenance must bind to the exact config content rather than the component name alone. Permissions must treat that config as executable admission. Posture must include a diff review of what changed. Pull-direction must reject silent mutation. The novelty here is not that AI requires alien security practices. It is that the thing needing those practices may be a Markdown skill file, a tool description or a JSON config rather than a compiled binary.

There is a sharper objection waiting, and it goes to whether the essay earns its keep at all. If two of the four public records are CVEs with patched versions, then for those two the entire prescription collapses into "keep your dependencies current", which every competent team already does. So where is the new discipline? Patch management is the floor here and nowhere near the ceiling, and the distance between the two is the whole point. Patching is reactive: it acts on a named flaw after a fix exists. It has nothing to say about a skill that was never buggy and was malicious by design, nothing to say about a hidden-Unicode payload that no CVE will ever be issued for, and nothing to say about the durable temptation, treating a prior approval as permanent trust, that outlives any single patched instance. None of that is speculative: any instruction-bearing component a model reads can become its operating context [11], and the upstream surfaces that feed it can be poisoned cheaply in practice [12]. The 4P Gate is not a replacement for patching but the layer that covers what patching structurally cannot: text admitted as authority, components malicious from birth, and trust that should have expired. If your supply chain were only compiled code with CVEs, you would not need the gate. It is not, and that is exactly why installation has stopped being a trust ceremony and become an authority transfer.

A harder objection than overhead is whether the gate is even falsifiable. If every bad outcome can be retro-labelled a failed P after the fact, the gate can never be wrong, and a scheme that can never be wrong is theatre. The fair test is not whether the gate has a row for every disaster in hindsight, because any four-box scheme can be drawn around a post-mortem. The test is whether, applied in writing before the click, it forces a stop the team would otherwise have skipped. On the calculator and on CVE-2025-54136 it does: Posture and Provenance respectively block at install, before any patch exists. The gate's real failure mode is a component that passes all four gates and still exfiltrates, and the honest answer is that a hidden-instruction

technique the Posture scan does not yet recognise is exactly that case. That is why Pull-direction denies the runtime channel the payload needs rather than trusting the scan alone: the layers are defence-in-depth, not circular labelling. The gate is a habit, not a certificate; the evidence carries the category, not a permanent census of how many MCP servers are deployed or vulnerable. That is precisely the argument for building it now rather than waiting for the clean number security people always want and adversaries rarely provide.

Go back to the calculator. The reason it slipped through was never that the attack was sophisticated; the addition tool added two numbers, and the approval window described it. The 4P Gate would have caught it not by being cleverer than the attacker but by insisting on reading what the model reads. Provenance asks who wrote this exact version and whether anyone reviewed the description the model sees, not the one you see. Posture asks whether a scanner read the tool description and checked it for hidden text and exfiltration patterns. Either question, asked in writing before the click, separates the calculator you approved from the instruction the model received. The gap was always there. The gate is just the habit of looking at it.

Carry This Forward

This week, on paper, do two things.

First, write your Agent Instruction BOM: list every component that can speak to your agent as instruction. Prompts, skills and their body files, MCP servers and the tool descriptions they expose, manifests, remote prompt URLs, mutable configs. If the list surprises you, that is the finding.

Second, pick one entry from that list, ideally one you installed in the last thirty days, and run it through the 4P Gate worksheet. So that two readers fill the grid the same way, here is the LiteLLM case from above transcribed into the worksheet first, then the blank template beneath it.

Worked specimen (LiteLLM, pre-1.83.7):

FIELD	YOUR ANSWER
Component and exact version	LiteLLM gateway, version < 1.83.7
Provenance	Known project, public advisory. Conditional pass: pin >= 1.83.7, reject unknown forks.
Permissions	BLOCK: launches programs on a valid access key alone, no role gate. Control: treat the connection config as executable authority and restrict program launch to a role-gated allowlist.
Posture	BLOCK until reviewed: advisory names the missing role check; needs config validation plus role-gate tests. Hidden-text check both readers re-derive verbatim: normalise the file to an NFC code-point stream and grep the zero-width and bidi ranges U+200B-200D, U+202A-202E, U+2066-2069, U+FEFF, stopping on the first hit, so two readers reach the same verdict rather than a paraphrase.
Pull-direction	BLOCK: user config chooses the launched program; pin commands, fresh review on change.
Gate verdict	Permissions blocks first

FIELD	YOUR ANSWER
If overridden	(only if admitting a patched deployment) named owner / risk tier / expiry / roll-back
Re-gate triggers	Re-gate on any config change

Blank template:

FIELD	YOUR ANSWER
Component and exact version	
Provenance	Who produced it, who reviewed the de-scription the model sees, can the version be verified?
Permissions	What can it reach on load or execution if it is compromised?
Posture	What scanner or review actually touched it, including a hidden-text check?
Pull-direction	Can it fetch new instructions, commands or config after approval?
Gate verdict	Pass, or which P blocks first
If overridden	Named owner / risk tier / expiry date / rollback plan
Re-gate triggers	Install done; re-run on update, and on any permission or pull-direction change

Answer it in writing, then ask which P blocks first and who owns that block. That ownership question is the line between a supply chain and a pile of useful things.

Hand-off: admission control is the first trust boundary; runtime control, the blast-radius question from the previous essay, is the second. A clean compon-ent can still read untrusted text during a run, so installation approval is the start of trust management, not the end, and the same owner should know which boundary failed. The next essay takes up what happens when the agent is running and the untrusted text arrives.

THE STACK SO FAR E10 · Essay 10 of 22 complete · Arc II: Evidence and authority

The Stack So Far. *Every essay adds one instrument to the operating model. The constellation shows which eight you are building, which are lit by essays you have read, and which is added right here.*

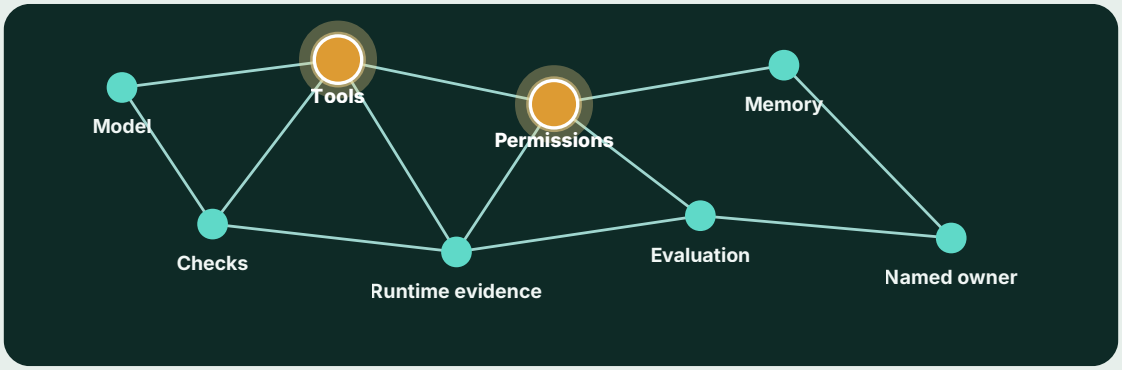
I See the object

II **Evidence and authority**
ESSAY 5 OF 5

III Runtime control

IV Proof and accountability

V Operating model



built in earlier essays

added in this essay

coming in later essays

01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22

Arc II complete. You can now ask for evidence and authority. Vendor evidence, self-report limits, governability, tool blast radius, supply chain.

You have just added.

The 4P Gate and Agent Instruction BOM

You can now inspect instruction-bearing components before they enter the agent.

Next. E11 asks how a single sentence can capture an agent's authority at runtime.

← PREVIOUS
E09 · Tools Give Models Hands

Essay 10 of 22 complete

NEXT →
E11 · The Sentence That Owns the Agent

References

Reference links for sources cited in this essay.

1. **MCP tool poisoning attacks**
   Invariant Labs
   https://invariantlabs.ai/blog/mcp-security-notification-tool-poisoning-attacks

2. **Introducing MCP-Scan**
   Invariant Labs
   https://invariantlabs.ai/blog/introducing-mcp-scan

3. **ToxicSkills: malicious AI agent skills**
   Snyk
   https://snyk.io/blog/toxicskills-malicious-ai-agent-skills-clawhub/

4. **OWASP Top 10 for Agentic Applications (2026)**
   OWASP GenAI Security Project
   https://genai.owasp.org/resource/owasp-top-10-for-agentic-applications-for-2026/

5. **CVE-2025-54136**
   NVD
   https://nvd.nist.gov/vuln/detail/CVE-2025-54136

6. **LiteLLM MCP command execution advisory**
   GitHub Advisory Database / BerriAI
   https://github.com/BerriAI/litellm/security/advisories/GHSA-v4p8-mg3p-g94g

7. **NIST AI Agent Standards Initiative**
   NIST CAISI
   https://www.nist.gov/caisi/ai-agent-standards-initiative

8. **ETSI TS 104 223 V1.1.1 (from UK DSIT code, adopted 8 May 2025)**
   ETSI
   https://www.etsi.org/deliver/etsi_ts/104200_104299/104223/01.01.01_60/ts_104223v010101p.pdf

9. **Prompt injection is not SQL injection (it may be worse)**
   UK NCSC
   https://www.ncsc.gov.uk/blog-post/prompt-injection-is-not-sql-injection

10. **PIPEDA Fair Information Principles (Principles 4.5, 4.8)**
    Office of the Privacy Commissioner of Canada
    https://www.priv.gc.ca/en/privacy-topics/privacy-laws-in-canada/the-personal-information-protection-and-electronic-documents-act-pipeda/p_principle/

11 **Not what you've signed up for: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection**
Greshake et al.
https://dl.acm.org/doi/10.1145/3605764.3623985

12 **Poisoning Web-Scale Training Datasets is Practical**
Carlini, Jagielski, Choquette-Choo, Paleka, Pearce, Anderson, Terzis, Thomas and Tramer
https://ieeexplore.ieee.org/document/10646610/

13 **AI Verify: AI governance testing framework**
Singapore IMDA / AI Verify Foundation
https://aiverifyfoundation.sg/

About the Author

ARCHITECTING THE AI COWORKER

Dr Peter McCann Strain

Dr Peter McCann Strain is a CTO, founder and senior AI engineer with a DPhil/PhD in AI from Oxford University. He builds production AI systems and writes about making agentic AI useful, inspectable, governable and safe enough for real work.

Architecting the AI Coworker · Essay 10, "The Supply Chain You Cannot See". Code-first figures, evidence-tiered references.

READ THE FULL SERIES

Substack (canonical)	petermccannstrain.substack.com
Medium	@peter.mccann.strain
LinkedIn	peter-strain-dphil-15a607128
Web	petermccannstrain.com
Cadence	New essays twice weekly, 2 June – 21 July 2026