

Workstation Hardening for AI Workflows

For devs, security, and operators · ~12 min read

Your workstation stopped being a single-purpose development box a while ago. It is now an AI-integrated platform — editor extensions that stream code to remote providers, CLI agents that can run arbitrary shell commands, MCP servers exposing tools to whichever client connects first, local model runners pulling weights from registries on a schedule, and browser extensions sitting on every tab with `<all_urls>` permission. Each of these is a feature someone asked for. Together, they are an attack surface that did not exist eighteen months ago and that most people have never fully inventoried. This lesson is about making that inventory, reasoning about the blast radius of each item on it, and then making deliberate choices to keep the footprint small.

Inventory first, harden second

The reason to start with an inventory is not bureaucracy, it is that you cannot harden what you have not enumerated. Every hardening guide in the world is useless against the extension you forgot to write down. The AI tooling problem is especially hostile to casual audits because the tools install themselves in five different places — the IDE's extension store, the shell's package manager, the browser's web store, a standalone installer, and a docker container that a teammate set up a year ago — and because the categories that matter do not map cleanly to any single package manager. A serious audit lists every AI-touching component on the machine, regardless of how it got there, and does not move on to hardening until that list is complete. Inventory is the gate, not the preamble.

The five categories

Five categories cover almost every AI-related component you are likely to find on a modern workstation. Each one has a characteristic risk profile and a characteristic mitigation.

IDE extensions. Risk: they read far more of your code than you think. Autocomplete in a modern AI extension typically sends the whole open buffer, often the nearby files, and sometimes the whole workspace index, on every trigger. The extension has full access to anything the editor has open, which usually means your private repos, your secrets files if they are open, and your scratch buffers with debugging credentials in them. Mitigation: read the extension's manifest, disable it on sensitive repos, turn off telemetry explicitly, and periodically run a network inspector against the editor to confirm where the traffic is actually going.

CLI agents. Risk: exec scope. A CLI agent with shell access is an entity that can run arbitrary commands in your environment with your permissions. Its filesystem scope is usually "cwd and below" at best and "anywhere you can cd" at worst; its exec scope is the full shell. Mitigation: run agents with explicit per-action approval (the "ask before running" mode), use filesystem-scoped project directories, review the tool permission set before enabling, and never run an agent as root.

MCP servers. Risk: aggregate permission and tool-surface sprawl. An MCP server is effectively a local daemon with a toolbelt, and the aggregate permission of the daemon is the union of every tool it exposes. A server with filesystem, shell, browser, and network tools is not "an MCP server" — it is a general-purpose RAT with a friendly protocol on top. Mitigation: prefer servers with narrow, single-purpose tool sets; audit the server source before running it; restrict filesystem mounts to the narrowest useful path; and treat it operationally like a service you run in production, because that's what it is.

Local model runners. Risk: supply chain, not prompt leakage. The inference itself is local, so your prompts don't leave the box — that is the headline win and it is real. The risk moves to other surfaces: the runner phones home for updates, sends telemetry on crashes and usage, pulls model weights from a registry that may or may not be signed, and loads those weights into a process with your permissions. Mitigation: disable auto-update where the runner supports it, pin model versions, verify weight signatures where possible, and block the runner's telemetry endpoints at the host firewall if you care about the data.

Browser extensions. Risk: the highest-footprint class by a wide margin. An AI browser extension with `<all_urls>` permission — and most of them request it — can read every tab, every form, every page. It sees your bank session, your webmail, your internal admin panels, your password manager's autofill, every prompt you type into

every other AI. Extension updates ship silently through the browser's store, so yesterday's reputable publisher can deliver today's compromised payload without user interaction. Mitigation: use a dedicated browser profile for AI extensions, keep your high-sensitivity browsing in a profile with no extensions at all, restrict site access to specific hosts instead of all URLs when the extension supports it, and audit the extension's publisher and permission set before install.

The blast-radius table

Once every component is on the list, the next step is reasoning about what each one can do. The cleanest way to do this is a table with seven columns. Use the same columns for every tool, in the same order, so the table becomes a structured comparison instead of a wall of prose.

- **Tool** — name and version. "Cursor 0.42" not "my IDE."
- **Reads** — what data the tool can read. Buffer, repo, home directory, clipboard, every tab, etc.
- **Writes** — what data the tool can modify or create. Files in workspace, model cache, config, etc.
- **Net** — what network endpoints the tool talks to. Provider API, telemetry endpoint, update server, none.
- **Exec** — whether the tool can run commands or spawn processes. Yes / no / yes-with-approval.
- **Telemetry** — whether the tool sends usage data, and whether that is on or off by default.
- **Verdict** — your decision. Keep, restrict, or remove.

A single filled row makes the pattern concrete:

Tool	Reads	Writes	Net	Exec	Telemetry	Verdict
Cursor (IDE)	Whole repo buffers	Files in open workspace	AI provider APIs	Yes via agent mode	On by default	Restrict

The value of the table is not the individual rows — it is that the rows force you to answer seven specific questions about every tool, and if any cell is blank you have not audited the tool. "I don't know" is a valid cell content and an immediate action item: find out before next review.

Choosing the smallest footprint

With a full inventory and a filled-in blast-radius table, the hardening decisions almost make themselves. Prefer tools that run locally over tools that send prompts to a provider, because local tools do not create a third-party log of your prompts. Prefer tools that require explicit per-action approval over tools that act silently and autonomously, because the approval prompt is a chance to notice that the next step is wrong. Prefer tools with an inspectable manifest — one you can read and understand in five minutes — over tools whose behavior is a black box. Prefer tools that ship with telemetry off by default, because "off by default" reflects the publisher's attitude toward your data and that attitude tends to be load-bearing over the life of the tool. When two tools offer the same feature and differ on these axes, the smaller-footprint tool is almost always the right pick, even when it is slightly less convenient. The cost of a convenience delta is measured in seconds per use; the cost of a footprint delta is measured in incidents.

Update channels are attack surface

The last category worth thinking hard about is the update channel. Every tool on your workstation — every IDE, every extension, every CLI, every runner — has some path by which new code gets onto the machine. That path is a supply-chain surface whether you treat it as one or not. The convenient path is auto-update: the tool phones home, sees a new version, fetches it, and installs it, all without asking you. The inconvenient path is pinning: you specify a version, you verify it, and new code only gets onto the machine when you explicitly approve it.

Auto-update is "someone else decides what code runs on your machine tonight."
Treat it like remote code execution, because that's what it is.

The trade-off is real and the right answer depends on the tool. For tools that ship frequent security fixes — browsers, OS packages — auto-update is usually worth the risk because the alternative is accumulating known-unpatched vulnerabilities. For AI tools specifically, the calculus is different: the tool's feature velocity is high, the publisher's operational maturity varies, and the blast radius of a compromised update is the full set of permissions the tool already has. Pinning and reviewing is the right default for the AI category. Where pinning is not supported, at minimum, subscribe to the publisher's release channel and read the changelog before the update lands.

Rules to remember

- You cannot harden what you haven't inventoried. List every AI-related component first.
- Every tool has a blast radius: reads, writes, network, exec, telemetry. Know it before you install.
- Prefer explicit-approval per action over silent autonomy.
- Auto-update is a supply-chain surface, not a feature. Pin versions where you can.
- Browser extensions are the highest-footprint class. Assume they see everything in every tab.