

From Demo to Production

The AI Agent Hardening Guide

by Soft Tech Solution

Every agent demo works. Production is a different discipline.

*The six hardening layers between a flashy prototype
and an agent a business can actually trust.*

1. The Gap - Demos Impress, Production Decides

Every AI agent demo works. That is precisely the problem. In a demo, the data is clean, the task is scripted, the tools behave, and a human is standing by to smile through the rough edges. The agent looks flawless - because the demo was engineered so it cannot fail.

Production is a different discipline. The inputs are **messy**: customers paste half a conversation, upload the wrong file, ask for three things in one message. The edge cases were **never scripted**: the refund that is also a partial exchange, the booking that lands on a public holiday, the supplier whose invoice format changed last week. And when the agent gets it wrong, it usually gets it wrong **silently** - no crash, no alarm, just a wrong answer sent, a wrong record updated, a wrong amount moved, discovered days later by a customer instead of by you.

Here is what surprises most teams: the model is rarely the weak point. The same model that dazzles in the demo fails in production because of everything **around** the model - what it is allowed to touch, whether anyone approves the risky steps, whether anyone can see what it did, whether anyone tested it after the last change, what happens when a tool errors, and how much it is allowed to spend while nobody is watching.

That surrounding engineering is called hardening, and it is a learnable, repeatable system. This guide lays out the six layers applied to every production agent build: guardrails, approval gates, observability, evals, fallbacks and recovery, and cost controls. None of them require a research lab. All of them are the difference between an agent you demo and an agent you deploy.

Read it as an engineer: each layer gets a plain-English explanation, the concrete mechanics, and the failure it prevents. At the end you get a step-by-step roadmap to a hardened agent skeleton, plus the 7-point pre-launch checklist used before any agent touches a live business.

2. The Six Hardening Layers (1-2)

Think of the six layers as a stack: each one assumes the others exist. Guardrails decide what the agent **can** do; approval gates decide what it may do **alone**; observability records what it **did**; evals prove it still works **after every change**; fallbacks decide what happens **when it fails**; cost controls cap what failure **can cost you**. Layers 1 and 2 are the perimeter - everything else operates inside them.

Layer 1 - Guardrails: Scoped Permissions and Sandboxed Tools

The first hardening question is never "how smart is the agent?" It is "what can the agent touch?" A production agent gets the **minimum access its one job requires** - and nothing else exists for it:

- **Scoped permissions.** The agent's credentials can read the order system and update a status field - they cannot delete records, export the customer list, or touch billing. If a capability is not needed for the job, it is not granted.
- **Allowlists, not blocklists.** The agent may call these five tools, contact these domains, spend in these categories. Anything not explicitly allowed is denied by default. Blocklists fail the first time the model gets creative; allowlists fail safe.
- **Sandboxed tools.** Code execution, file handling and browsing run in an isolated environment with no route to your production database, your email, or the open internet beyond the allowlist. A prompt injection in a customer message then has nowhere to go.
- **Least-privilege API keys.** Separate keys per agent and per environment, read-only wherever reading is enough, rotated on a schedule, stored in a secrets manager - never in the prompt, never in the code. The **OWASP Top 10 for LLM Applications** (owasp.org) calls out excessive agency and insecure plugin design for a reason: most real-world agent incidents are access incidents, not intelligence incidents.

Implementation is boring on purpose: permission scopes in the API console, a tool registry in code that lists exactly what the agent may invoke, and a deny-by-default middleware in front of every tool call. Frameworks like **Guardrails AI** (guardrailsai.com) and **NVIDIA NeMo Guardrails** (github.com/NVIDIA/NeMo-Guardrails) add input/output validation on top; the perimeter itself is yours to set.

Layer 2 - Approval Gates: Humans Approve What Money Cannot Undo

Some actions are reversible - drafting a reply, summarising a ticket. Some are not: sending money, issuing a refund, deleting data, emailing a customer a commitment. Hardening rule: **an agent never executes an irreversible or money-moving action autonomously**. It prepares the action completely - recipient, amount, reason, draft - and then it stops and asks.

- **The one-tap pattern.** The owner gets one message: what the agent wants to do, why, and the exact amount or text. Approve or reject with a single tap or a YES reply. No dashboard login, no digging - the decision takes five seconds, which is why it actually happens.
- **Tiered autonomy.** Low-risk actions (tagging, filing, internal notes) run autonomously. Medium-risk actions run but are logged and reviewable. Money, refunds, external commitments and deletions always wait for the gate. The tiers are written down before launch, not improvised after an incident.
- **Single-use instruments.** Where an agent must pay, it never holds a reusable card. Payment rails like **Stripe** (stripe.com) and Stripe Issuing can create a **single-use virtual card capped at the exact approved amount** - the agent spends that and nothing else, and the credential dies with the

transaction.

This is also the design the industry is converging on: the major agent platforms now ship approval checkpoints for sensitive actions as a headline feature. The gate is not a lack of trust in the model - it is the mechanism that makes delegating to the model rational.

3. The Six Hardening Layers (3-4)

Layer 3 - Observability: If It Is Not Logged, It Did Not Happen Safely

An agent you cannot audit is an employee who keeps no records and never reports. Observability means every run leaves evidence:

- **Immutable activity logs.** Every action appended to a log the agent itself cannot edit or delete: timestamp, input received, tools called, data changed, output sent. If it is not in the log, support's answer is "we don't know" - which is never an acceptable answer about your own business.
- **Per-run traces.** Beyond the final answer, capture the steps: which prompt version ran, what the model saw, which tool returned what, where the tokens and time went. Tracing tools built for this - **LangSmith** (langsmith.com), **Langfuse** (langfuse.com) - turn a mysterious wrong answer into a five-minute diagnosis.
- **Off-script alerting.** Define the shapes of a bad run in advance - a tool called with unusual parameters, a spend above a threshold, a reply containing words the agent should never send - and alert a human the moment one fires. Detection in minutes beats discovery by customer complaint in days.

The working rule from production support: **when something goes wrong, the log should answer what happened, in what order, with what data, in under five minutes.** If it cannot, observability is the layer you are missing, and every other incident will be a guessing game.

Layer 4 - Evals: Prove It Still Works - After Every Change

Agents regress silently. You improve the prompt for one case and quietly break two others. You swap a model version and the tone, the format and the edge-case behaviour all shift at once. Without evals, you find out from customers. With evals, you find out in the test run - before deploy.

- **A fixed test set.** 30-100 real cases drawn from actual tickets, orders and messages - including the ugly edge cases - each with the expected behaviour written down. This set never changes casually; it is the contract the agent is measured against.
- **A target score.** Not vibes - a number. "The agent must pass at least 95% of the eval set, and 100% of the safety cases, before any change ships." A change that drops the score does not deploy, however good it looks on the one case it was fixing.
- **Run before every change.** Prompt edit, model swap, new tool, new data source - the eval suite runs first, automatically. Tooling ranges from simple scripts to **promptfoo** (promptfoo.dev) and the eval features inside LangSmith; the tool matters less than the discipline of never skipping the run.

Keep a small **golden set of disasters** inside the suite: the refund that must never auto-approve, the message that must always escalate, the injection attempt that must always fail. If those pass, the agent's worst day is bounded. If they are not tested, they are hopes, not properties.

4. The Six Hardening Layers (5-6)

Layer 5 - Fallbacks and Recovery: Plan the Failure Before It Happens

Production tools fail: APIs time out, providers return errors, files arrive corrupted, models produce unparseable output. A hardened agent treats failure as a scheduled event, not a surprise:

- **Retries with backoff.** Transient errors retry a small number of times with increasing waits - then stop. Infinite retry loops are how a small outage becomes a large bill and a flooded inbox.
- **Graceful degradation.** When a capability fails, the agent sheds load instead of collapsing: it can still answer from the knowledge base when the order system is down, and it says so honestly instead of inventing an order status.
- **Human handoff.** Defined triggers - low confidence, angry customer, out-of-policy request, repeated failure - route the conversation to a human with the full context attached. The handoff is a feature users trust, not an admission of defeat.
- **Undo and rollback.** Where the platform supports it, actions are staged so they can be reversed: drafts before sends, soft-deletes before hard deletes, versioned records before overwrites. Where undo does not exist, the approval gate (Layer 2) stands in its place.

Write the failure table before launch: for each tool the agent uses, what does failure look like, what does the agent do, what does the customer see, and who gets told. Twenty minutes of table-writing replaces the worst meeting you will otherwise have this year.

Layer 6 - Cost Controls: Cap What a Bad Day Can Cost

An autonomous agent with an uncapped API key and a payment method is a financial exposure measured in hours, not cents. Cost hardening is simple, mechanical, and non-negotiable:

- **Per-run ceilings.** Every run has a maximum token budget, a maximum number of tool calls and a maximum wall-clock time. A run that hits its ceiling stops and reports - it never quietly loops.
- **Per-day ceilings.** Total spend across all runs is capped per day, with an alert at 50% and a hard stop at 100%. The cap is set from the business case, not from optimism.
- **Single-use payment instruments.** As in Layer 2: purchases happen on virtual cards capped at the exact approved amount, issued per transaction. There is no standing balance for an agent to drain.
- **A kill switch that actually kills.** One control - a flag the runtime checks on every step - that halts the agent mid-run, revokes its session, and pages a human. Test it. A kill switch that has never been tested is a decoration.

Together, Layers 1-6 convert the risk profile of an agent from **unbounded and invisible** to **bounded and audited** - which is the entire difference between a prototype and a production system.

5. Why It Matters - The Business Impact

Hardening is sometimes dismissed as gold-plating - process for its own sake. It is not. Each layer maps to money that either stays in the business or leaves it. These figures are **illustrative** - run them with your own numbers - but the shape of the arithmetic does not change:

The unhardened failure	What it typically costs
One unsupervised refund/purchase at the wrong amount	\$200-\$2,000+ per incident
The approval gate that would have caught it	~5 seconds of owner time
Silent failure discovered by a customer, not the log	Days of wrong data + the trust repair
A logging habit that surfaces it in minutes	Near-zero marginal cost per run
One prompt change that quietly regresses the agent	Weeks of degraded answers before anyone notices
An eval suite that catches it pre-deploy	One automated test run

Two honest observations from these numbers. First, **the expensive failures are rarely dramatic** - no single catastrophic event, just a wrong refund here, three days of silently wrong status answers there, a month of slightly worse replies quietly costing repeat business. Hardening is priced against the drip, not the flood. Second, **the layers are cheap relative to what they bound**: permission scopes are console settings, logs are a database table, an eval set is a spreadsheet of real cases, a kill switch is a flag. The costly part of production AI was never the hardening - it is operating without it.

There is also a quieter return: **trust compounds**. A business owner who can see exactly what the agent did, who approved what, and what it spent, delegates more over time - which is where the real productivity of agents lives. An owner burned once by an opaque failure delegates nothing ever again, and no model upgrade fixes that.

6. The Build Roadmap - A Hardened Agent Skeleton

This roadmap builds a minimum hardened agent: one job, all six layers present from the first commit. The stack is deliberately mainstream - swap any piece for your preferred equivalent and the layers still apply.

Step 1 - Define one job and its permission scope

Write down the agent's single job in one sentence ("answer order-status questions and escalate everything else"), the exact list of systems it may touch, and the tier of every action it can take (autonomous / logged / approval-gated). This document becomes your Layer 1 and Layer 2 configuration. If you cannot write it, you are not ready to build - and neither is anyone you hire.

Step 2 - Scaffold the agent with a vibe-coding tool

Start from a working skeleton instead of a blank file. Generate the base app in **v0** (v0.dev), **Lovable** (lovable.dev), **Bolt** (bolt.new) or **Replit** (replit.com) using the master prompt below - it bakes the six layers into the skeleton instead of bolting them on later:

COPY-PASTE PROMPT - the hardened skeleton:

```
Build a single-job AI support agent for a small business.

JOB: {the one job, one sentence}
MODEL: Claude API or OpenAI API (key in environment variables only)

HARDENING REQUIREMENTS - implement all six:
1. GUARDRAILS: a tool registry listing every tool the agent may call.
   Deny by default. Read-only tools unless marked otherwise.
   API keys scoped least-privilege, loaded from env, never logged.
2. APPROVAL GATES: actions tagged MONEY or IRREVERSIBLE never execute
   directly. Stage them, show owner a summary + amount, execute only
   after an explicit approve action.
3. OBSERVABILITY: append-only run log (timestamp, input, tool calls,
   outputs, cost) + a per-run trace view in an admin page.
4. EVALS: an evals/ folder with a test set in JSON and a runner script
   that scores the agent and fails the build below target score.
5. FALLBACKS: retries with backoff (max 3), graceful degradation when a
   tool is down, and a human-handoff action with full context.
6. COST CONTROLS: per-run token/tool-call caps, a per-day spend cap in
   config, and a KILL_SWITCH flag checked before every step.

DELIVERABLE: the agent, the admin trace page, the eval runner, and a
README listing every permission granted and why.
```

Read what the tool generates against the six requirements line by line - vibe coding accelerates the build, it does not transfer the responsibility. Anything missing gets a follow-up prompt naming the exact layer and requirement.

Step 3 - Wire the model and tools properly

Connect the **Claude API** (console.anthropic.com; docs at docs.claude.com) or the **OpenAI API** (platform.openai.com) with function/tool calling. Issue **separate API keys per environment** (dev/production), set provider-side spend limits as a second ceiling, and put every business tool behind your deny-by-default registry from Step 2 - the model proposes tool calls, your code disposes.

Step 4 - Add observability and the approval channel

Point the run log at a real store (a Postgres table, or a tracing tool like **LangSmith** or **Langfuse**) and build the approval channel where the owner actually lives - for most small businesses that is a **WhatsApp** message via the WhatsApp Cloud API (developers.facebook.com) or **Twilio** (twilio.com), with Approve / Reject actions wired to the staged action from Step 2. Test the full loop with a dummy money action before any real one exists.

Step 5 - Build the eval set from real cases

Pull 30-100 real interactions - tickets, chats, orders - anonymise them, and write the expected behaviour next to each. Include your disaster cases explicitly. Wire the runner (a script, or **promptfoo** at promptfoo.dev) into your deploy process so **no change ships on a failing score**. First run will be humbling; that is the point - better humbled in the test report than in the customer inbox.

Step 6 - Deploy behind the ceilings, then loosen slowly

Deploy on **Vercel** (vercel.com) or **Netlify** (netlify.com) for the app, with the agent worker on the same platform or a small always-on host. Launch with tight ceilings and most actions approval-gated. Over the first month, promote specific action types from gated to autonomous **only** as their logged track record earns it - autonomy is granted by evidence, tier by tier. That graduated trust is what production hardening actually looks like in practice.

ADVANCE - when the skeleton is boringly reliable

- **Prompt/version registry.** Every prompt and model change versioned, with the eval score that approved it - so any regression has a return address.
- **Weekly failure review.** Thirty minutes reading the week's fallbacks, handoffs and near-misses; each one becomes a new eval case. The suite grows teeth over time.
- **Second agent, same skeleton.** New jobs clone the hardened skeleton - layers included - instead of starting from a fresh demo. Hardening is a template, not a project.

7. The Pre-Launch Checklist

Seven checks. An agent that cannot tick every box is a demo with a deploy button - keep it away from live customers and real money until it can:

- ☐ **Scoped permissions:** the agent's credentials grant the minimum its job needs - read-only by default, nothing deletable or exportable without cause, keys per environment
- ☐ **Approval gates:** every money-moving or irreversible action stages for a human approve/reject - tested end-to-end with a dummy transaction
- ☐ **Immutable logs:** every run is recorded in an append-only log with inputs, tool calls, outputs and cost - and someone has actually read one
- ☐ **Eval set:** 30+ real cases (including the disaster cases) with a written target score; the suite runs before every change and blocks failing deploys
- ☐ **Human fallback:** handoff triggers defined (low confidence, anger, out-of-policy, repeated failure) and the handoff carries full context to a real person
- ☐ **Cost ceiling:** per-run and per-day caps configured with alerts, provider-side limits set as a backstop, payments only via capped single-use instruments
- ☐ **Kill switch:** one control halts the agent immediately - and it has been tested live, not just implemented

8. What's Next

You now have the full picture: the gap between demo and production is not intelligence, it is engineering - six layers of it, each one concrete, testable and cheap relative to the failure it bounds. Build the skeleton, run the evals, gate the money, log everything, cap the spend, test the kill switch. Then let the agent earn autonomy the way every good system does: with a track record.

Want it built for you instead? This is what we do at Soft Tech Solution: custom AI agents, apps and websites for businesses - support agents, ordering and booking systems, workflow automation - built production-hardened from the first commit, wired into the tools you already run, with your approval gates and your rules. The free guide above is the full recipe either way: harden it yourself, or hand the build to someone who does it every week.