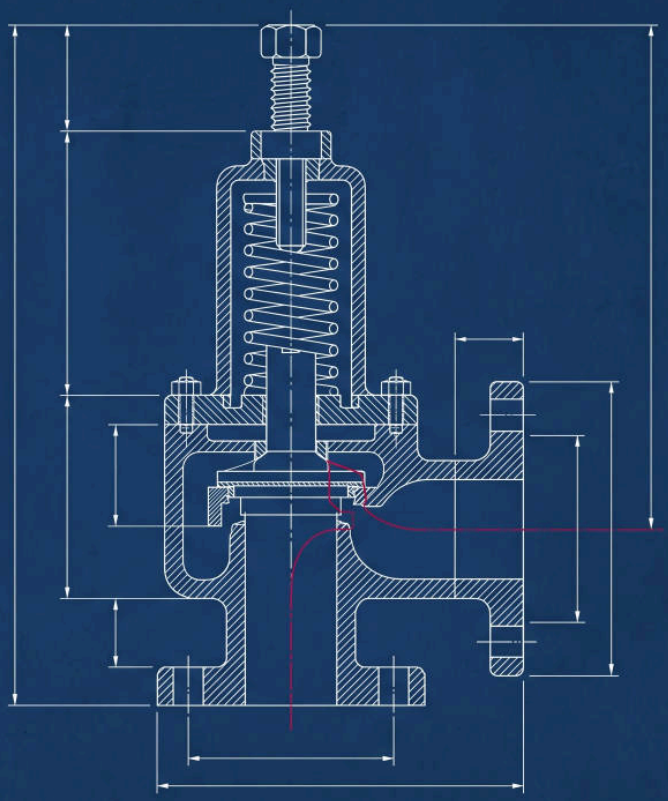

CHAPTER FIFTEEN

Safety, Guardrails and Human-in-the-Loop

SECTIONS 15.1 TO 15.10

Evaluation tells you what the system does. This chapter is about what it is permitted to do. You cannot control what the model reads, only what its output is allowed to cause.

CHAPTER FIFTEEN

Contents

PART IV · PRODUCTION

Evaluation tells you what the system does. This chapter is about what it is permitted to do. Section 5.9 stated the threat model as an argument; here it becomes enforced controls, and the organising claim is that you cannot control what the model reads, only what its output is allowed to cause.

15.1 The Output Policy

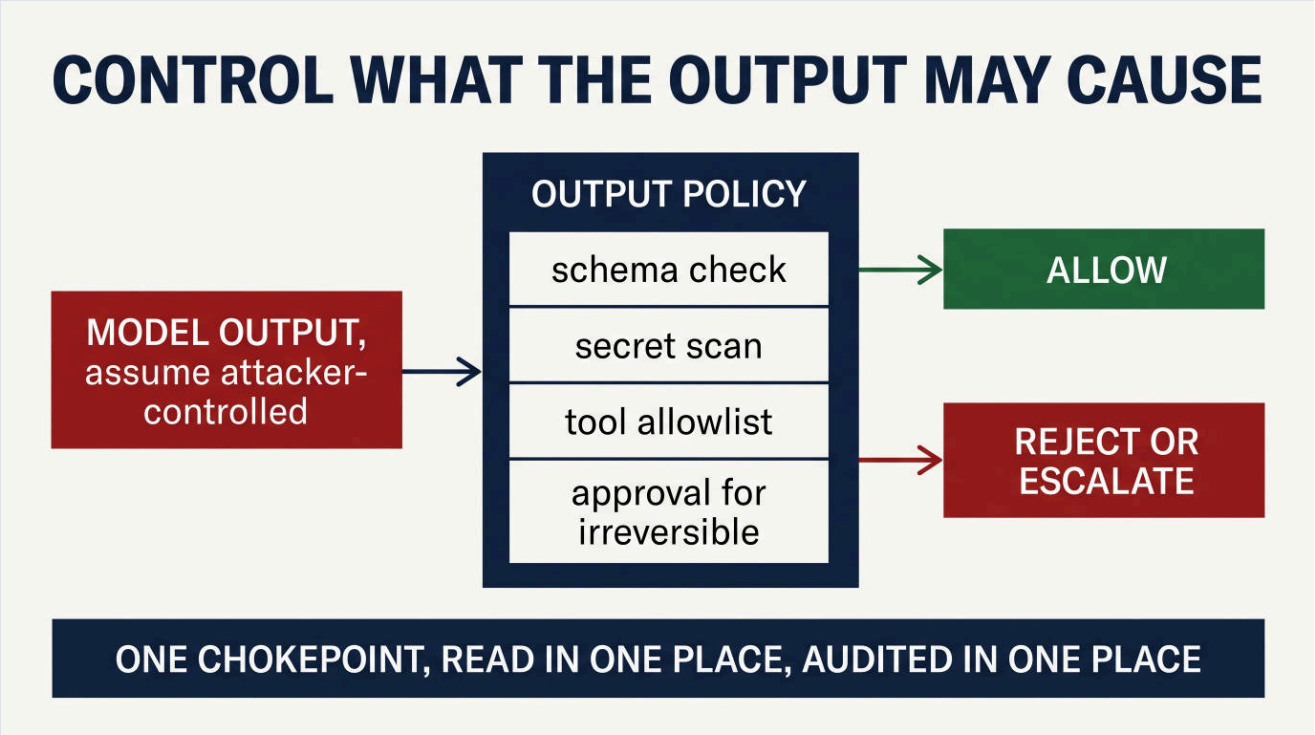

Figure 15.1 One chokepoint, four checks, read and audited in one place.

Every control in this chapter converges on a single object, and section 5.9 sketched it before there was anything to protect.

```
class OutputPolicy:
    """Applied to the model's output regardless of what the model 'decided'.

    The test: assume the model is fully compromised and emits the worst
    possible output. What still stops the damage? That is your real posture.
    """

    def check(self, out: BaseModel, ctx: Context) -> Verdict:
        if not isinstance(out, ctx.expected_schema):        # structural
            return Verdict.reject("schema")
        if leaked := self.secrets.scan(out):                # never, ever
            return Verdict.reject("secret material", leaked)
        if breach := self.content.scan(out, ctx.scoped_topics):  # harm in the prose
            return Verdict.reject("content out of scope", breach)
        for call in getattr(out, "tool_calls", []):
            if call.name not in ctx.allowed_tools:          # allowlist
                return Verdict.reject("tool not permitted for this role")
            if TOOLS[call.name].risk is Risk.IRREVERSIBLE and not ctx.approved:
                return Verdict.require_approval(call)
        return Verdict.allow(out)
```

The property that makes it a control rather than a convention is that there is exactly one of it. A policy applied in three places is a policy with three versions and one gap. Chapter 1's seams exist so this object has somewhere to live; Chapter 9 put it at the loop's chokepoint; Chapter 10 gave it per-node granularity; Chapter 12 extended it to tool results. This section is where those become one enforced path.

Two properties worth stating explicitly. **It runs after generation and before any effect**, which is the only position from which it can be complete. And **every verdict is recorded**, because section 15.8's audit trail is built from exactly these decisions and cannot be reconstructed later.

Three of the four checks are security-shaped: they bound what the output can cause in your systems. The content check is the fourth dimension, and it bounds what the text can cause in the user. A model that explains how to defeat a safety interlock, offers medical or legal advice beyond the product's scope, or asserts something damaging about a real person has produced a harm no schema check or allowlist sees, because the damage is in the prose itself.

The mechanism is the same chokepoint, not a new one. The content check runs on the outbound text against a scoped-topics list that product owns: what this product is permitted to advise on, and what it must not. That list is a versioned artefact like section 5.8's prompts, because "may we discuss dosages" is not an engineering decision, and its changes deserve the same gate.

High-stakes domains do not need a cleverer check; they need a different tier. Route medical, legal and financial questions to the REDUCED or EXTRACTIVE tier of section 15.3, where the system quotes its sources instead of advising. Quoting a document the user already holds is a materially smaller surface than composing advice from it.

Provider-side moderation endpoints exist and are worth wiring in as a signal, in the same spirit as section 15.2's classifiers. The gate is still yours: a provider's threshold encodes their risk posture averaged across every customer, and yours is specific to what this product does and for whom.

On the secret scanner, be clear what it is: pattern rules for known credential shapes, an entropy screen for strings that look generated, and canary tokens planted in the secret store so that exfiltration is detectable the moment one appears in an output. Its weakness is the false negative, because a paraphrased secret matches no pattern. That is why the real control is chapter 1's `os.environ.pop`, the model never reading secrets at all, and the scanner is defence in depth behind it.

15.2 Input-Side Controls and Their Limits

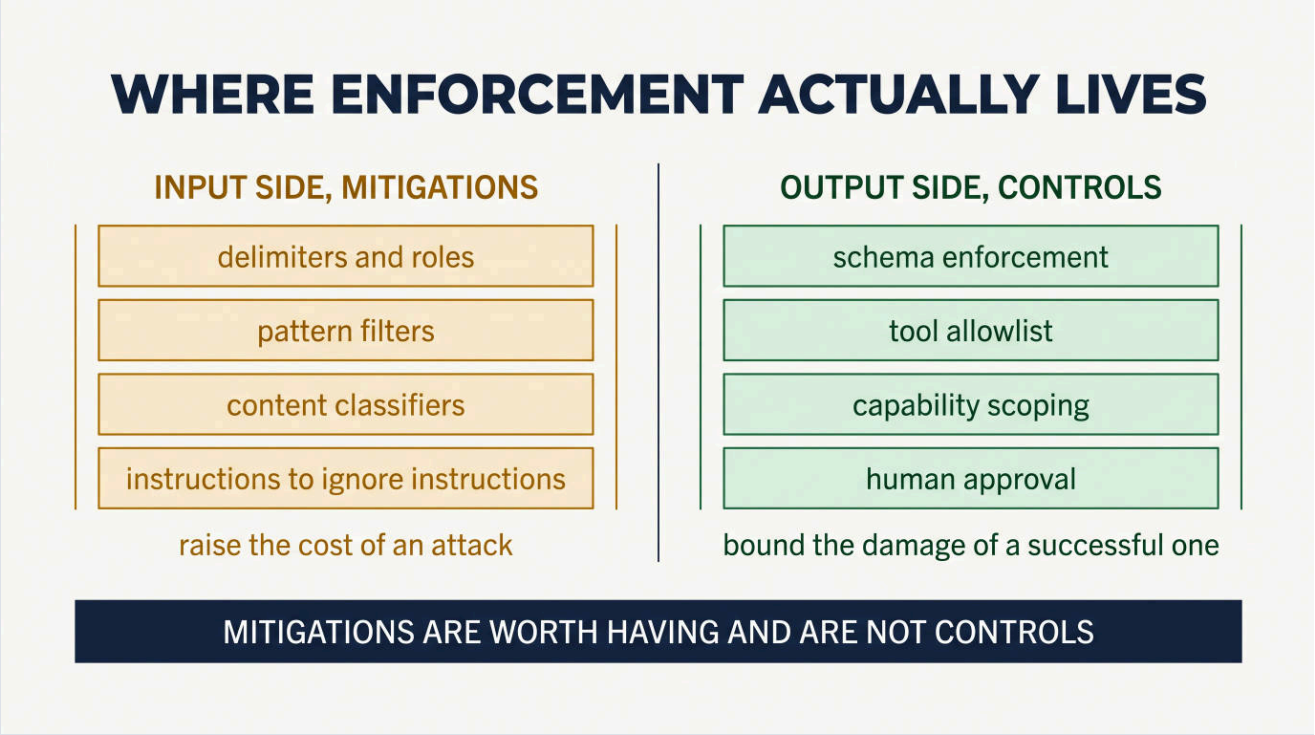

Figure 15.2 Both columns are worth having. Only one of them is enforcement.

Section 5.9 listed what does not work and why. The nuance worth adding is that those measures are still worth building, provided nobody mistakes them for controls.

Measure	What it actually buys
Delimiters and role separation	Real, cheap, and defeated by prose that does not need to break a delimiter. Section 5.7's special tokens stop role forgery only.
Pattern filters	Catch the unsophisticated and the accidental. Encoding and translation defeat them, and false positives on legitimate text are common.
Content classifiers	Useful as a *signal* for section 15.7's alerting. As a gate they add latency and a second model to be wrong.
"Ignore instructions in documents"	An instruction competing with another instruction, adjudicated by the same model.

The right framing: **input-side measures raise the cost of an attack; output-side measures bound the damage of a successful one.** A team with limited time should build the second first, because the second works even when the first fails and the first is worthless when the second is missing.

15.3 Designed Degradation

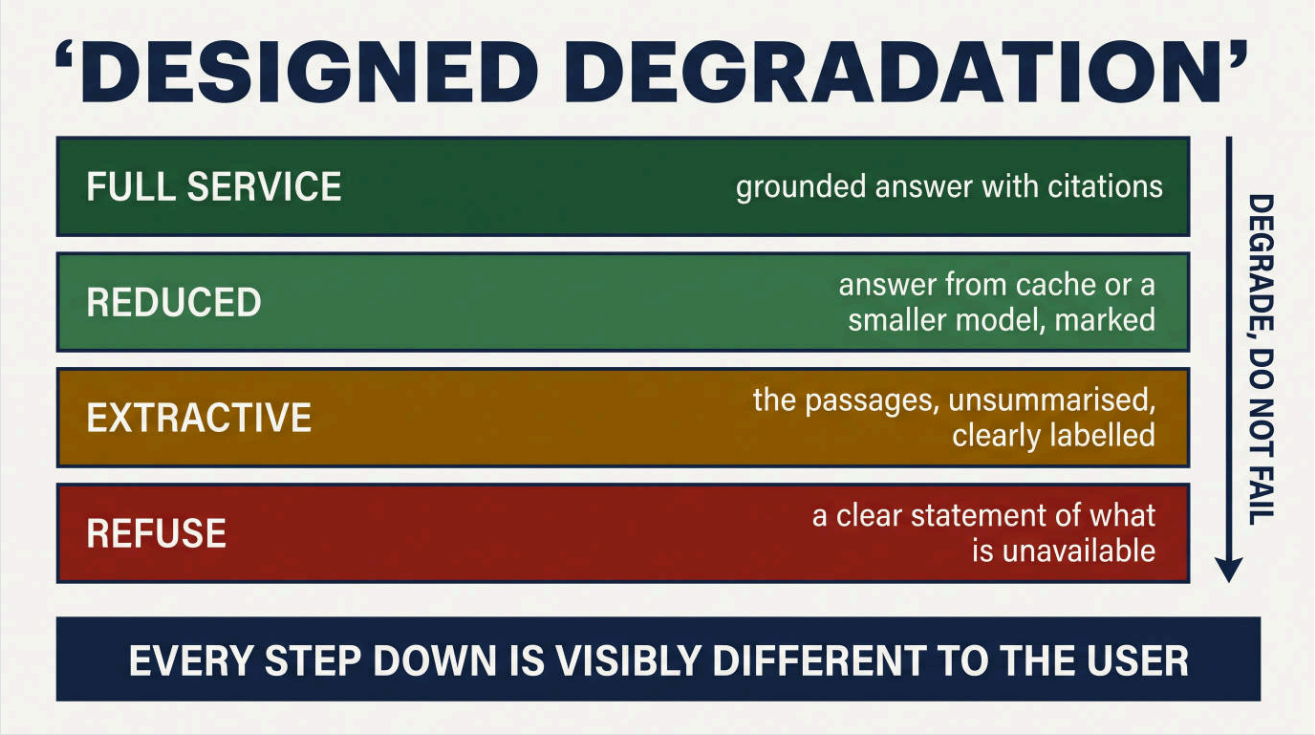

Figure 15.3 Four states, each visibly different to the user.

Section 3.6 built a degraded extractive answer for a provider outage. Section 6.9 built abstention for weak evidence. This section makes them one ladder, because a system with two failure states and one success state will improvise a third under pressure.

The rule that keeps the ladder honest is the one from section 3.6: **a degraded answer must be visibly degraded.** Silently returning a worse answer trains users to trust output they should not, which is a larger harm than the outage that caused it.

```
# The tier is a first-class field on the response, not an implementation
# detail, because the UI and the eval both need it (14.5's abstention rate).
@dataclass(frozen=True)
class Answer:
    text: str | None
    tier: ServiceTier          # FULL | REDUCED | EXTRACTIVE | REFUSED
    citations: tuple[str, ...] = ()
    reason: str = ""           # why this tier, in words a user can read
```

Two design points that are easy to miss. **Degradation must be a decision, not an accident**: if the system drops to a smaller model because the budget ledger refused, the user should be told the answer is from a reduced service. And **each tier needs its own eval**, because a degraded path nobody tests is a path that will be broken on the day it is needed. The ladder is also where section 15.1's content dimension lands: a question in a high-stakes domain drops to REDUCED or EXTRACTIVE by policy, and the tier field records that the system chose to quote rather than advise.

15.4 PII, Retention and Redaction

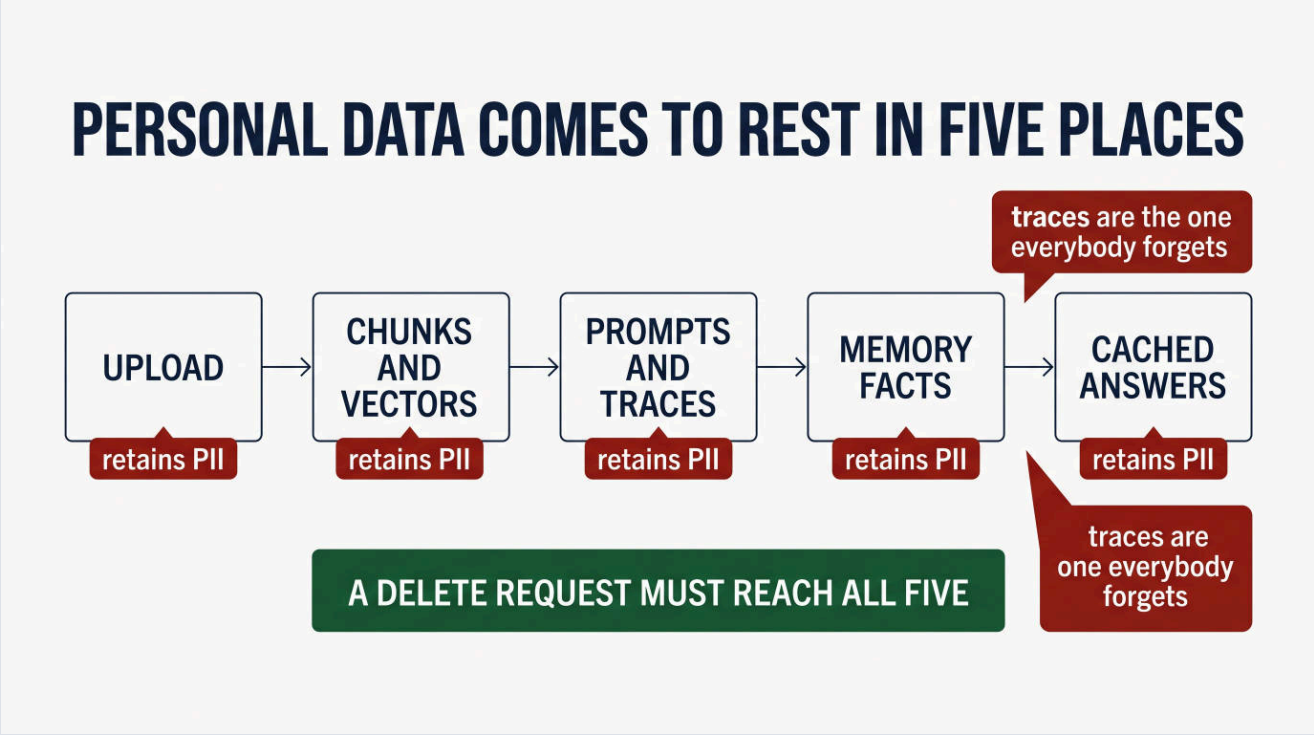

Figure 15.4 A deletion request must reach all five, and traces are the one everybody forgets.

Section 6.10 made deletion real for documents and section 11.8 for memory. The complete picture has more surfaces than either chapter needed on its own.

Surface	What deletion requires
Source documents	Object storage plus every derived artefact.
Chunks and vectors	Section 6.4's `doc_id` is what makes this a query rather than a scan.
Prompts and traces	The forgotten one. Section 1.6's trace records contain the retrieved text, which contains the personal data.
Memory facts	Section 11.8, including superseded ancestors and their embeddings.
Cached answers	Section 3.6's namespace is what makes targeted invalidation possible at all.

TRACES ARE WHERE GOOD OBSERVABILITY BECOMES A LIABILITY

Chapter 1 argued for recording the inputs that determined the output, and that argument stands. It also means your trace store holds document text, user questions and model outputs, at high volume and often with a longer retention than the primary store.

Two mitigations. Redact at write time for known-sensitive fields rather than at read time, because a redaction you apply on query does not survive an export. And keep trace retention shorter than you instinctively want: thirty days answers nearly every debugging question and reduces the surface by an order of magnitude.

On redaction generally: it is a *reduction*, not a removal. A redacted transcript with the name replaced still contains the case details that identify the person. Treat redaction as reducing casual exposure, and retention limits as the control that actually bounds the risk.

Data leakage: every path out

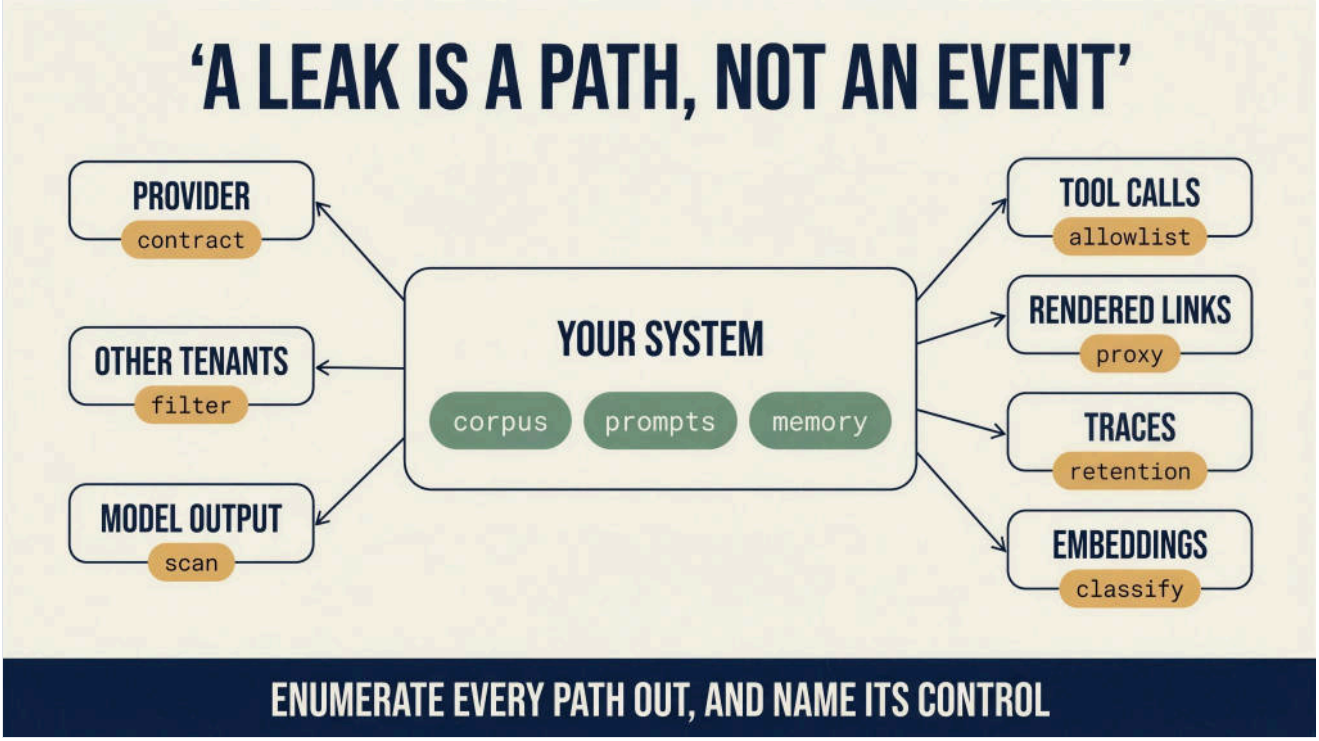

Figure 15.5 Seven paths out of an enterprise AI system, and the control that closes each one.

The five surfaces above are about data at rest. Leakage is the complementary problem: data in motion, crossing a boundary it should not, from one tenant to another, from a user to a colleague without the clearance, from your system to a provider, or from your system to an attacker. It is the question every enterprise deployment gets asked first, and the honest answer is not a product or a filter. It is the same discipline the deletion table applies: **enumerate the paths out, name the control on each, and treat any path you have not enumerated as open.** Seven paths cover an enterprise AI system of this book's shape.

The provider. Every prompt is a disclosure: retrieved documents, the user's question, memory, all of it leaves for someone else's infrastructure on every call. The control is contractual and architectural at once. Contractually, the questions of section 16.5: training-use clauses, retention at the provider, subprocessors, and the enterprise tier that answers them in writing. Architecturally, minimisation: the prompt carries the evidence this request needs, never the corpus, which the budget discipline of section 2.10 has been quietly enforcing all along.

Another tenant. The index, the semantic cache and memory are the three stores that two tenants can share by accident. Each already has its control: section 6.10's query-time assertion for the index, section 3.6's tenant-namespaced cache key, section 11.8's scoped memory subjects. The audit question is mechanical: list every store two tenants share, and name the identity-derived filter on each read path. A store missing from the list is the finding.

The model's output. Whatever the model read, it can repeat to anyone able to ask. Section 15.1's secret scan and the content dimension gate single answers; section 15.6's breadth-per-user baseline catches the slow version, a legitimate account walking the corpus one polite question at a time.

A tool call. The channel injection actually uses: a hostile instruction does not need to print the secret, it needs one tool whose argument can carry it out, which is section 15.7's exfiltration signal. The controls are chapter 12's: capabilities scoped per run and per tenant, an egress allowlist around anything that executes code, and section 9.7's chokepoint logging denied calls, because a denied exfiltration attempt is the cheapest detection you will ever get.

Rendered output. The tool-free variant: a markdown image or link whose URL encodes data exfiltrates the moment a client renders it, with no action from the user at all. The output policy is the place to close it, since it already reads every outbound answer: strip remote fetches from rendered output, or route them through a proxy that logs and allowlists.

Traces. The callout above, read as an exit rather than a store: an observability stack readable by the whole engineering organisation is a leak with dashboards. Trace access is scoped like production data access, redaction happens at write time, and retention does the bounding that redaction cannot.

Embeddings. Vectors are not anonymisation. Enough of the source text is recoverable from an embedding for the vector store to inherit the classification of the documents it indexes, which is why it sits in the deletion table above and in section 16.5's residency footprint, not outside them.

What makes this enterprise rather than merely multi-tenant is that the boundaries multiply: user, team, tenant, provider, jurisdiction. The two questions per path stay fixed, who can read what this carries, and where is that decision recorded, which is section 15.8's audit trail doing its second job. And one path runs outside your system entirely: the corpus pasted into a consumer chatbot because the governed tool was not good enough or not there. Policy can forbid that; only a governed system people prefer actually stops it, which is as good a one-line justification for this book's whole programme as any.

15.5 Refusal and Abstention as Product

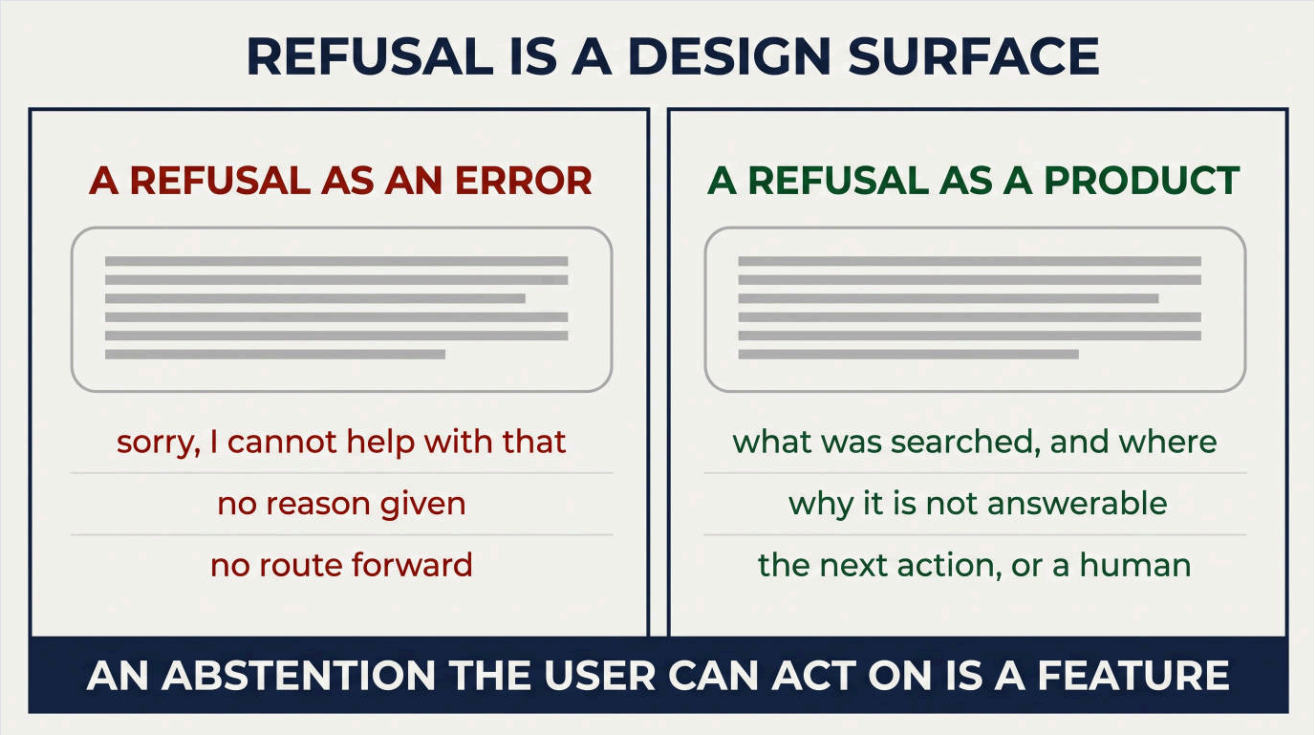

Figure 15.6 The same decision, rendered two ways. Only one of them leaves the user anywhere to go.

Section 6.9 won the argument for abstaining with two numbers. This section is about what happens after the decision, which is usually treated as an error path and is actually a product surface.

Three components, and the second is the one that turns a refusal into something useful.

What was searched. "I checked the master agreement and both amendments" tells the user the scope was right and the answer is genuinely absent. Without it, the user cannot distinguish "not in your documents" from "the system did not look properly", and will assume the latter.

Why it is not answerable. Evidence too weak, question outside scope, permission missing, an approval pending. These are different situations with different next actions, and collapsing them into one apology discards the information the user needs.

The route forward. Upload the missing document, rephrase, ask a colleague, escalate to a human. A refusal with no route is where a user gives up on the product rather than on the question.

> **Route to a human deliberately, and carry the state.** A handoff that starts the person from nothing wastes everything the system established. Chapter 10's checkpoint already holds the question, the evidence retrieved, the steps taken and the cost. Hand that over, and the human starts where the system stopped.

Escalation and approval are two faces of the same pattern, the one the chapter title promises: **human-in-the-loop**, any point where the automated flow stops for a decision the system is not entitled to make alone. The book has built it in pieces. Section 9.7's authorise chokepoint decides which actions need a person, section 10.7's interrupt makes the wait affordable, and this section decides what the person sees and where their attention is spent. The pattern lives or dies on that last part, because a human who reviews everything reviews nothing.

The human side of approval has its own failure mode. Per-action review degrades into approval fatigue: a reviewer asked to approve everything approves everything, and section 15.8's audit trail will happily record a rubber stamp as if it were a control. The measure of whether review is real is the disagreement rate, so track it; a reviewer who never rejects is not a control, whatever the process diagram says.

Design the approval surface for informed consent, not completeness. Show the action, its target, whether it is reversible, and the evidence it rests on, and not the whole transcript, because a wall of context is how a reviewer stops reading. Section 10.7's checkpoint payload already carries exactly these fields.

Then spend review where it changes the outcome. Reserve per-action approval for IRREVERSIBLE actions, and use sampled oversight below that: review a few percent after the fact, stratified per section 14.10. Sampling keeps the reviewer's judgement sharp, because most of what they see is worth judging, and it scales in a way a queue of routine approvals never will.

15.6 Rate Limiting and Abuse

Conventional rate limits count requests. That is the wrong unit here, for the reason section 3.1 gave about token limits: one request can cost a hundredth of a cent or several dollars.

```
# Limit the expensive thing, not the countable thing.
limits = {
    "requests_per_minute": 60,        # cheap backstop
    "tokens_per_minute": 250_000,     # the real constraint (3.1)
    "usd_per_hour": 4.00,             # the one that stops the bleeding
    "concurrent_runs": 3,             # agent runs, per Chapter 9
}
```

Three abuse patterns specific to this domain, none of which a request-rate limit catches:

Pattern	What it looks like, and what stops it
Extraction	Systematic querying to reconstruct a corpus. Low rate, high coverage. Detected by breadth of documents touched per user, not by request rate.
Cost amplification	Long inputs, maximum output, deep agent runs. Section 9.6's ceilings and the per-tenant ledger of section 3.6 are the controls.
Free-tier arbitrage	Using your product as a cheaper API. Detected by request shape: no interface signals, uniform prompts, machine cadence.

Breadth needs a baseline before it is a signal. A tenant's own analysts may legitimately touch most of the corpus in a normal week, so alert on deviation from that tenant's own history rather than on an absolute document count, or the first power user becomes the first false positive.

The response to all three should be *degradation before refusal*, per section 15.3: drop the tenant to a cheaper tier and a lower concurrency before cutting them off. That is section 3.6's argument that a policy is a ceiling rather than a refusal, arriving in the abuse path.

15.7 Injection Detection and Response

You cannot prevent injection attempts. You can notice them, and noticing is worth more than most teams assume, because the signals are already in the traces.

Signal	Where it came from
Denied tool calls, by tool and tenant	Section 9.10. A rate change is the alert, not an individual event.
Output policy rejections	Section 15.1. Especially secret-scan hits, which should be zero.
Memory write attempts from non-user sources	Section 11.9. Should be structurally impossible, so any occurrence is a bug or an attack.
Tool results failing schema or size caps	Section 12.7. A vendor result that suddenly grew tenfold.
Answers citing evidence never retrieved	Section 6.9. Ungrounded citations rising is a quality signal and a security one.

> DETECTION WITHOUT RESPONSE IS A DASHBOARD
>
> Decide in advance what happens when the rate moves. A useful default ladder: **log** at baseline, **drop the tenant's risk ceiling** to read-only when denials spike, **quarantine the document** when a specific source correlates with denials across tenants, and **page a human** when a secret-scan hit occurs at all.
>
> The third is the one worth building carefully. An injected document in a shared corpus attacks every tenant who retrieves it, and the correlation is visible only if you record which document was in context when the denial fired.

One thing not to do: **do not respond by editing the system prompt to forbid the specific attack.** That is section 5.9's failed mitigation, it does not generalise, and each addition costs tokens on every request forever. Fix the control, add the case to section 14.9's suite, and move on.

15.8 Audit Trails

An audit trail answers a question asked months later by someone who was not there: *what did the system do for this person, on what basis, and who permitted it?* Chapter 1's traces are the raw material and are not sufficient, because they are engineering artefacts with engineering retention.

Record	Why it must be separate from the trace
Every irreversible action	Who approved it, what was rendered to them, what state was in the checkpoint. Section 10.7's payload is the evidence.
Every policy verdict	Section 15.1. A rejection is as important as an allow, and it cannot be reconstructed.
Every memory write and deletion	Section 11.8's obligations are unanswerable without it.
Every degradation	Section 15.3. "Why was this answer poor on Tuesday" has an answer if the tier was recorded.

Three properties distinguish an audit trail from a log. It is **append-only**, so it cannot be tidied. Its **retention is set by obligation** rather than by debugging convenience, and is usually years where traces are weeks. And it is **readable by a non-engineer**, because the person asking will be from legal, support or a customer.

The efficient arrangement is to derive it: emit audit events from the same chokepoints that already produce verdicts, into a separate store with its own retention. Building it from traces after the fact does not work, because traces will have expired and were never designed to be complete.

It helps to name what these mechanisms answer to. The deletion path of section 15.4 is GDPR-shaped: the right to erasure is why deletion must reach all five surfaces, and retention set by obligation is that regime's vocabulary. The EU AI Act's transparency and record-keeping obligations are what this section's audit trail and section 13.8's disclosure exist to satisfy. The obligations will change and vary by jurisdiction, which is why this book specifies the mechanisms rather than the clauses; but when section 15.10's promotion trigger says a regulated customer, these two regimes are what that customer's questionnaire will cite.

15.9 Incident Response for AI Systems

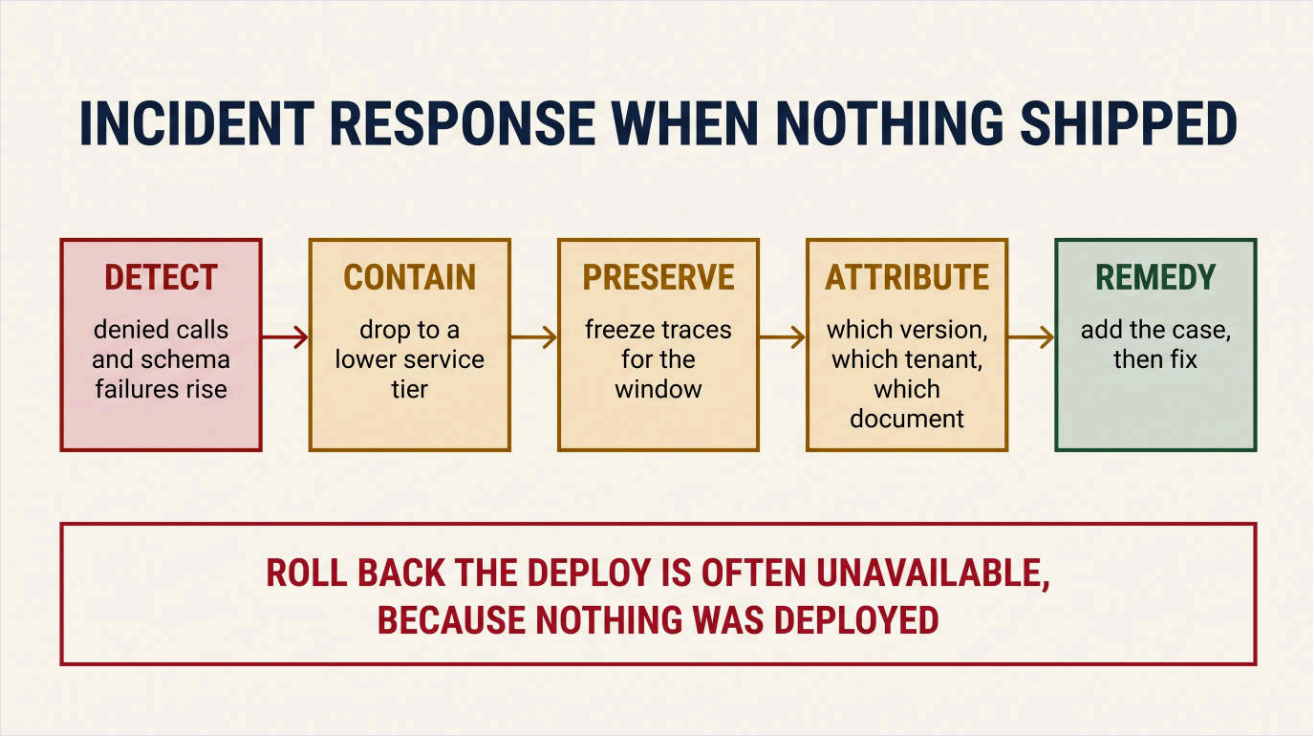

Figure 15.7 The first move of conventional incident response is often unavailable here.

Conventional response begins with "what changed" and usually ends with a rollback. Section 14.5 established that quality here moves without anything shipping, so the playbook needs adapting.

Detect. Section 14.5's six proxies plus section 15.7's five signals. Both are already in the traces, which is why the trace seam was the first thing Chapter 1 built.

Contain. Drop a service tier before diagnosing. Section 15.3's ladder means containment is a config change, not a decision: reduce the risk ceiling, disable the affected tool, or move to extractive answers. A system whose only

containment is "turn it off" will stay on too long.

Preserve. Freeze traces for the window immediately. Default retention is short and an investigation takes longer than the retention, which is a mistake you get to make once.

Attribute. Which prompt version, which model version, which index build, which tenant, which document. Section 1.2's fingerprint exists precisely so this is a query rather than an archaeology project.

Remedy. Add the case to section 14.2's set *first*, then fix. Fixing without the case means the same failure returns in six months with nobody remembering why the code looked like that.

Rehearse the one that is hardest: a provider changed a model and quality fell, with no deploy of yours. There is nothing to roll back. The response is to pin to a dated model version, which you can only do if section 3.1's advice about aliases was taken, and to route to the secondary provider from section 3.3. A team that has not rehearsed this discovers both dependencies during the incident.

15.10 Three Safety Architectures Compared

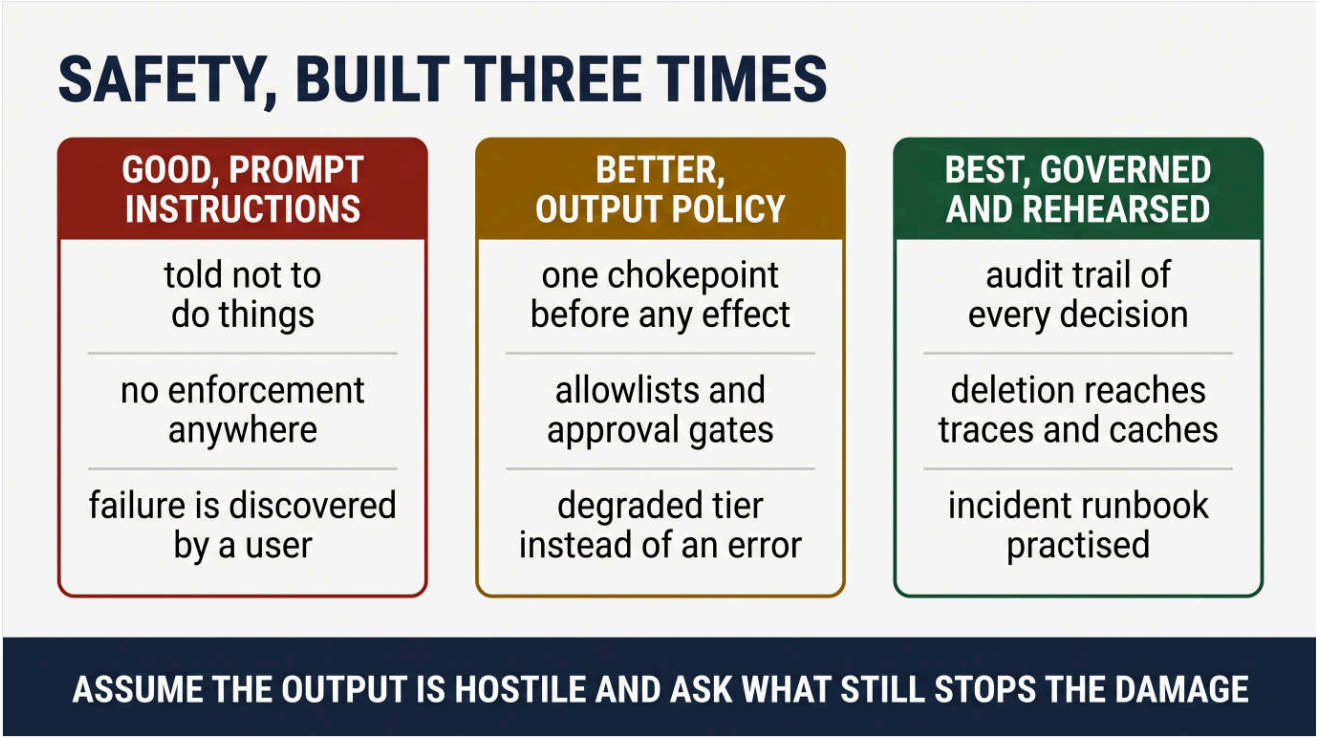

Figure 15.8 The question that separates the tiers is what still stops the damage when the model is fully compromised.

GOOD

Prompt instructions

The system prompt tells the model what not to do. This is where nearly every system starts and it is not nothing: it shapes behaviour in the ordinary case, which is most cases.

What it leaves unbounded: everything, under adversarial input. There is no enforcement, so the answer to "what stops the damage" is "the model chooses not to", which section 5.9 established is not an answer. Failures are discovered by users.

BETTER

An enforced output policy

Section 15.1's chokepoint before any effect: schema, secret scan, tool allowlist, approval for irreversible actions. Plus section 15.3's degradation ladder, so failure produces a reduced service rather than an error or a confident wrong answer.

What it still does not solve. There is no audit trail separate from traces, so a question asked in six months is unanswerable. Deletion reaches documents but not traces and caches. And nobody has rehearsed the response, so the first incident is also the first practice.

BEST

Governed and rehearsed

Everything above, plus the audit trail of section 15.8 with its own retention, deletion that reaches all five surfaces of section 15.4, injection signals wired to a response ladder, and section 15.9's runbook practised including the provider-drift scenario.

Axis	Good: instructions	Better: output policy	Best: governed and rehearsed
Under a hostile output	Nothing stops it	Bounded by allowlist and approval	Bounded, recorded, alerted
Failure behaviour	Error or confident wrong answer	Visible degradation	Visible degradation, each tier tested
Deletion	Documents only, if that	Documents and memory	All five surfaces including traces
Answerable in six months	No	Only while traces live	Audit trail with its own retention
Injection	Undetected	Rejected at the chokepoint	Rejected, correlated, quarantined
Build cost	An afternoon	Two to three weeks	Six to ten weeks

PROMOTION TRIGGERS

Good to better, before the first tool with a side effect, and before the first external user. Not negotiable in either case: an output policy is two weeks and is the difference between a bounded and an unbounded failure.

Better to best, at the first of: a regulated customer, personal data in the corpus, or an irreversible action taken on a user's behalf. *Build the audit trail before you need it*, because it is the one control that cannot be added retroactively: the events it needed to record have already happened and were not recorded.

What Chapter Fifteen Establishes

What exists now	What the last chapters put through it
Output policy at one chokepoint	Rollout risk and blast radius *(16.3)*
Service tier on every response	SLOs defined on tiers, not just uptime *(16.7)*
Deletion across five surfaces	Data residency and regional deletion *(16.5)*
Audit trail with its own retention	The capstone's decision log *(17.2)*
Injection signals and response ladder	On-call alerting and runbooks *(16.9)*

Chapter checklist

Can you...	Section
State the test that decides your real security posture	15.1
Say why a policy applied in three places is not a control	15.1
Distinguish a mitigation from a control, with an example of each	15.2
Name four service tiers and say why each must be visibly different	15.3
List five places personal data comes to rest	15.4
Explain why redaction is a reduction rather than a removal	15.4
Give the three components of a refusal a user can act on	15.5
Say why request-rate limiting is the wrong unit here	15.6
Name five injection signals already present in your traces	15.7
Say why you should not respond to an injection by editing the prompt	15.7
Give three properties that distinguish an audit trail from a log	15.8
Work the incident where nothing was deployed and quality fell	15.9

What Chapter 16 builds

Chapter 16 is the operational chapter: what changes when the system is not yours to restart at will. Cost attribution down to the tenant and the feature, so a margin conversation has numbers. Rollout and rollback for artefacts that are not code. Autoscaling on the signals section 8.11 identified, and capacity planning from the arithmetic of section 2.11. Multi-region and data residency, which is where section 4.11's fourth reason for self hosting becomes a requirement

rather than a preference. And the on-call experience for a system whose pager fires on a rate rather than an outage.