Keep PII Out of Your LLM

Practical strategies for detecting, masking, and never sending personal data to a model.

A YOU-SOURCE BOOK

CONTENTS

Keep PII Out of Your LLM

Practical strategies for detecting, masking, and never sending personal data to a model.

You shipped the feature. A support summariser, a CV screener, a "chat with your documents" box, something that took a fortnight and made everyone happy. Somewhere in the payload you send to the model, there is a person's name, their address, their case history, or their salary. You did not decide to send it. You serialised an object and the object had fields.

This book is about closing that gap.

It is written for the people who build these systems: engineering leads, senior developers, architects. You write code, you can read a Docker Compose file, and you are not a privacy lawyer. Your CTO or founder will read the chapter openings and the last part, because they are the ones who sign the security questionnaire.

The thesis

> ***You cannot detect your way to privacy. A PII detector that scores 0.95 in the vendor's demo scores 0.17 on your data. Detection is a second net. The wall is architecture: minimise at source, keep the real value out of band, and prove both.***

Most writing on this topic sells you a detector and stops. This book spends its fourth chapter showing you, with independent benchmark numbers, how far short detectors fall. Not to talk you out of using one. To move it to the position it can actually hold.

Detection is a net, not a wall.

How the book is laid out

Part I shows how much is leaking and maps the five doors it leaves through. **Part II** opens with the accuracy reckoning, then builds the best detector you honestly can. **Part III** covers what to do with what you caught, which is a longer menu than "replace it with `[REDACTED]`". **Part IV** builds the wall. It opens with the complete reference architecture, including a straight answer on what "100%" can and cannot mean, then builds each component in turn. **Part V** makes the whole thing provable, and gives you a thirty-day sequence.

Every chapter stands on its own. If you arrived at Chapter 9 from a search result, you will not need Chapter 3 to follow it.

What is in scope, and what is not

The code is C# on .NET 9. Detection runs in a container you host, called over HTTP. There are no installation walkthroughs here, no `pip install`, no "first, create a resource group". One Docker Compose fragment appears in the entire book, in Chapter 14. Everything else is usage and architecture, because that is the part that is actually hard.

Five things are deliberately out of scope, and this is the only time they are mentioned:

- **Training your own NER model.** Fine-tuning a detector on your domain is a real option and a real project. The book points at it and moves on.
- **Formal differential privacy.** A serious field with its own literature, and overkill for keeping a customer's address out of a prompt.
- **Complete container and Kubernetes hardening.** Chapter 14 summarises what makes your detection container trustworthy and gives you a checklist. A full hardening programme is a different book, by different authors.
- **Choosing a DLP product.** Vendor selection ages badly and depends on what you already own.

- **Legal advice.** Regulations are named where they create an engineering obligation. Your counsel decides what applies to you.

Two warnings about numbers

Every figure in this book traces to a source, and the sources are listed in Appendix D with access dates.

Two categories need care from you, the reader. **Provider retention terms changed at least twice during 2026.** Where this book states one, it prints the date it was checked. Check it again before you rely on it. And **accuracy figures are meaningless without a dataset.** A detector's score on the data it was trained on tells you almost nothing about its score on yours. Chapter 4 is entirely about that gap, and the book applies the same standard to itself.

A note on language

You will not read that a technique makes your data "safe", "anonymous", or "compliant" here. Pseudonymised data is still personal data under GDPR, and a control that reduces risk is not a control that removes it. Every technique in this book is described by what it does and what it leaves behind. That is less comfortable and considerably more useful.

Don't detect what you can refuse to send.

The Leak You Can't See

This chapter is about volume. Not about a breach, not about a fine, just about how much personal data is already moving into language models and why almost nobody can see it happening.

The number that should bother you

Cyberhaven runs endpoint agents inside a lot of companies, which means it can watch what employees actually paste rather than what they say they paste. On its 2026 telemetry, **39.7% of enterprise AI interactions expose sensitive data**. Not 39.7% of employees. Not 39.7% of companies. Four in ten of the individual interactions.

The direction matters more than the point estimate. The share of corporate data entering AI tools that qualifies as sensitive was **10.7%** two years ago, **27.4%** last year, and **34.8%** now (Cyberhaven, 2026). It has roughly tripled while everyone was busy arguing about which model to use. Nearly 40% of the files people upload contain PII or payment card data, and 22% of pasted text contains information subject to some regulation.

The second number explains why your dashboards are quiet. Around **77% of employees** paste data into generative AI prompts, and **82% of those pastes come from personal accounts** outside company oversight. Your gateway logs, your enterprise tenant, your carefully negotiated data processing agreement: none of them see the four in five.

A caveat, because this book applies its own standards to itself. That is vendor telemetry from endpoint software, not a census and not a peer-reviewed survey. It is drawn from the population of companies who had already bought monitoring, which is not a neutral sample. Treat the exact figures as indicative. Treat the trend as real, because every independent source points the same way.

This is not a story about bad employees

The temptation is to read those numbers as a discipline problem and respond with a policy. Policies have been tried. They produce a training module, a signed acknowledgement, and no measurable change, because the behaviour is not deviant. It is the path of least resistance doing exactly what paths of least resistance do.

Think about what a person is actually doing when they paste a customer email thread into a chat window. They have a job to do, a tool that does it well, and a clipboard. The alternative is to spend four minutes manually stripping names from a message so they can spend thirty seconds getting a summary. Nobody does the four minutes. The friction is on the wrong side of the decision, so people route around it, and they route around it through an account you cannot see.

That is the consumer half of the problem, and it is mostly solved with procurement and sanctioned tooling. Give people a good enterprise option and much of the shadow traffic comes into the light.

The half this book is about is yours.

Your feature is one of the doors

Here is the part that catches engineering teams, and it has nothing to do with employees pasting anything.

You built a support-ticket summariser. The ticket object has a `customer` navigation property. You serialised the ticket, because serialising the ticket was one line and selecting fields would have been eleven. The prompt that leaves your service every few seconds now contains a full name, an email address, a postal address, a phone number, and whatever the customer typed into the free-text field, which in support tickets is frequently a date of birth, an account number, or a photograph of a document.

Nobody decided that. There is no meeting where someone said "let us send the postal address." It arrived because object graphs are convenient and prompts accept strings.

This pattern shows up everywhere once you look:

- A CV screener that sends the whole document because parsing it into fields was going to be a sprint.

- A "chat with your knowledge base" feature whose knowledge base is an export of Zendesk.
- An agent that calls an internal API, gets back a customer record, and puts the record in its context for the next turn.
- An error handler that attaches the request body to the exception so that debugging is easier.

Four different teams. Four different features. The same root cause: the sensitive value travelled because nothing in the code path was responsible for stopping it.

Blast radius

Before you size a control, size the damage. This book uses **blast radius** for the question: if this specific leak happens, what actually follows?

It varies by more than people expect. A single customer's name reaching a model under a zero-retention contract has a small blast radius. It is still a processing event you need a basis for, and it is not a catastrophe. The same name inside a support corpus you embedded into a vector store, which is queried by every user in a multi-tenant application, has a large one, because the leak is not one record and it is not one time.

Three questions size it quickly:

1. **How many people's data, and how sensitive?** One record or the corpus. Contact details or health data.
2. **Where does it come to rest?** A prompt under zero retention evaporates. A vector index persists, is backed up, and gets copied to a staging environment by someone who needed realistic test data.
3. **Who can reach it afterwards?** Only the requesting user, every user of the tenant, or every tenant.

Answer those three and you know whether you are dealing with something that deserves an architecture (Part IV) or something a field projection fixes in an afternoon (Chapter 12). Most teams apply uniform controls to wildly non-uniform risks, which is how you end up with a redaction layer on your internal changelog summariser and nothing at all on the RAG index that contains the case notes.

Why the obvious fix is the wrong shape

The reflex, at this point, is to buy or build a detector. Put something in front of the model that spots personal data and strips it. It is an appealing shape for the problem: one component, one integration point, one line in the security questionnaire.

That instinct is the reason Chapter 4 exists, and Chapter 4 is uncomfortable. The independent cross-domain benchmarks put the best open-source detectors at an F1 around 0.48 to 0.54, against vendor claims of 0.92 to 0.99. One well-regarded model scores 0.780 on the data it was trained on and **0.169** on financial text it has not seen. Your data is, by definition, data it has not seen.

That does not mean detection is worthless. It means detection cannot be the wall. It is a **second net**, and a second net is a genuinely useful thing to have, as long as you have not fired the wall to pay for it.

The wall is architecture. It is choosing which fields go into the prompt, keeping identifiers out of band, and designing so that a detection miss is survivable rather than fatal. That is cheaper than it sounds and it is mostly available to you today, in code you already control.

Don't detect what you can refuse to send.

What you should take from this chapter

Personal data is flowing into models at a volume that has tripled in two years, and the large majority of it moves through channels your monitoring does not cover. Some of that is employees with a clipboard, which procurement can address. Some of it is your own services, sending whole objects because whole objects were convenient, and that part belongs to you.

Before you can stop it leaving, you need to agree on what "it" is. That turns out to be harder than it looks, because three different definitions of personal data are in common use and they do not agree with each other.

SOURCES FOR THIS CHAPTER

- **39.7% of AI interactions expose sensitive data; 34.8% of data entering AI tools is sensitive (up from 27.4% and 10.7%); ~40% of uploaded files contain PII or PCI** — Cyberhaven, 2026 AI Adoption & Risk Report: https://www.cyberhaven.com/blog/sensitive-data-flowing-into-ai-tools · *Vendor endpoint telemetry from organisations that had already deployed monitoring, not a survey. The caveat is stated in the text.*
- **77% of employees paste into GenAI prompts; 82% of those from personal accounts** — https://www.esecurityplanet.com/news/shadow-ai-chatgpt-dlp/ · *Secondary reporting.*
- **Detector accuracy figures referenced ahead of Chapter 4** — sourced in full there.

Unsourced, and offered as the author's argument rather than research: that the dominant cause of prompt leakage is object serialisation rather than deliberate choice, and the three-question blast radius model. Both reason from the mechanism; neither is a measurement.

What Counts as PII

This chapter settles the definition, because three definitions are in common use, they disagree, and the disagreement is where engineering teams build the wrong thing.

Three definitions that do not agree

Ask a US security engineer, a healthcare compliance lead, and a European privacy officer what counts as personal data. You get three answers.

State law: "personal information." The CCPA, as amended by the CPRA, defines it as information that identifies, relates to, describes, is reasonably capable of being associated with, or could reasonably be linked with a particular consumer **or household**. That last word does more work than engineers expect: a household identifier is in scope even when no individual is named. California is not alone. **Twenty states have a comprehensive consumer privacy law in effect**, and four more were enacted during 2026 with future effective dates, taking the enacted total to twenty-four (IAPP tracker, 2026). Counts published elsewhere differ only because some report laws enacted and others laws in force. The definitions differ in the details and agree on the shape.

HIPAA's 18 identifiers. A literal checklist of what to remove from health data: names, all geographic subdivisions smaller than a state, all date elements except year, phone numbers, email addresses, Social Security numbers, medical record numbers, account numbers, biometric identifiers, full-face photographs, and the rest. Remove all eighteen and you have met the Safe Harbor standard. If you touch health data, this is your working entity list, and Appendix A uses it as a coverage checklist.

GDPR: "personal data." Any information relating to an identified or identifiable natural person. The broadest of the three by a distance, and it applies to you if you have customers in the EU regardless of where your company sits. It is not a list, it is a test, and the test is about identifiability rather than about which field the value sits in. An IP address can be personal data. So can a device identifier, a pseudonymous customer reference, or a free-text note that says "the tall guy in accounts who drives the red truck."

Build to an enumerated list alone and you will build something that passes your own tests and misses most of your obligation, because every one of these regimes reaches data your list does not name. The practical answer is to use the broadest available test to decide what matters and HIPAA's list to decide what to detect first, which is what the rest of this book does.

The distinction engineers get wrong

Direct identifiers point at one person on their own. A full name, an email address, a Social Security number, a passport number. These are the ones you think of, and they are the ones detectors are best at, because many of them have structure.

Quasi-identifiers do not point at anyone on their own and point at exactly one person in combination. ZIP code. Date of birth. Gender. Job title. Employer. The date of a hospital admission.

None of those is personal data in isolation, in any intuitive sense. Together they routinely identify a single individual.

There is a number attached to this, and it is worth handling carefully because it is one of the most misquoted figures in privacy. Latanya Sweeney, working from **1990** US Census data, found that **87%** of the US population was likely unique on the combination of five-digit ZIP code, gender, and full date of birth (Sweeney, 2000). That figure is repeated constantly, usually without its date. When Philippe Golle repeated the analysis on **2000** Census data, the proportion came out at **63%** (Golle, 2006).

Both numbers make the same point and the honest one to quote is the lower, more recent one. Roughly three Americans in five are uniquely identified by three attributes that no engineer would flag as personal data in a code review. That is the whole argument for taking quasi-identifiers seriously, and it does not need the inflated version.

It is also the reason HIPAA's Safe Harbor list, at **45 CFR §164.514(b)(2)**, is both useful and blunt. Safe Harbor removes geographic subdivisions smaller than a state and truncates all dates to the year specifically because ZIP code and date of birth are the classic re-identifying pair. And the standard's own guidance is candid that removing the eighteen does not eliminate re-identification risk from combinations of what remains. That is why the alternative route exists: **Expert Determination**, at **§164.514(b)(1)**, where a qualified person applies statistical methods, certifies that the risk of re-identification is very small, and documents the analysis. Note what the standard does not do: HHS sets no numeric threshold for "very small". The expert defines it for the dataset and the

release environment, and must retain the justification for HHS to inspect. If you were hoping for a number to engineer against, there isn't one. It retains more detail, such as month-level dates and sub-state geography, at the cost of needing a defensible analysis rather than a checklist.

For prompt data specifically, quasi-identifiers matter less than they do for datasets, because you are usually sending one record rather than a population. They matter enormously the moment you build a corpus, which is Chapter 15's problem.

Pseudonymised data is still personal data

This is the single most consequential sentence in the chapter, and it is worth stating without hedging.

Under GDPR, replacing a name with a token does not take the data outside the regulation. If the mapping from token back to person exists anywhere, under anyone's control, the data is pseudonymised, and pseudonymised data is personal data. Your obligations travel with it.

US state law reaches the same place by a different route. The CCPA carves out "deidentified" information, and then defines it strictly: you must take reasonable measures to prevent reidentification, publicly commit to keeping it deidentified, and contractually bind anyone you give it to. A token plus a lookup table you still hold is not that. It is pseudonymisation, and the data remains personal information.

Anonymisation is a much higher bar: the mapping must not exist, for anyone, irreversibly. If you built the round trip in Chapter 9, where a vault holds `PERSON_1 → Sarah Whitfield` so the answer can be restored before the user sees it, you have built pseudonymisation. That is a good and useful control. It reduces exposure at the model, it limits what a provider retains, it shrinks blast radius. It does not move you outside the regulation, and any vendor telling you it does is selling something.

Why this matters in practice: it changes what you promise. "We anonymise data before sending it to the model" is a claim you probably cannot support and should not put in a security questionnaire. "We pseudonymise personal data before it reaches the model provider, and the mapping never leaves our infrastructure" is accurate, defensible, and describes a stronger control than most of your competitors have.

Quiet data

Most teams can name the fields that hold personal data in their primary tables. Almost no team can name where it ends up.

Quiet data is personal data sitting somewhere nobody classified. It shows up in five places with grim reliability.

Free-text fields. Notes, descriptions, comments, the "anything else we should know?" box. Users put anything in there: medical conditions, other people's names, account numbers, the reason they are disputing a charge. A free-text field is an unschematised store of arbitrary personal data and it is usually the field most likely to be sent to a model, because free text is exactly what language models are for.

Exception messages and logs. An exception thrown while processing a record often carries the record. Serialised request bodies attached to error reports. A structured logger that helpfully logs the whole DTO at debug level in production because someone was debugging in March. Chapter 16 goes at this properly, because for many organisations the log store holds more personal data than the database does, with weaker access controls and longer retention.

Document and file metadata. A Word document carries its author. A photograph carries GPS coordinates and a device identifier. A PDF carries the name of whoever printed it. If you are extracting text from uploaded files and sending it onward, you are probably also sending the metadata, or discarding it without noticing that you had it.

Identifiers that look like plumbing. A customer reference, a session token, an internal user ID. These feel like infrastructure and they function as pseudonyms, which by the paragraph above makes them personal data when a mapping exists. It does exist. It is in your users table.

Derived and inferred values. A risk score, a churn probability, a segment label. These relate to an identified person and can be more sensitive than the inputs, because an inference about someone's health or finances is an inference regardless of how it was computed.

A working definition for the rest of this book

You need something you can implement against, so here it is.

> *Treat a value as personal data if it identifies a person on its own, if it identifies a person in combination with other values in the same payload, or if a mapping exists anywhere that ties it back to a person. Apply this to free text and file contents, not only to typed columns.*

That definition is deliberately broader than what you will end up sending to a detector, and it should be. The definition sets the scope of your thinking. The detector's entity list is a subset you choose knowingly, with the gap documented, which is Chapter 7's job.

Two practical consequences follow immediately, and both are cheap.

Build an inventory before you build a control. For each LLM feature, write down what fields enter the prompt, what enters any corpus, and what enters logs. Most teams discover at least one field they did not know they were sending, and the discovery is usually free to fix.

Classify the sensitivity tiers you actually have. Contact details, financial identifiers, government identifiers, health data, and data about children are not the same risk and should not attract the same safeguard. A single global policy forces you to pick between over-redacting everything and under-protecting the serious categories.

With a definition in hand, the next question is where it physically escapes. There are five exits, most teams guard one of them, and the one they guard is not the widest.

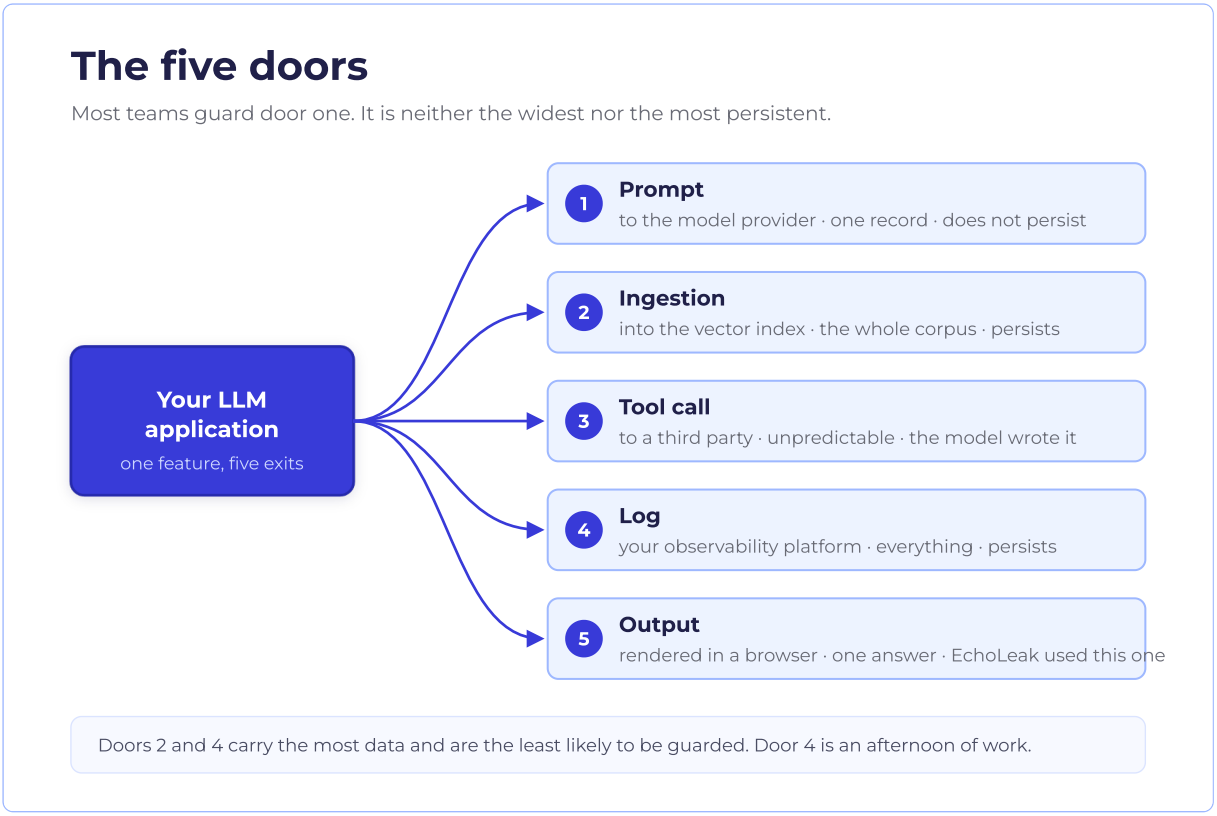
SOURCES FOR THIS CHAPTER

- **CCPA/CPRA definition of personal information including household; 24 states with comprehensive privacy laws as of August 2026; the CCPA "deidentified" standard** — US state privacy law trackers: https://www.osano.com/us-data-privacy-laws · https://secureprivacy.ai/blog/us-state-privacy-law-tracker-2026 · *Statutory text is the authority; these are current summaries. Verify the state count before quoting it, since four 2026 laws are enacted but not yet effective.*
- **HIPAA 18 identifiers; Safe Harbor (45 CFR §164.514(b)(2)) vs Expert Determination (§164.514(b)(1)); no numeric standard for "very small" risk; residual combination risk** — HHS, *Guidance Regarding Methods for De-identification of Protected Health Information*: https://www.hhs.gov/hipaa/for-professionals/special-topics/de-identification/index.html · *Primary. Regulatory text at 45 CFR §164.514.*
- **87% of the US population unique on ZIP + gender + date of birth (1990 Census)** — L. Sweeney, *Simple Demographics Often Identify People Uniquely*, Carnegie Mellon Data Privacy Working Paper 3, 2000: https://dataprivacylab.org/projects/identifiability/paper1.pdf
- **63% on the same combination using 2000 Census data** — P. Golle, *Revisiting the Uniqueness of Simple Demographics in the US Population*, ACM WPES 2006: https://crypto.stanford.edu/~pgolle/papers/census.pdf · *The book quotes the lower, more recent figure in the prose and names both, because the 87% number is widely repeated without its 1990 date.*
- **GDPR "personal data" (Art. 4); pseudonymised data remains personal data (Recital 26)** — statutory.

Unsourced and offered as the author's argument: the "quiet data" taxonomy (free text, logs, file metadata, plumbing identifiers, derived values) and the working definition at the end of the chapter. They organise the problem; they are not findings.

The Five Doors

This chapter is the map for the rest of the book. Personal data leaves an LLM application through five exits. Most teams guard one, and it is not the widest.



Door one: the prompt

The message you send. The obvious one, the one every article is about, and the one where the tooling market concentrates.

It leaks for the reason Chapter 1 gave: whole objects are convenient. A `Ticket` becomes a JSON blob, a `Candidate` becomes the raw CV text, a conversation history accumulates everything anyone said over forty turns and sends all of it, every turn, forever.

Prompt leakage has one property that makes it the easiest door to close: it is synchronous, single-shot, and entirely within your control. You construct the string. Nothing happens between your code and the network that you did not write. Chapter 12 closes it with a field projection, which is the cheapest control in the book, and Chapter 13 enforces it at a chokepoint so that the next developer cannot reopen it.

It is also the door where the least data usually sits, which is worth saying out loud. One prompt is one record. That is a small blast radius compared to what follows.

Door two: ingestion

Everything you loaded in advance. RAG corpora, embedded documents, uploaded files, fine-tuning sets.

This door is different in kind. Door one sends one record at a time, in response to a user action, and forgets it. Ingestion takes your whole support history, your whole document library, or your whole case file archive, and copies it into a new store that was built for retrieval speed and not for access control.

Three properties make ingestion the highest blast radius door in most applications.

It is **bulk**. You did not send one customer. You sent the corpus.

It is **persistent**. A prompt under a zero-retention agreement is gone. A vector index is a database. It gets backed up, replicated, snapshotted into staging, and copied onto a laptop by someone who needed realistic data.

And embeddings are **not** a safe representation. This is the part that surprises people. A vector looks like noise to a human, so it feels anonymised. It is not. Embedding inversion attacks recover **50% to 70% of the original input words** from compromised vectors, and reconstruction reaches near-optimal accuracy with as few as a thousand samples against black-box encoders. OWASP lists vector and embedding weaknesses as a category in its own right: **LLM08:2025**, renumbered **LLM09:2026** in the current edition. If your vector store contains customer text, it contains customer text, and encoding it did not change that.

Chapter 15 handles this door.

Door three: the tool call

The arguments your agent passes outward.

An agent takes a user's question, decides it needs the customer record, calls `GetCustomer(id)`, receives a full record, and puts that record into its context so it can answer. Now the record is in the prompt for every subsequent turn. Then the agent decides it should file a ticket, and calls `CreateTicket(summary)` where `summary` is a paragraph it wrote containing everything it learned.

Nobody wrote that paragraph. The model composed it, and its contents are not predictable in advance, which is what makes this door structurally different from door one. With a prompt, you control the string. With a tool call, the model controls the string, and your only leverage is what you allow it to call and what you allow the result to reach.

This gets worse when the tool has an outward-facing side effect. A tool that sends an email, posts a webhook, writes to a shared document, or performs a search against a third-party API is an exfiltration path that runs at machine speed and leaves no trace in your prompt logs, because the data left through the tool rather than the completion.

The research literature is direct about this. Simple prompt injection has been shown to leak personal data that agents observed during ordinary task execution, and backdoored tool use is a demonstrated exfiltration technique. The defence is not better prompting. It is capability design: limiting what an agent can do so that a successful injection has a small blast radius. Chapter 15 again.

Door four: the log

Your own telemetry. Application logs, traces, error reports, the analytics event you fire so product can count usage.

This is the widest door in most organisations, and it is guarded least, for a specific and slightly embarrassing reason: the log is not where anyone is looking for a privacy problem. Logs are an operations concern. They live in a different tool, owned by a different team, with retention set by whoever configured the platform three years ago.

Consider what your LLM integration is logging right now. Almost certainly the request and response bodies, because when a model gives a strange answer the first thing anyone wants is the exact prompt. That means your log store holds a verbatim copy of every piece of personal data that went to the model, retained for however long the platform default is, searchable by everyone with access to the dashboard, replicated wherever the vendor replicates.

The detector makes this sharper rather than better. A PII detection service sees, by definition, 100% of your personal data. Turn on its request logging and you have built a single, indexed, centralised store of exactly the thing you were trying to protect. Chapter 14 treats this as a first-class design constraint.

Door four is also where provider-side retention lives: what the model vendor keeps, for how long, and who can look at it. That is a contract question rather than a code question, and Chapter 16 covers both halves.

Door five: the output

What the model emits, and where that lands.

The output door has three distinct failure modes, and they are worth separating.

Regurgitation. The model returns personal data it was given, to a user who should not see it. A support agent asks for a summary of similar past cases and gets another customer's details, because retrieval pulled them in and nothing checked the answer before rendering.

Inference. The model states something about a person that was not in the input. A summary that concludes a customer is in financial difficulty is an inference about an identified person, which is personal data, generated by you.

Rendered exfiltration. The most technically interesting, and the one most teams have never considered. The model's output is not just text. It is text you render. If your client renders markdown, an output containing an image tag causes the browser to fetch a URL, and the attacker chose the URL. Put data in the query string and it leaves silently. The user sees a small broken thumbnail, if they see anything.

This is not theoretical. **EchoLeak, CVE-2025-32711**, was zero-click data exfiltration from Microsoft 365 Copilot: a crafted email, reference-style markdown that slipped past link redaction, and auto-fetched images that carried the data out. No click, no dialogue, no user error. Disclosed by Aim Security in June 2025 and rated CVSS 9.3, it was **patched server-side by Microsoft that month, with no exploitation observed in the wild**. Cite it as a demonstrated technique rather than a live exposure: the pattern outlived the specific bug.

Chapter 13 filters this door in the gateway, in the same place and the same way as door one, which is the argument for having a gateway at all.

Which door to close first

Not the one you are thinking of.

DOOR	VOLUME PER EVENT	PERSISTS?	TYPICAL EFFORT TO CLOSE	USUALLY GUARDED?
Prompt	One record	No	Low	Yes
Ingestion	The corpus	Yes	High	Rarely
Tool call	Unpredictable	Sometimes	Medium	Almost never
Log	Everything, twice	Yes	Low	Almost never
Output	One answer	Depends on rendering	Medium	Rarely

Two things fall out of that table.

The prompt is the most-guarded door and neither the widest nor the most persistent. That is not an argument for ignoring it, since it is cheap to close and it is where the sensitive text originates. It is an argument against stopping there.

The log is low effort and almost never done. If you read one chapter of this book after this one, make it Chapter 16. Turning off request-body logging and redacting at the sink is an afternoon of work that closes the exit holding the largest volume of personal data in most companies.

There is a sixth path worth naming and dismissing, because it comes up in every security review. **Training-data memorisation**: models can be induced to emit verbatim training data, including personal data. One study collected thousands of email addresses, phone numbers and URLs this way, and combining attack techniques roughly doubles extraction rates. That is a real risk and it is a training-time risk, which makes it someone else's threat model unless you are fine-tuning on customer data. For you, it reduces to a contract term: confirm the provider does not train on your API traffic, and Chapter 16 tells you how to check rather than assume.

Start with door one, because everyone does and because the tooling promises the most there. That promise is where Part II begins, and it does not survive contact with the evidence.

SOURCES FOR THIS CHAPTER

- **Embedding inversion recovers 50–70% of original input words; near-optimal reconstruction from ~1,000 samples** — *Transferable Embedding Inversion Attack*, arXiv 2406.10280: https://arxiv.org/pdf/2406.10280 · summary: https://www.promptfoo.dev/lm-security-db/vuln/embedding-inversion-privacy-leak-bd566a31/
- **OWASP 2026 top ten for LLM applications lists vector and embedding weaknesses** — https://genai.owasp.org/ · *Numbered LLM08 in the 2025 edition and LLM09 in the 2026 edition; the book gives both.*
- **EchoLeak, CVE-2025-32711: zero-click exfiltration from M365 Copilot via markdown images.** Disclosed by Aim Security June 2025, CVSS 9.3, patched server-side by Microsoft in the June 2025 update, no exploitation in the wild — peer-reviewed write-up: *EchoLeak: The First Real-World Zero-Click Prompt Injection Exploit in a Production LLM System*, arXiv 2509.10540: https://arxiv.org/abs/2509.10540 · narrative account: https://wraith.sh/learn/markdown-image-exfiltration
- **Training-data extraction yielding thousands of emails, phone numbers and URLs; combining attacks roughly doubles extraction risk** — arXiv 2409.12367: https://arxiv.org/pdf/2409.12367 · TACL: https://direct.mit.edu/tacl/article/doi/10.1162/TACL.a.62/134536/
- **Agent tool-call leakage** — arXiv 2506.01055: https://arxiv.org/pdf/2506.01055

The five-door taxonomy is the author's framing, as is the table ranking doors by volume and effort. The claim that the log is the widest and least-guarded door is an assertion from practice, not a measured result.

The Accuracy Reckoning

This chapter is about one number, and the number is not the one in the marketing. If you read a single chapter of this book, read this one, because it changes what you build.

The claim, and the measurement

Commercial PII detection is sold on accuracy. F1 scores of 0.92 to 0.99 appear routinely in vendor material, from established names. Those numbers are not fabricated. They are measured, on a dataset, and the dataset is the entire story.

In June 2026, an independent benchmark tested five detection approaches across four datasets in two languages, using six entity types that all of the systems claim to support: person name, email, phone, location, credit card, and IBAN. The datasets spanned mixed synthetic text, synthetic financial documents, a corpus covering fifty-plus industries, and real Dutch newspaper text with human annotations.

Here is what came out.

DETECTOR	AI4PRIVACY	GRETEL FINANCE	NEMOTRON-PII	CONLL-2002	AVERAGE F1
Piiranha (86M DeBERTa-v3)	0.780	0.169	0.699	0.519	**0.542**
GLiNER v1 (209M)	0.455	0.607	0.484	0.595	**0.535**
Presidio	0.359	0.298	0.487	0.780	**0.481**
GLiNER v2 (205M)	0.469	0.373	0.510	0.558	**0.478**
Regex only	0.207	0.179	0.297	0.000	**0.171**

The best average across domains is **0.542**.

Read along a row rather than down the column, because the row is where the argument lives. Piiranha scores **0.780** on AI4Privacy and **0.169** on synthetic financial text. Same model, same entity types, same week. A **78% collapse**.

Why this happens

There is nothing dishonest in either number. They measure different things, and only one of them predicts what will happen to you.

AI4Privacy is Piiranha's training distribution. The benchmark scored it on the validation split, which is standard practice and produces the number you would put in a model card. The Gretel financial set is text the model had not seen: different sentence structures, different entity densities, different ways of writing a name next to an account number.

Detectors are evaluated **in-distribution** and deployed **out-of-distribution**. Your support tickets are not AI4Privacy. Your CVs are not AI4Privacy. Your case notes, full of internal product names and a company-specific reference format, are nothing like AI4Privacy. The published score was measured on data that resembles the training data, and your data does not.

This book calls that difference **the demo gap**. Every detector has one. The only question is whether you have measured yours.

The pattern repeats in the same benchmark. Piiranha degrades 78% out of distribution. GLiNER v2 degrades 20%. Presidio degrades 17%. GLiNER v1 degrades 0%, which reads well until you notice it achieves consistency by being mediocre everywhere rather than good anywhere.

A second, larger study reaches the same place from a different direction. PIIBench covers **82 entity types across ten source datasets**. A straightforwardly fine-tuned DeBERTa model scored **F1 0.6476**. More elaborate approaches did worse: a source-conditioned hierarchical model managed 0.5899, and a curriculum-learning variant 0.2772. The strongest previously published comparator on that benchmark scored **0.1723** (Jha, arXiv 2605.25816, 2026).

Note the precise claim there. That is the best published comparator **on PIIBench**, not a statement that the best detector in the world scores 0.17. The point is what happens when you evaluate broad coverage across genuinely varied sources rather than one curated set. The conclusion of the paper is worth carrying: diverse task-specific training data and a simple loss function beat architectural sophistication. Nobody is one clever model away from solving this.

The hybrid that does not work

If a model gets 0.54 and regex gets 0.17, the obvious move is to run both and take the union. Two independent methods, complementary strengths, better coverage.

The benchmark tested it. Adding regex detection to the model-based detectors produced:

- Piiranha: **+0.012** F1
- GLiNER v1: **+0.001** F1
- Presidio: **+0.001** F1

Nothing. And it occasionally made things worse, because regex matches displaced correct model detections with false positives.

That result kills a reflex worth killing. The models have already internalised the easy structural patterns. Layering rules on top adds noise, not recall. Rules earn their place for a different reason, which Chapter 5 is about, and the reason is precision on checksummed identifiers rather than incremental F1.

What the benchmark does not prove

Applying the chapter's own standard to itself.

Three of the four datasets are **synthetic**. Only CoNLL-2002 carries human annotations of real text, and it only covers two of the six shared entity types. Synthetic PII data has regularities that real data does not.

The sample is small. A paired t-test across the four datasets found **no statistically significant difference between the top three models** (p well above 0.10). Piiranha at 0.542 and Presidio at 0.481 are not reliably distinguishable on this evidence. Do not use this table to pick a winner by decimal places.

Span matching used a ±5 character tolerance, which is a judgement call that moves the numbers.

And these are six entity types. A production system usually wants twenty or more, including ones with no public benchmark at all, like your internal customer reference format.

None of that rescues the headline. The caveats affect the ranking. They do not touch the finding, which is that cross-domain PII detection lands somewhere around 0.5 rather than somewhere around 0.95, and that the gap between published and achieved is large enough to invalidate a design that assumes the published number.

What 0.54 means on a Tuesday

F1 is a harmonic mean of precision and recall, which makes it a poor number for making decisions, because it hides the trade-off you actually care about.

Take a detector with recall of 0.85 on your traffic, which would be a good day. You process 10,000 prompts. Each contains an average of three personal data entities. That is 30,000 entities, and **4,500 of them go through untouched**.

Not 4,500 catastrophes. Most will be a name in a context where the name was harmless. Some proportion will be the account number, the medical detail, the home address.

Now raise the threshold to catch them. Recall goes up, precision goes down, and you begin redacting things that were not personal data: the product name that looked like a surname, the order reference that looked like a national ID, the word "Paris" in a sentence

about an office. Your summaries get worse. Users notice, and users route around controls that make their tools worse, which is where Chapter 1 came in.

There is no threshold that makes both problems go away. That is the nature of a probabilistic classifier, and pretending otherwise is how teams end up surprised.

The right conclusion

The wrong conclusion is that detection is pointless. The benchmark's own author, having produced these numbers, still chose Presidio for production. Not on accuracy, on architecture: a complete framework, a replaceable NER backend, a working anonymisation and de-anonymisation pipeline, active maintenance, and **8 to 11 times the speed** of the transformer detectors (15.1 ms versus 118 to 198 ms per text on CPU).

That is exactly the right reasoning, and this book follows it. You pick a detector for how well it fits an architecture you can improve later, not for a decimal place you cannot reproduce.

The right conclusion is that detection changes job. It is not the wall. It is the **second net**: the thing that catches what slipped past a design that was already trying not to send the data. A second net that catches half of what reaches it is genuinely valuable, as long as you sized the rest of the system knowing that is what it does.

Every defensive decision in Part IV follows from this chapter. Fail closed when the detector is unavailable, because you cannot assume it was catching much. Minimise at source, because the field that never entered the prompt cannot be missed. Keep the identifier out of band, because then a miss is a miss on a token rather than on a person. Filter the output as well as the input, because the model may surface what the detector let through.

Detection is a net, not a wall.

Knowing that, build the best net you reasonably can. Start with the one layer that genuinely is close to certain.

SOURCES FOR THIS CHAPTER

Everything quantitative here comes from two independent sources. Both are worth reading directly if you intend to act on the conclusion.

- **The cross-domain F1 table, latency figures, degradation percentages, the regex-hybrid result and the 64% Presidio reversibility rate** — A. Sikkema, *Benchmarking Open-Source PII Detection Across Domains*, June 2026: https://albertsikkema.com/python/security/privacy/2026/06/01/benchmarking-open-source-pii-detection.html · *Method: five approaches, four datasets (AI4Privacy EN+NL, Gretel Finance EN+NL, Nemotron-PII, CoNLL-2002 NL), six shared entity types, ±5 character span tolerance. The caveats in this chapter are the benchmark author's own, reproduced rather than paraphrased.*
- **PIIBench: 82 entity types across ten datasets; fine-tuned DeBERTa F1 0.6476; strongest published comparator 0.1723** — P. Jha, arXiv 2605.25816, 25 May 2026: https://arxiv.org/abs/2605.25816
- **Multilingual PII extraction, for context on the model landscape** — GLiNER2-PII, arXiv 2605.09973: https://arxiv.org/pdf/2605.09973
- **Vendor-claimed F1 of 0.92–0.99, and a commercial filter reported dropping 0.96 → 0.18 out of distribution** — secondary reporting, labelled as such in the text. These are the subject of the argument, not evidence this book verified.

The worked example of 10,000 prompts and 4,500 missed entities is arithmetic on an assumed recall of 0.85, presented as illustration. "Demo gap" is this book's coinage for a phenomenon the sources describe but do not name.

Deterministic Detection

This chapter builds the layer you can actually trust. It is narrow, it runs in microseconds, it needs no model and no container, and on its own it catches about a sixth of what matters.

Where rules genuinely win

Chapter 4 put regex at 0.171 average F1, last by a wide margin. That is a fair verdict on regex as a general-purpose PII detector and a misleading one as a verdict on rules.

Rules win in one specific place: **identifiers with a checksum**. Payment cards, ABA routing numbers, National Provider Identifiers, IBANs, ISBNs. These are not natural language. They are designed artefacts with a defined structure and a built-in integrity check, and the integrity check is the thing that turns a guess into near-certainty.

A regular expression that matches sixteen digits will match an order reference, a timestamp concatenation, a product SKU, and a phone number with the spaces stripped. The same expression plus a Luhn check will reject essentially all of those and accept payment cards. You have gone from a pattern match to a validated identification, and you did it in a hundredth of a millisecond.

That distinction is the whole chapter. **Match, then validate.** A pattern without a validator is a false-positive generator, and false positives are how you end up redacting the order number in an email about an order.

The performance argument

The numbers from the same benchmark:

APPROACH	MEDIAN LATENCY	THROUGHPUT
Regex + checksum	0.1 ms	3,861 texts/sec
Presidio	15.1 ms	63 texts/sec
Piiranha	118.5 ms	8.3 texts/sec
GLiNER v1	161.3 ms	5.9 texts/sec

Deterministic detection is free. Not cheap: free, relative to anything else in the request. It runs in-process, with no network call, no container, no GPU, and no failure mode other than your own code. That means you can run it everywhere, on every payload, without a budget conversation. Run it on prompts, on completions, on log lines at the sink, on tool-call arguments, on file metadata. It costs nothing to add another place.

The model-based layer cannot make that claim, which is why Chapter 13 puts these in different positions in the pipeline.

Luhn, in C#

The check digit algorithm behind payment cards, and behind several national identifiers.

```
public static bool IsLuhnValid(ReadOnlySpan<char> digits)
{
    int sum = 0;
    bool doubling = false;

    for (int i = digits.Length - 1; i >= 0; i--)
    {
        if (!char.IsDigit(digits[i])) return false;

        int d = digits[i] - '0';
        if (doubling)
        {
            d *= 2;
            if (d > 9) d -= 9;
        }

        sum += d;
        doubling = !doubling;
    }

    return digits.Length >= 12 && sum % 10 == 0;
}
```

Used properly, it sits behind a pattern rather than in front of it:

```
private static readonly Regex CardCandidate =
    new(@"\b(?:\d[ -]*?){12,19}\b", RegexOptions.Compiled);

public static IEnumerable<Match> FindPaymentCards(string text)
{
    foreach (Match m in CardCandidate.Matches(text))
    {
        var digits = new string(m.Value.Where(char.IsDigit).ToArray());
        if (IsLuhnValid(digits)) yield return m;
    }
}
```

The regex is deliberately loose because the validator is strict. That is the correct division of labour: cast wide, then verify. Inverting it, with a tight regex and no validator, gives you the worst of both.

ABA routing number, in C#

The nine-digit routing number on every US check and ACH transfer. Weights `3, 7, 1` repeating across the nine positions, and the weighted sum must be a multiple of ten.

```
private static readonly int[] AbaWeights = [3, 7, 1, 3, 7, 1, 3, 7, 1];

public static bool IsAbaRoutingValid(ReadOnlySpan<char> routing)
{
    if (routing.Length != 9) return false;

    int sum = 0;
    for (int i = 0; i < 9; i++)
    {
        if (!char.IsDigit(routing[i])) return false;
        sum += (routing[i] - '0') * AbaWeights[i];
    }

    return sum % 10 == 0;
}
```

Known-valid test values: `021000021`, `011000015`.

A routing number identifies a bank rather than a person, so on its own it is not personal data. Next to an account number it is, and the pair is what appears in the ACH details a customer pastes into a support ticket. Detect the routing number, then look for digits near it.

NPI, in C#

The National Provider Identifier, issued by CMS to US healthcare providers. Ten digits, and the check digit is a standard Luhn, computed after prepending the fixed prefix `80840`. That prefix is the part implementations miss, and without it every valid NPI fails.

```
public static bool IsNpiValid(string npi)
{
    var digits = new string(npi.Where(char.IsDigit).ToArray());
    if (digits.Length != 10) return false;

    // CMS specifies Luhn over the NPI prefixed with 80840.
    return IsLuhnValid("80840" + digits);
}
```

Known-valid test values: `1993999998`, `1234567893`.

Appendix A carries the rest: Social Security numbers, EINs, ITINs, Medicare Beneficiary Identifiers, IBANs for international customers, and the false-positive characteristics of each.

What this layer cannot do

Look back at the Chapter 4 table, at the CoNLL-2002 column. Regex scored **0.000**. Not low. Zero.

That dataset is newspaper text annotated for person names and locations. There is no pattern for a name. There is no checksum for "Springfield". The entire category of unstructured personal data, which is most personal data, is invisible to this layer.

So the honest summary of deterministic detection is a narrow instrument with excellent precision:

ENTITY	DETERMINISTIC?	CONFIDENCE
Payment card	Yes, Luhn	Very high
ABA routing number	Yes, weighted mod-10	Very high
NPI	Yes, Luhn with 80840 prefix	Very high
IBAN	Yes, mod-97	Very high
SSN, EIN, ITIN	Range rules only	Medium, needs context
Email address	Yes, RFC pattern	High
IP address	Yes	High, but often not personal data in context
Phone number	Partly	Medium, format varies wildly
ZIP code	Partly	Medium, collides with other codes
Person name	**No**	**None**
Address	**No**	**None**
Free-text disclosure	**No**	**None**

The bottom three rows are why Chapter 6 exists.

The gap is structural rather than a matter of effort. A checksum works because someone designed the identifier to be machine-verifiable. Nobody designed names. There is no length a name must be, no character set it must use, no rule distinguishing "Brook" the person from "brook" the watercourse, and no way to tell that "Mercedes" is a colleague rather than a car without reading the sentence. The same applies to addresses, to dates that are sometimes dates of birth, and to the sentence where a customer volunteers a medical condition.

That is the boundary. Everything with a designed structure belongs to this layer and belongs to it completely. Everything a human wrote in their own words belongs to a model, with the accuracy that Chapter 4 established, which is worse than you would like and better than nothing.

Two practical warnings

Do not tune these for recall. The value of this layer is that when it fires, it is right. If you loosen the card pattern to catch cards written with unusual separators, you start matching order references, and every false positive damages a downstream result while adding nothing you can rely on. Let this layer be precise and let the model layer chase recall.

Validators drift. Identifier schemes change. Countries add digits, reassign ranges, introduce new formats. A validator written in 2021 against a scheme revised in 2025 fails silently, returning `false` for legitimate identifiers, which reads as "no PII found". Put a dated comment on every validator naming the scheme version, and add a golden test with known-valid examples, so a scheme change fails a test rather than failing open in production.

That second warning generalises into a principle for the whole book: **a detection component that fails should fail loudly**, because a detector that returns nothing looks exactly like clean input.

You now have a fast, precise layer that finds structured identifiers and is blind to names, addresses, and everything a person typed in their own words. That is the majority of the problem, and closing it means a model. There are three ways to get one into a .NET application, and they differ in the one dimension this book cares about most.

SOURCES FOR THIS CHAPTER

- **ABA routing number check digit, weights 3-7-1 mod 10** — Apache Commons Validator reference implementation: https://commons.apache.org/validator/apidocs/org/apache/commons/validator/routines/checkdigit/ABANumberCheckDigit.html · algorithm walkthrough: http://www.brainjar.com/js/validation/
- **NPI check digit: Luhn over `80840` + NPI** — CMS, *Requirements for National Provider Identifier (NPI) and NPI Check Digit*: https://www.cms.gov/Regulations-and-Guidance/Administrative-Simplification/NationalProvIdentStand/Downloads/NPIcheckdigit.pdf · worked example: https://www.johndcook.com/blog/2024/06/26/npi-number/
- **Luhn** — ISO/IEC 7812-1. **IBAN mod-97** — ISO 13616.
- **Latency figures and the 0.171 regex average F1** — Sikkema benchmark, cited in full in Chapter 4.

Every validator in this chapter and in Appendix A was executed against the stated known-valid test values before publication. The argument that this layer's value is precision rather than recall is the author's, supported by the 0.000 regex score on CoNLL-2002 in the Chapter 4 table.

The Three-Way Choice

This chapter picks the model-based detector. There are three routes from a .NET application, and they differ on the one question that matters: where does your text physically go?

The options

Self-host Presidio as a container on your own network and call it over HTTP. **Call Azure AI Language PII detection**, which has a first-class C# SDK and runs in Microsoft's cloud. Or **write nothing but native C# rules** and accept the ceiling from Chapter 5.

	PRESIDIO SIDECAR	AZURE AI LANGUAGE	NATIVE C# RULES
.NET access	REST via `HttpClient`	Official SDK	In-process
Where the text goes	Your network	Microsoft's cloud	Nowhere
Entity coverage	Broad, extensible	Broad, fixed	Checksummed identifiers only
Latency	~15 ms plus a local hop	Network round trip	~0.1 ms
Cross-domain F1	~0.48	Not independently benchmarked here	~0.17
You operate	A container	A subscription	Nothing
Reversible pipeline	Built in	Redaction policies	Build it yourself

What Presidio is, and who owns it now

Presidio detects and de-identifies personal data in text, images and structured data. It has four components:

- **Analyzer** finds entities, using spaCy or HuggingFace NER plus regex recognisers, checksum validation, and contextual scoring.
- **Anonymizer** applies an operator to each finding: replace, redact, mask, hash, encrypt.
- **Image Redactor** does the same over OCR output.
- **Structured** handles tabular and semi-structured data.

One fact that most writing on this subject has not caught up with: **Presidio is no longer a Microsoft project.** The documentation states it is transitioning to a community-owned project, maintained by the Data Privacy Stack organisation. The old `microsoft.github.io/presidio` address redirects twice before landing on `presidio.dataprivacystack.org`. You will still find articles published this year calling it "Microsoft Presidio" and linking to a URL that no longer serves content.

This matters beyond pedantry. Your security review will ask who maintains a dependency that sees 100% of your personal data. The honest answer changed. It is not a reason to avoid Presidio, which remains actively developed, with recent releases adding country-specific recognisers and an optional country filter on the predefined set. It is a reason to treat provenance as a standing question rather than a box you ticked at adoption, which is a theme Chapter 14 returns to.

There is no official .NET SDK

Presidio ships Python packages, Docker images, Kubernetes manifests, and REST endpoints. For any language that is not Python, REST is the route.

There is one unofficial C# client, and it deserves a paragraph rather than a chapter. `Presidio.SDK` by StefH is a RestEase-based client on NuGet. Version **0.0.2**, published **June 2025**, roughly **5.6K** total downloads, fourteen commits, targeting .NET 6.0 and .NET Standard 2.1. The analyzer is demonstrated; anonymizer coverage is not evidenced. A companion `Presidio.SDK.Extensions` package adds a handful of locale-specific pattern recognisers, at around 970 downloads.

Know it exists. Do not build on it. A component in this position, handling every piece of sensitive text in your application, should not depend on a pre-1.0 package with one maintainer and a year since its last release. The REST API is four endpoints and `HttpClient` is in the framework.

Calling the analyzer from C#

Assume the container is running and reachable at `http://presidio-analyzer:3000`. The request is a JSON body with the text, a language, and optionally the entities you care about.

```
public sealed record PiiFinding(
    string EntityType,
    int Start,
    int End,
    double Score);

public sealed class PresidioAnalyzer(HttpClient http)
{
    private static readonly JsonSerializerOptions Json = new()
    {
        PropertyNamingPolicy = JsonNamingPolicy.SnakeCaseLower
    };

    public async Task<IReadOnlyList<PiiFinding>> AnalyzeAsync(
        string text,
        string language = "en",
        CancellationToken ct = default)
    {
        var request = new
        {
            text,
            language,
            score_threshold = 0.5
        };

        using var response = await http.PostAsJsonAsync("/analyze", request, ct);
        response.EnsureSuccessStatusCode();

        return await response.Content
            .ReadFromJsonAsync<List<PiiFinding>>(Json, ct) ?? [];
    }
}
```

Register it as a typed client so that timeouts and retries are policy rather than an afterthought:

```
builder.Services.AddHttpClient<PresidioAnalyzer>(c =>
{
    c.BaseAddress = new Uri(builder.Configuration["Presidio:AnalyzerUrl"]!);
    c.Timeout = TimeSpan.FromSeconds(2);
});
```

Two decisions are already embedded in that snippet and both deserve to be conscious.

The **score threshold** is your precision and recall dial, and 0.5 is a default rather than an answer. Chapter 7 is about choosing it on your own data.

The **two-second timeout** sets up the question Chapter 13 answers properly: what happens when this call fails? A `catch` that logs a warning and proceeds is a system that silently stops protecting anything the moment the container restarts. That is failing open, and failing open is the leak.

Restricting the entity list

By default the analyzer looks for everything it knows. That is slower and noisier than you want. Name the entities your policy actually covers:

```
var request = new
{
    text,
    language = "en",
    entities = new[]
    {
        "PERSON", "EMAIL_ADDRESS", "PHONE_NUMBER",
        "US_SSN", "US_BANK_NUMBER", "CREDIT_CARD",
        "LOCATION", "US_DRIVER_LICENSE"
    },
    score_threshold = 0.5
};
```

Write that list down somewhere a human reads, because it is the exact boundary of what you claim to detect. Everything outside it is undetected by design, and "by design" is a much better answer in an audit than discovering the gap during one.

The Azure option, and the tension in it

Azure AI Language PII detection is the serious .NET-native alternative, and it is a good product. Three feature types: **Text PII** for synchronous string payloads, **Conversation PII** for turn-based transcripts, and **Document PII** for native `.pdf`, `.docx` and `.txt` files, which returns redacted files with the document structure preserved plus machine-readable metadata. Official client libraries for C#, Java, JavaScript and Python. Configurable `redactionPolicies` from the 2025-11-15-preview API version, with more than one policy per request. Entity categories come back with confidence scores, and the model is not customised on your data.

For a .NET team this is the path of least resistance by a distance. Add a package, add a key, call a method.

And now the tension, which no vendor page will state for you: **a cloud PII-detection API requires you to transmit the sensitive data in order to locate it.**

Read that twice, because it is easy to nod past. To find out whether a payload contains personal data, you send the payload. The detection call is itself a disclosure to a third-party processor. You have not removed a data flow, you have added one.

That is sometimes completely fine. Azure offers regional processing with contractual commitments, you likely already have a data processing agreement with Microsoft, and adding one more Microsoft processor to a Microsoft-hosted application changes your risk profile very little. If your model calls are going to Azure OpenAI anyway, routing detection through Azure AI Language adds no new party at all.

It is sometimes exactly wrong. If you self-host your model to keep text inside your perimeter, and then call a cloud API to detect the personal data in that text, you have defeated the entire design with the control that was supposed to enforce it. Teams do this. It happens because detection feels like a security function rather than a data flow, and security functions do not feel like they need the same scrutiny as features.

The test is simple. Draw your trust boundary. Ask whether the detection call crosses it. If it does, that is a decision, and it needs to be a deliberate one rather than a default.

Azure's own data, privacy and security documentation answers the question this book flagged as open in earlier drafts, and the answer is favourable. Language **does not store or process customer data outside the region where you deploy the instance**, and encrypts all content at rest. Request data may be held for up to 48 hours for debugging by on-call engineers after a catastrophic failure, controlled by the `LoggingOptOut` query parameter, and **that parameter defaults to true on the PII and health endpoints specifically** — so the temporary storage that applies to sentiment or key-phrase calls does not apply to the calls you would be making. Verified against Microsoft's documentation on 14 September 2026.

That does not dissolve the tension above. Regional processing and a 48-hour opt-out default are strong terms, and they are still terms: the text crosses your perimeter and you are relying on a contract rather than on a mechanism. Chapter 16 makes that distinction properly.

What this book chooses, and why

The **self-hosted sidecar**, for one reason that outranks the others: it is the only option where the text never crosses your perimeter.

The supporting reasons are the ones Chapter 4's benchmark author gave when making the same call. Presidio is a complete framework rather than a model, so the NER backend is replaceable. If a better detector appears, or you fine-tune one on your own data, you swap the recogniser and keep the pipeline. It runs at roughly 15 ms against 118 to 198 ms for the transformer alternatives, which is 8 to 11 times faster and the difference between a control you can run on everything and one you ration. Its anonymise and de-anonymise pipeline is the round trip in Chapter 9, already built. And it is actively maintained.

Notice what is not in that list. Accuracy. Presidio averaged **0.481** in the benchmark, third of four, and the difference between it and the leader was not statistically significant. You are choosing an architecture that can be improved, not a decimal place you cannot reproduce.

The sidecar also happens to be the shape Part IV argues for anyway: a self-hosted service behind a gateway, driven by versioned policy. Choosing it here costs nothing later.

The container is running and returning findings. What those findings are worth on your data is still entirely unknown, and that is a measurable thing rather than a matter of faith.

SOURCES FOR THIS CHAPTER

- **Presidio's four components, detection method, deployment options, and its move to community ownership under the Data Privacy Stack organisation** — https://presidio.dataprivacystack.org/ · *The former `microsoft.github.io/presidio` address redirects twice to this. Checked 11 September 2026.*
- `Presidio.SDK` **v0.0.2, June 2025, ~5.6K downloads, 14 commits, .NET 6.0 / .NET Standard 2.1, analyzer coverage only** — https://www.nuget.org/packages/Presidio.SDK · https://github.com/StefH/Presidio.SDK
- **Azure AI Language PII: three feature types, official C#/Java/JS/Python SDKs,** `redactionPolicies` **from API version 2025-11-15-preview, confidence-scored categories, no model customisation on customer data** — https://learn.microsoft.com/en-us/azure/ai-services/language-service/personally-identifiable-information/overview · *Page dated 30 June 2026, updated 3 August 2026.*
- **Azure AI Language data handling: no storage or processing outside the deployment region; all content encrypted at rest; up to 48 hours temporary storage for catastrophic-failure debugging, governed by** `LoggingOptOut`**, which defaults to true on the PII and health endpoints** — Microsoft, *Data, privacy, and security for Azure Language*: https://learn.microsoft.com/en-us/azure/ai-foundry/responsible-ai/language-service/data-privacy · *Page dated 1 April 2026, updated 17 August 2026. Verified 14 September 2026.*

The argument that a cloud detection API constitutes a disclosure in order to prevent one is the author's, and it describes a data flow rather than a defect in the service. The book's choice of the self-hosted sidecar is a judgement made on the architectural grounds listed, not on measured accuracy.

Measuring Your Own Demo Gap

Chapter 4 said every detector has a demo gap. This chapter measures yours, in about a day, using a few hundred examples from your own traffic.

Why a benchmark cannot answer this

The published numbers tell you how a detector performs on somebody else's text. Useful for ruling things out, useless for deciding a threshold, and dangerous as a basis for a claim in a security questionnaire.

What you need is smaller than a benchmark and far more valuable: a few hundred labelled examples of **your** data, scored against **your** entity list, at **your** threshold. Three hundred examples will tell you more about your production risk than every published F1 score combined, because the distribution is right.

This is a day of work. Teams skip it because labelling feels tedious next to building, and then run a detector for two years without knowing whether it catches 40% or 90% of what passes through.

Building the golden set

Sample real traffic, stratified. Not the first 300 requests, which will over-represent whatever ran that morning. Pull across features, across time of day, across languages, across your tenants if you are multi-tenant. Include the ugly cases deliberately: the longest prompts, the ones with pasted documents, the ones with non-Latin names, the ones from the customer whose free-text notes are a novel.

Aim for 200 to 500 samples. Below 200 the confidence intervals are too wide to act on. Above 500 you are spending labelling effort that would be better spent on a second round later.

Labelling is itself a personal data processing operation. You are about to copy production data containing personal data into a working set that humans will read. Do this properly or you have created the exact problem you are solving. Keep it inside your perimeter, restrict access to named people, set a deletion date, record the lawful basis, and never put it in a public repository or a shared spreadsheet. If that sounds heavy, note that the alternative is a folder called `pii-test-data` on someone's laptop, which is where these things actually end up.

Label spans, not documents. For each sample, record the character offsets, the entity type, and the true value. Document-level labels ("this one contains PII") cannot measure partial detection, and partial detection is the normal failure mode. A detector that finds the surname and misses the given name has leaked a person's given name, and a document-level label scores that as a success.

Have two people label a subset. Take 50 samples and label them independently. Where the two disagree, your entity definitions are ambiguous, and ambiguous definitions produce a measurement that drifts. Common disagreements: is a company name a `LOCATION`? Is a job title personal data? Is the sender's own name in a signature block in scope? Settle these in writing before the main pass.

Scoring

Store the set as JSON alongside your test suite, then score the detector against it.

```
public sealed record LabelledSpan(int Start, int End, string EntityType);

public sealed record Scores(double Precision, double Recall, double F1)
{
    public static Scores From(int tp, int fp, int fn)
    {
        double p = tp + fp == 0 ? 1 : (double)tp / (tp + fp);
        double r = tp + fn == 0 ? 1 : (double)tp / (tp + fn);
        double f = p + r == 0 ? 0 : 2 * p * r / (p + r);
        return new Scores(p, r, f);
    }
}
```

Matching needs a tolerance, because detectors disagree with humans about whether a trailing space or an honorific is part of the span.

```
private const int OffsetTolerance = 5;

private static bool Matches(LabelledSpan expected, PiiFinding actual) =>
    expected.EntityType == actual.EntityType
    && Math.Abs(expected.Start - actual.Start) <= OffsetTolerance
    && Math.Abs(expected.End - actual.End) <= OffsetTolerance;
```

Then the loop, counting each labelled span once:

```
public static Scores Evaluate(
    IReadOnlyList<(string Text, IReadOnlyList<LabelledSpan> Expected)> samples,
    Func<string, IReadOnlyList<PiiFinding>> detect)
{
    int tp = 0, fp = 0, fn = 0;

    foreach (var (text, expected) in samples)
    {
        var found = detect(text).ToList();
        var unmatched = expected.ToList();

        foreach (var finding in found)
        {
            var hit = unmatched.FirstOrDefault(e => Matches(e, finding));
            if (hit is not null) { tp++; unmatched.Remove(hit); }
            else fp++;
        }

        fn += unmatched.Count;
    }

    return Scores.From(tp, fp, fn);
}
```

Score per entity type, never only in aggregate. An overall F1 of 0.62 can hide `EMAIL_ADDRESS` at 0.97 and `PERSON` at 0.31. Those need different responses, and the aggregate tells you to do nothing in particular. The public benchmarks show exactly this spread: email detection reaching 0.96 to 0.99 while person names swing between 0.14 and 0.79 depending on the corpus.

Choosing the operating point

Now the decision the F1 score was hiding.

A false negative leaks. A false positive annoys. These are not symmetric, and the ratio between them is yours to set rather than the tool's.

Sweep the threshold and look at the shape:

THRESHOLD	PRECISION	RECALL	WHAT IT MEANS
0.30	0.54	0.91	Catches most things, redacts a lot that was fine
0.50	0.78	0.74	The default nobody chose
0.70	0.91	0.52	Clean output, misses half
0.85	0.97	0.28	Barely firing

Those figures are illustrative. The shape is not: recall falls away faster than precision rises, which means the cost of the last few points of recall is steep.

Three things should drive where you sit.

Blast radius, from Chapter 1. A prompt to a zero-retention endpoint that vanishes after the call tolerates a lower threshold than text you are about to embed into a vector store forever.

Entity severity. Run different thresholds per entity. A missed `CREDIT_CARD` and a missed `LOCATION` are not the same event. Threshold cards and national identifiers aggressively, names moderately, locations loosely.

Whether a human is in the loop. If output goes straight to a customer, false positives are a product defect. If it goes to an internal reviewer who can see the original, they are a minor annoyance and you should be far more aggressive.

Write the choice down with its reasoning. When an auditor asks why the threshold is 0.6, "it was the default" is a poor answer and "we measured precision and recall on 340 labelled production samples and accepted 0.74 recall on `PERSON` because the output is reviewed by an internal agent before sending" is an excellent one.

What to do with a bad number

You will get a bad number. The benchmarks put cross-domain detection around 0.5, and you have no reason to expect better.

Do not respond by raising the threshold until the number looks good. That changes the measurement, not the risk.

Four responses actually help.

Minimise the input. Every field you stop sending is a set of entities the detector no longer has to catch. This is Chapter 12 and it beats every detector improvement available to you, usually by a large margin, usually in less code.

Add deterministic rules for what you can validate. Chapter 5's layer will not move your F1 much, but it converts your worst-consequence entities from probabilistic to near-certain, and consequence is what you care about.

Add a domain recogniser. If your data contains a customer reference in a known format, Presidio lets you register a pattern recogniser for it. Your own identifiers are exactly the entities no general detector will ever find, and exactly the ones you can detect perfectly.

Design for the miss. Fail closed, filter the output as well as the input, keep the real identifier out of band. That is Part IV, and it is the only response that helps when the detector is simply wrong.

Re-measure, because the gap reopens

The demo gap is not a one-time measurement. It widens whenever your inputs change and you did not notice.

Re-measure when you enter a new market or language, when a new customer segment onboards with different data shapes, when you add a feature that accepts a new input type such as uploaded documents, when you upgrade the detector or its NER model, and on a schedule regardless, because gradual drift produces no event to trigger a check.

Make it a test that runs in CI. Load the golden set, run the detector, assert that per-entity recall has not dropped below the thresholds you committed to. A model upgrade that quietly halves recall on `PERSON` then fails a build instead of shipping.

```
[Fact]
public void PersonRecall_MeetsCommittedFloor()
{
    var scores = Evaluate(GoldenSet.ForEntity("PERSON"), Detector.Detect);
    Assert.True(scores.Recall >= 0.70,
        $"PERSON recall fell to {scores.Recall:P0}; committed floor is 70%.");
}
```

That test is also your evidence. When Chapter 17 asks what you can show an auditor about detector performance, this is the artefact, and it has been running every day since you wrote it.

You know what your detector finds, and you know what it misses. The next question is what to do with what it found, and the menu is considerably longer than replacing it with `[REDACTED]`.

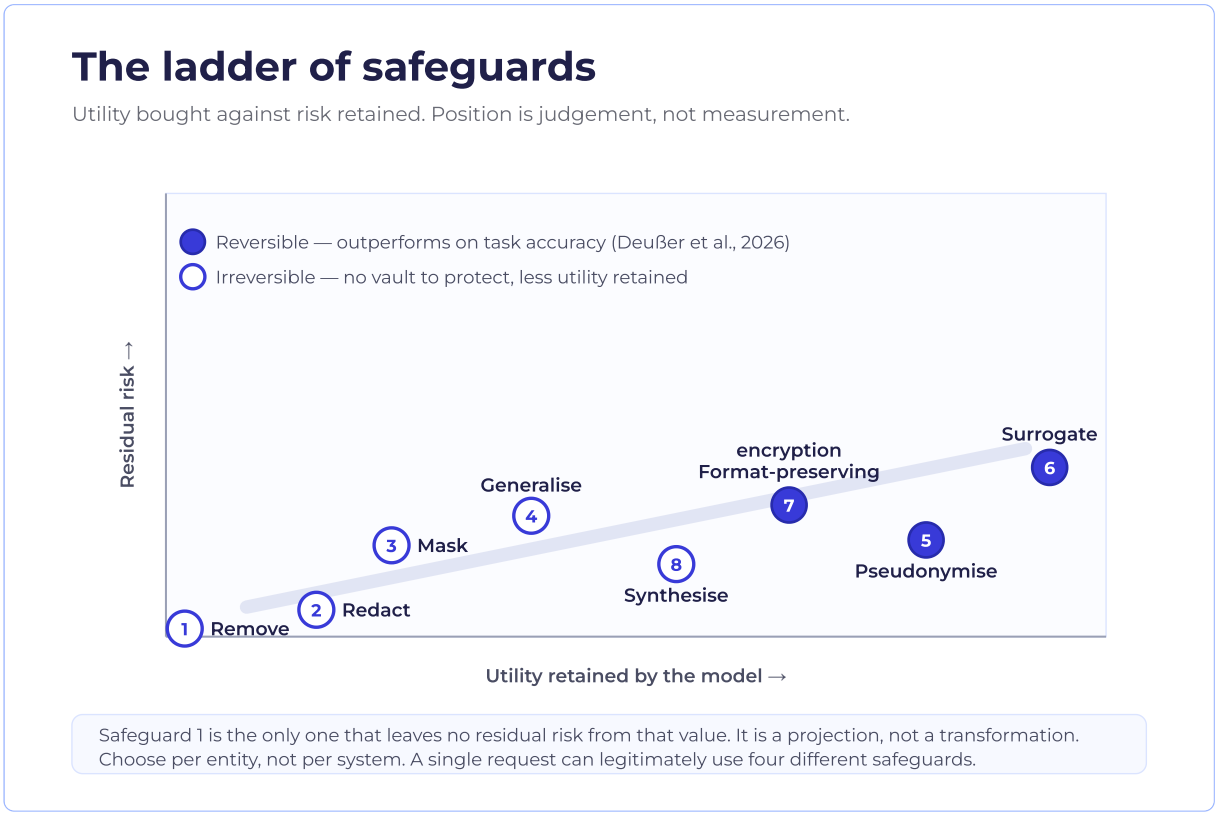
SOURCES FOR THIS CHAPTER

- **Per-entity accuracy spread (email 0.96–0.99 against person names 0.14–0.79 depending on corpus)** — Sikkema benchmark, cited in full in Chapter 4: https://albertsikkema.com/python/security/privacy/2026/06/01/benchmarking-open-source-pii-detection.html
- **±5 character span-matching tolerance, and the finding that the top three models are not statistically separable** — same source.

The threshold sweep table is illustrative and labelled as such in the text: the shape is real, the specific numbers are not measurements. The golden-set method, the 200–500 sample guidance, and the false-negative-versus-false-positive asymmetry are the author's practice rather than published findings. Treating the labelling set as a personal data processing operation follows from Chapter 2's definition, not from a cited source.

The Ladder of Safeguards

You found something. This chapter is the menu of what to do with it, which has eight options rather than the one everybody reaches for.



The eight safeguards

Every safeguard trades utility for risk reduction. The trade is not linear and the right safeguard differs per entity, per task, and sometimes per tenant.

#	SAFEGUARD	EXAMPLE	MODEL CAN STILL	RESIDUAL RISK
1	**Remove**	field never sent	nothing with it	None from this value
2	**Redact**	`[REDACTED]`	know something was there	Very low
3	**Mask**	`****1234`	match, compare suffixes	Low, but partial values leak
4	**Generalise**	`1985-03-12` → `35-44`	reason about the band	Low alone, real in combination
5	**Pseudonymise**	`PERSON_1`	track the entity, keep coreference	Low at the model, full at your vault
6	**Surrogate**	`Sarah Whitfield` → `Megan Alvarez`	everything, naturally	Low, plus a plausibility hazard
7	**Format-preserving encrypt**	valid-looking card number	validate format	Low, key-dependent
8	**Synthesise**	regenerate the record	work on realistic data	Low, needs evaluation

Read it top to bottom as increasing utility and increasing complexity. Safeguard 1 is a `select` statement. Safeguard 8 is a project.

Removal, and why it is first

The field never enters the prompt. Nothing to detect, nothing to restore, nothing to leak.

Removal is first on the list because it is first in practice and teams reach for it last. The question is not "how do I mask the customer's address" but "does the summarisation task need the address at all?" For a support-ticket summariser, almost never. For a delivery-issue triage tool, sometimes. Ask before you engineer.

This is Chapter 12's whole subject and the single highest-leverage control in the book. Everything below safeguard 1 exists for values that genuinely need to be there.

Redaction and masking

Redaction replaces the value with a marker. `[REDACTED]`, `[PERSON]`, `***`. The model knows something was removed and knows roughly what kind of thing it was, if you typed the marker.

It has one significant flaw for conversational work: it destroys identity. Two different people both become `[PERSON]`, so the model cannot tell whether the sentence is about one person or two. For a one-shot classification that does not matter. For a summary of a three-party email thread it produces nonsense.

Masking keeps part of the value. `****1234` for a card, `j***@example.com` for an email, `+31 6 ** ** 12 34` for a phone.

Masking is for humans, not models. Its purpose is letting a support agent confirm "yes, the card ending 1234" without exposing the number. A model gains almost nothing from the last four digits, and the retained characters are a real disclosure: the last four digits of a card plus a name plus a merchant is enough to be useful to someone. Use masking when a person will read the output. Do not use it as a model-facing control and assume it is stronger than redaction, because it is weaker.

Generalisation

Replace a precise value with a band. Exact date of birth becomes an age range. A five-digit ZIP becomes its first three digits. Salary becomes a decile. A precise timestamp becomes the month.

Generalisation is under-used, because it requires thinking about the task rather than applying a uniform rule. If you are asking a model to classify support tickets by urgency, it does not need the customer's date of birth, and it might genuinely benefit from knowing the customer is over 65. Give it the band.

One caution, and it corrects an intuition many people have here. Generalisation feels like the sophisticated choice because it preserves meaning, but measured against model task performance it does **worse** than a plain token. Deußer et al. (2026) evaluated five anonymisation strategies across eleven benchmarks and five models, and found that the reversible methods (pseudonymisation, randomisation, masking) significantly outperformed the irreversible ones. On the MUSR reasoning benchmark, generalisation scored **0.58** against pseudonymisation's **0.70**. Generalisation's value is in reducing the precision of a quasi-identifier, which is a privacy gain. It is not a utility gain, and you should not choose it expecting one.

The caution comes straight from Chapter 2. Generalised values are quasi-identifiers, and quasi-identifiers combine. Age band plus partial ZIP plus gender plus employer can identify one person as surely as a name. Generalisation reduces the risk of a single value and does very little about the risk of a set of them. When you are sending one record it is a strong control. When you are building a corpus, count the combinations.

Pseudonymisation

Replace each entity with a stable token. `Sarah Whitfield` becomes `PERSON_1` everywhere it appears, and the mapping lives in a vault you control.

This is the default for production LLM work, because it preserves the one thing redaction destroys: **coreference**. The model can follow that `PERSON_1` appears in turn one and turn seven and is the same person. It can reason about relationships between `PERSON_1` and `PERSON_2`. It can produce a summary that tracks who did what, and you can restore the real names before anyone reads it.

Two things must be stated plainly, both from Chapter 2.

Pseudonymised data is still personal data. The mapping exists, so GDPR still applies in full. This reduces exposure at the model, it does not take you outside the regulation, and you should never describe it as anonymisation in a security questionnaire.

The vault is now your most sensitive store. You have concentrated the mapping from token to real person into one place. That place needs the safeguard Chapter 14 describes, and it is worth being honest that you have traded a diffuse risk for a concentrated one. Concentrated risk is easier to protect and worse when it fails.

Chapter 9 builds this end to end.

Surrogates

Replace the real value with a realistic fake one. `Sarah Whitfield` becomes `Megan Alvarez`, not `PERSON_1`.

Surrogates preserve utility better than tokens because the text stays natural. Models were trained on text containing names, not text containing `PERSON_1`. The measured picture is more nuanced than "tokens are bad": Deußer et al. (2026) found reversible strategies including randomisation, which is close to a surrogate, outperforming irreversible ones across eleven benchmarks, while the degradation from any anonymisation was **highly task-dependent** rather than uniform. A surrogate reads like the original, so the model tends to behave as if it had the original. Whether that matters for your task is something you measure, not something you assume.

Two hazards, one obvious and one not.

The obvious one is consistency. If the same person becomes `Megan Alvarez` in one paragraph and `Dana Reyes` in the next, you have destroyed coreference more thoroughly than redaction would have. Surrogate assignment must be deterministic per entity per context, which means a vault, the same as pseudonymisation.

The non-obvious one is the **plausibility hazard**. A surrogate is indistinguishable from real data to a downstream human. If the model's output reaches a person who does not know substitution happened, they will act on a fabricated name as though it were real. Support agents have contacted the wrong customer this way. If you use surrogates, the boundary where values are restored must be unambiguous, and anything that escapes that boundary must be labelled.

Format-preserving encryption

Encrypt the value so the ciphertext has the same shape as the plaintext. A sixteen-digit card becomes a different sixteen-digit number that still passes a Luhn check. A national identifier becomes a different, well-formed national identifier.

FPE earns its place when a downstream system validates format and would reject a token. Legacy schemas with strict column types, systems that check a checksum before accepting a value, pipelines you cannot change. NIST SP 800-38G defines the standard modes, and they have documented caveats for small domains, which matter more than they sound: a value drawn from a small set can be attacked by enumeration regardless of the cipher.

For LLM work specifically, FPE is rarely the right safeguard. The model does not validate your card numbers. You are paying key-management cost for a property nothing downstream uses. It belongs in the toolkit for the data-pipeline side of your estate, and is listed here so you recognise it when someone proposes it.

Synthesis

Generate a new record that has the statistical properties of the original and describes nobody.

This is a genuine research direction and a genuine option for training sets, evaluation corpora, and demo environments. It is not a safeguard for a live request, because you cannot synthesise the specific customer's specific problem and still answer it.

Where it does pay off immediately is test data. The `pii-test-data` folder from Chapter 7 is a real liability, and a synthesised equivalent removes it. Treat synthesis as the answer to "how do we build realistic test and demo data without copying production", which is a question most teams have answered badly, and not as an answer to "how do we handle this request".

Choosing a safeguard

Four questions, in order:

1. **Does the task need this value at all?** If no, safeguard 1. Most of the time, for most fields, the answer is no.
2. **Does the output need the real value restored?** If yes, you need a reversible safeguard: 5, 6, or 7. If no, you can use an irreversible one and skip the vault entirely, which is a large simplification.
3. **Does the model need to tell entities apart?** If yes, safeguard 5 or 6. If no, safeguard 2 is simpler and safer.
4. **Will a human read the output before it is trusted?** If no, avoid safeguard 6, because nobody will catch a fabricated value.

Apply per entity, not per system. A single request can legitimately remove the address, pseudonymise the names, generalise the date of birth, and redact the card number, and that combination is better than any uniform policy.

Most production systems answer "yes" to question 2, because the user expects an answer about their actual customer. That means a round trip, and the round trip has sharper edges than the diagram suggests.

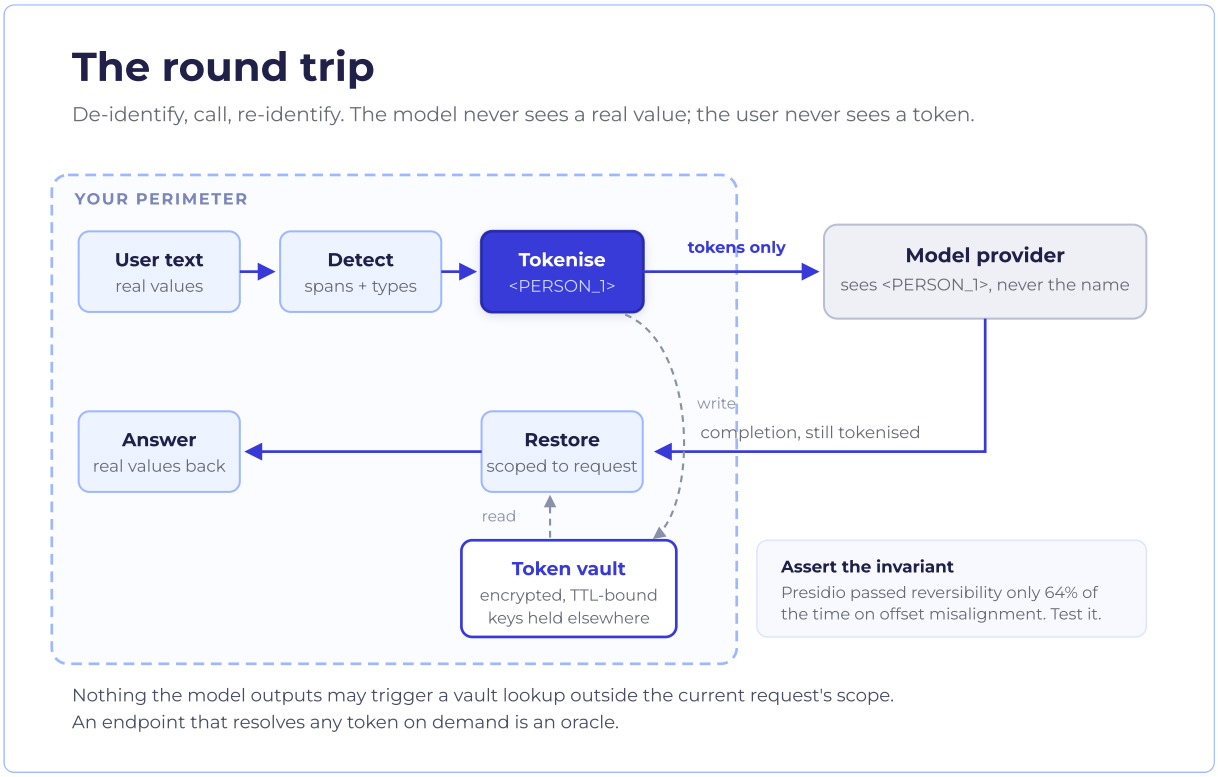
SOURCES FOR THIS CHAPTER

- **Reversible strategies (pseudonymisation, randomisation, masking) outperform irreversible ones (generalisation, redaction); generalisation 0.58 against pseudonymisation 0.70 on MUSR** — T. Deußer, M. Hahnbück, L. Sparrenberg, T. Uelwer, C. Bauckhage, R. Sifa, *On the Impact of Anonymization on the Performance of Large Language Models*, arXiv 2609.11335, 10 September 2026: https://arxiv.org/html/2609.11335 · *Five strategies, eleven benchmarks, five models.*
- **Pseudonymised data remains personal data** — GDPR Recital 26; CCPA "deidentified" standard, cited in Chapter 2.
- **Format-preserving encryption modes and their small-domain caveats** — NIST SP 800-38G.

The ladder of eight safeguards and the four questions for choosing one are the author's framework. The ordering by utility and residual risk is a judgement informed by the Deußer results rather than a reproduction of them. An earlier draft called generalisation the best choice for analytics work; the Deußer measurements contradicted that and the text was corrected.

The Round Trip

De-identify, call the model, re-identify before anyone sees the answer. This is the core production pattern, and this chapter builds it with the failure modes attached.



The shape

```
user text
   ↓
detect  →  replace entities with stable tokens  →  vault stores token → value
   ↓
prompt (tokens only) → model → completion (tokens only)
   ↓
restore tokens from vault (scoped to this request)
   ↓
answer the user sees
```

The model never sees a real name. The user never sees a token. The vault sits on your side of the boundary and is never reachable from anything the model influences.

Presidio gives you both halves. The anonymizer replaces findings with operators of your choosing, and its deanonymize endpoint reverses an encrypted or mapped replacement. You can also hold the mapping yourself, which is what the code below does, because the mapping is the asset and you want it under your own storage and access controls.

De-identifying

Take the findings from Chapter 6, sort them, and replace from the end of the string backwards so earlier offsets stay valid.

```
public sealed record Deidentified(
    string Text,
    IReadOnlyDictionary<string, string> Map);

public static Deidentified Deidentify(
    string text,
    IReadOnlyList<PiiFinding> findings)
{
    var map = new Dictionary<string, string>(StringComparer.Ordinal);
    var counters = new Dictionary<string, int>(StringComparer.Ordinal);
    var byValue = new Dictionary<string, string>(StringComparer.Ordinal);

    var sb = new StringBuilder(text);

    foreach (var f in findings.OrderByDescending(f => f.Start))
    {
        var original = text[f.Start..f.End];

        // Same value gets the same token, so coreference survives.
        if (!byValue.TryGetValue(original, out var token))
        {
            var n = counters.GetValueOrDefault(f.EntityType) + 1;
            counters[f.EntityType] = n;
            token = $"<{f.EntityType}_{n}>";
            byValue[original] = token;
            map[token] = original;
        }

        sb.Remove(f.Start, f.End - f.Start).Insert(f.Start, token);
    }

    return new Deidentified(sb.ToString(), map);
}
```

Three decisions are load-bearing.

Replace from the end. Offsets from the detector refer to the original string. Replacing forwards invalidates every subsequent offset the moment a token is a different length from the value it replaced, which it always is.

Same value, same token. The `byValue` dictionary is what preserves coreference. Without it, `Sarah Whitfield` mentioned three times becomes three different people and the summary is wrong.

Angle brackets, not bare words. `<PERSON_1>` survives a model paraphrasing the text. A bare `PERSON_1` gets reformatted, lowercased, pluralised, or wrapped in quotes, and then your restore pass misses it. Delimiters that are unusual in prose are the difference between a restore that works and one that silently leaves tokens in the output.

Offsets are where this breaks

The Chapter 4 benchmark tested exactly this: redact, then restore, and check you get the original text back. Regex, Piiranha, GLiNER v1 and GLiNER v2 all passed **100%** of the time.

Presidio passed 64%.

The cause was character offset misalignment, arising from how tokenisation maps back to character positions. That is fixable in post-processing, so treat it as a warning about offsets rather than a verdict on Presidio. What it should change is your confidence: assume nothing, and check.

Assert it rather than trusting it:

```
public static string Reidentify(
    string completion,
    IReadOnlyDictionary<string, string> map)
{
    var sb = new StringBuilder(completion);
    foreach (var (token, original) in map)
        sb.Replace(token, original);
    return sb.ToString();
}

// Round-trip invariant: deidentify then reidentify must be lossless.
[Fact]
public void RoundTrip_RestoresOriginalExactly()
{
    const string input = "Contact Sarah Whitfield at swhitfield@example.com about 021000021.";
    var findings = Detector.Detect(input);
    var d = Deidentify(input, findings);

    Assert.Equal(input, Reidentify(d.Text, d.Map));
    Assert.DoesNotContain("Sarah Whitfield", d.Text, StringComparison.Ordinal);
}
```

Run that over your Chapter 7 golden set, not over one example. A 64% pass rate means one in three of your requests comes back mangled, and the visible symptom is a corrupted answer rather than an error, so nothing alerts.

The vault

For a single request, a `Dictionary` on the stack is the vault, and that is the right answer for stateless calls. It lives for the duration of the request and is garbage collected. Nothing persists, so nothing leaks.

Conversations need more, because turn seven must use the same tokens as turn one. That means storage, and storage means the vault becomes an asset.

```
public interface ITokenVault
{
    Task<string> TokenForAsync(
        string conversationId, string entityType, string value, CancellationToken ct);

    Task<IReadOnlyDictionary<string, string>> MapForAsync(
        string conversationId, CancellationToken ct);
}
```

Four properties are not optional.

Scoped. Every operation takes a conversation or request identifier. A vault keyed only by token is a cross-tenant data leak waiting for a token collision, and `PERSON_1` collides immediately because every conversation has one.

Encrypted at rest, with the key held elsewhere. The vault holds the mapping from token to real person, which makes it the most sensitive store you own. Chapter 14 covers this; the short version is that the service holding the ciphertext should not hold the key.

Short-lived. Set a TTL that matches the conversation lifetime and let it expire. There is no business reason to retain a de-identification map for ninety days, and every day it exists is a day it can be stolen.

Unreachable from the model path. Nothing the model outputs should be able to cause a vault lookup outside the current request's scope. This is the one that gets built wrong. A helpful "resolve any token in this text" endpoint, called with model-influenced input, is an oracle that will dump your mapping to anyone who can make the model emit token names.

Restoring safely

Restore only what this request tokenised. The scoping is the control.

```
public async Task<string> CompleteAsync(
    string conversationId, string userText, CancellationToken ct)
{
    var findings = await _detector.AnalyzeAsync(userText, ct: ct);
    var d = await _deidentifier.ApplyAsync(conversationId, userText, findings, ct);

    var completion = await _model.CompleteAsync(d.Text, ct);

    // Scope-limited: only this conversation's map is in play.
    var map = await _vault.MapForAsync(conversationId, ct);
    var answer = Reidentify(completion, map);

    if (TokenPattern.IsMatch(answer))
        _logger.LogWarning("Unresolved token in completion for {Conversation}", conversationId);

    return answer;
}
```

That last check matters. An unresolved token in the output means the model invented a token name, or your map was incomplete, or the restore missed a reformatted token. Users will report it as gibberish. Catch it yourself first.

Do not log `userText`, `d.Map`, or `answer` in that method. Chapter 16 explains why at length; for now, note that the most natural place to add a debug log is the one place that sees both the tokens and the values.

Fail closed

The detector is a network call to a container. Containers restart, get OOM-killed, and hit their connection limits. What your code does in that two-second timeout is a policy decision, and the default that most code falls into is the wrong one.

```
try
{
    findings = await _detector.AnalyzeAsync(text, ct: ct);
}
catch (Exception ex) when (ex is HttpRequestException or TaskCanceledException)
{
    _logger.LogError(ex, "PII detector unavailable");
    throw new DetectorUnavailableException();   // do not proceed
}
```

An empty findings list and a failed call are indistinguishable downstream. A `catch` that logs and continues with zero findings sends the raw text to the model, and it does so precisely when your protection is broken. The system appears healthy, because from the user's perspective the feature is working better than usual.

Failing closed has a cost and you should choose it deliberately. Options, roughly in order of preference:

Reject the request with a clear error. Correct for anything handling sensitive data.

Degrade to deterministic-only and proceed with reduced coverage, logging that you did. Defensible when Chapter 5's layer covers your highest-consequence entities and the remainder is low blast radius. Record the degradation, because it is a fact an auditor will ask about.

Queue for later. Works for asynchronous jobs, useless for interactive chat.

What you never do is proceed as though the check passed. If that means your LLM feature has an availability dependency on your detection container, that is true and you should say so in the design document rather than discovering it during an incident.

Multi-turn drift

One last failure mode, easy to miss and unpleasant in production.

A user says "Sarah Whitfield" in turn one and "Sarah" in turn four. Your detector finds `PERSON` in both, but the strings differ, so `byValue` assigns a second token. The model now believes there are two people.

The fix is normalisation at the vault boundary: before assigning a token, check whether the value is a subset or variant of an existing value in this conversation. Surname-only, given-name-only, and initialised forms cover most real cases.

```
private static bool IsLikelySamePerson(string candidate, string known) =>
    known.Contains(candidate, StringComparison.OrdinalIgnoreCase)
    || candidate.Contains(known, StringComparison.OrdinalIgnoreCase);
```

That heuristic is crude and it is better than nothing. Get it wrong in the merging direction and you conflate two people, which is worse than splitting one, so bias it towards splitting and accept the occasional duplicate.

The round trip works, it is reversible, and it keeps real values away from the model. It also, sometimes, makes the model worse at the job you asked it to do.

SOURCES FOR THIS CHAPTER

- **Presidio's 64% reversibility pass rate, caused by character offset misalignment, against 100% for regex, Piiranha and both GLiNER models** — Sikkema benchmark: https://albertsikkema.com/python/security/privacy/2026/06/01/benchmarking-open-source-pii-detection.html · *This specific result is why the round-trip invariant test exists in this chapter.*
- **Sanitising sensitive prompts with reversible mappings** — *Preempt*, arXiv 2504.05147: https://arxiv.org/pdf/2504.05147
- **Keeping the token vault outside model reach, preserving only the relationships the task needs** — practitioner consensus; see the architecture sources in Chapter 13.

The implementation details are the author's: replacing from the end of the string, the `byValue` *map for coreference, angle-bracket delimiters surviving paraphrase, the four vault properties, and the multi-turn name-variant heuristic. These are engineering positions, argued in the text rather than cited.*

When Masking Breaks the Task

The honest chapter. Some tasks need the real value, and removing it does not produce a refusal. It produces a confident wrong answer.

The failure is silent

A model given `[REDACTED]` where a date should be does not say "I cannot answer without the date." It writes a summary that omits the date, or invents a plausible one, or reasons about a value it does not have. The output looks exactly like a good output.

That is what makes this failure mode dangerous compared to the ones in Chapter 9. A mangled restore produces visible gibberish. A broken task produces a clean, fluent, wrong answer, and nobody investigates a clean answer.

Four patterns account for most of it.

Numeric ambiguity

You redact a number. The model no longer knows what kind of number it was, and neither does your detector.

Consider `47` in a clinical note. It could be an age, a dose in milligrams, a bed number, a systolic reading, or the patient's weight. A detector tuned to catch ages will catch all of them, because at the character level they are identical. Redact them all and the note is useless. Redact none and you leak the age.

This is a genuinely hard problem and there is research on it specifically, in tutoring and clinical contexts, precisely because numbers carry both the identifying information and the task-critical information in the same token.

Two practical moves. **Use context, not the value.** A number preceded by "aged" or followed by "years old" is an age; a number inside a `dose:` field is not. Presidio's contextual scoring does some of this and you can extend it with your own recognisers, which is one more argument for a framework you can plug into. **Generalise rather than redact.** Safeguard 4 from Chapter 8 keeps the number's category and blurs its precision. `47` becomes `40-49`, the note stays readable, the model can still reason about an adult in middle age, and you have removed a strong quasi-identifier.

Coreference collapse

Covered in Chapter 8 and worth the repetition because it is the most common cause of degraded output.

Redaction maps many people onto one marker. An email thread with four participants becomes a thread about `[PERSON]` talking to `[PERSON]` about `[PERSON]`. Any summary of it is fiction.

Pseudonymisation fixes this and is why it is the default. `<PERSON_1>` through `<PERSON_4>` keeps the relationships intact. If you are using redaction on anything conversational, this is almost certainly why your summaries are poor, and the fix is a safeguard on the ladder rather than a better prompt.

Signal in the name

Sometimes the identifier is the thing the task is about.

A model asked to detect potential bias in a hiring process needs to see the names, because the names are the variable under examination. A model asked to check whether a translation preserved a person's name needs the name. A model routing correspondence by language may be using the name as a signal.

Notice that the first example is also the classic argument *for* masking: Microsoft's own documentation lists masking identifiers in resume screening to reduce bias as a use case, and it is a good one. Both are true. **Blind the screening decision, and show the names to the audit that checks whether the screening was biased.** Same data, opposite safeguards, because the tasks are different.

This is the general principle: decide per task, never per system. A single global redaction policy will be wrong for some task in your application, and it will be wrong silently.

Plausibility hazard

Surrogates preserve utility better than tokens, and they introduce a risk that tokens do not.

`<PERSON_1>` is obviously not a person. `Megan Alvarez` is obviously a person, and is not the person. If a surrogate escapes the boundary where values are restored, a human downstream has no way to tell, and will act on it as real. The failure is not that someone is careless. It is that a well-chosen surrogate is designed to be indistinguishable, and you have removed the signal a reader would need in order to doubt it.

If you use surrogates, three rules.

The restore boundary must be explicit and total. One place in the code restores values, and nothing renders model output that has not passed through it.

Anything that escapes must be labelled. If a surrogate can reach a log, an export, a webhook, or a downstream system, it must carry a marker saying so.

Never use surrogates where the output feeds an action. Summaries for humans, yes. Arguments to a tool that sends an email, no. Chapter 15 has more on why model-composed values reaching actions is its own category of problem.

Deciding, per task

Work through four questions for each LLM feature. They take ten minutes and they prevent the silent-wrong-answer class entirely.

What does the task actually operate on? Write the sentence. "Classify the urgency of this ticket from its text." Urgency comes from the complaint, not the complainant. The name can go.

Which entities are load-bearing? For each entity type your detector catches, ask whether removing it changes the answer. Most of the time, for most entity types, it does not. When it does, you have found the ones that need a reversible safeguard rather than removal.

Can a generalisation carry the signal? Usually yes, and this is the most under-used answer in the book. The task rarely needs the exact date of birth, exact salary, exact ZIP code, or exact timestamp. It needs the decade, the band, the region, the month.

What happens if the model gets it wrong? If output goes to a human who can check it against the source, a degraded answer is an annoyance. If it triggers an action, a degraded answer is an incident, and you should be much more conservative about masking anything the decision depends on.

Measure the degradation

There is now measured evidence on how much this costs, and the headline is that it varies far more than you would guess. Deußer et al. (2026) ran five anonymisation strategies across eleven benchmarks and five models. Three findings are worth carrying into your own design.

The damage is task-specific, not uniform. On RGB, a retrieval-grounded benchmark, scores fell by between **0.22 and 0.47**. On TruthfulQA, anonymisation *improved* results across every model tested, with Llama-3.1 gaining **8 percentage points**, apparently because stripping named entities stopped the model reaching for incorrect memorised associations. The same safeguard helped one task and gutted another.

More capable models lose more. Qwen2.5-72B degraded by **6.9 percentage points** on average against **2.3** for the much smaller Teuken-7B. The stronger model was relying on entity knowledge that anonymisation removed. Upgrading your model does not buy you headroom here, and may cost you some.

Telling the model does not help. Adding a prefix explaining that the text has been anonymised produced no significant or consistent improvement. If you were planning to solve this with prompt engineering, that is the experiment already run.

Which leads to the one thing you should actually do: **evaluate your task with and without the safeguard applied.**

Take fifty real examples. Run them through the model with the original text and with the de-identified text. Compare the outputs. Not with an automated metric, at least not at first; read them side by side. You will find out in an afternoon whether your safeguard costs you nothing, costs you a little, or has quietly broken the feature.

```
[Fact]
public async Task Deidentification_DoesNotDegradeSummaryQuality()
{
    foreach (var sample in EvalSet.Load(50))
    {
        var raw = await _model.SummariseAsync(sample.Text);
        var treated = await _model.SummariseAsync(Deidentify(sample.Text).Text);

        // Judged offline and recorded; this asserts the artefact exists,
        // the comparison itself is a human review.
        EvalReport.Record(sample.Id, raw, treated);
    }

    Assert.True(EvalReport.AgreementRate >= 0.90);
}
```

Teams run detector accuracy tests and almost never run task quality tests. The result is a privacy control that measurably works and a feature that quietly stopped working, and the second one is what users experience.

When the answer comes back that the task genuinely needs the real value, do not go looking for a cleverer mask. There is no mask that both hides a value and preserves it. What you need is a different design, where the sensitive value never needed to be in the prompt in the first place.

That is the rest of the book.

SOURCES FOR THIS CHAPTER

- **Task-specific degradation (RGB −0.22 to −0.47), TruthfulQA improving under anonymisation (Llama-3.1 +8pp), larger models degrading more (Qwen2.5-72B −6.9pp against Teuken-7B −2.3pp), and anonymisation-awareness prompt prefixes producing no consistent improvement** — Deußer et al., *On the Impact of Anonymization on the Performance of Large Language Models*, arXiv 2609.11335, 10 September 2026: https://arxiv.org/html/2609.11335
- **Numeric ambiguity in de-identified text** — *Utility-Preserving De-Identification*, arXiv 2602.16571: https://arxiv.org/pdf/2602.16571
- **Masking identifiers in resume screening to reduce bias, as a vendor-documented use case** — https://learn.microsoft.com/en-us/azure/ai-services/language-service/personally-identifiable-information/overview

The plausibility hazard, the four per-task questions, and the rule against surrogates where output feeds an action are the author's argument. No incident data is cited for surrogates reaching end users, and the text does not assert that it has happened.

The Architecture That Holds

Part II said detection is a net. This chapter draws the wall: the complete reference architecture, what to build in what order, and an honest answer to the question everyone asks, which is whether any of this gets you to 100%.

The honest answer about 100%

There is exactly one guarantee in this book that is absolute, and it is narrow:

> ***A value that is structurally incapable of entering the payload cannot reach the model.***

Not "unlikely to." Cannot. If your projection record has four fields and none of them is the customer's Social Security number, then no detector threshold, no container restart, no model upgrade and no clever prompt can cause that number to reach the provider from that code path. The guarantee comes from the shape of the code, not from the accuracy of a classifier.

Everything else in this book is probability.

So the architecture is not a machine that makes personal data safe. It is a machine that **moves as much of your data as possible into the category where the guarantee is structural**, handles the next category deterministically, and leaves you with a small, named, measured residue that you manage rather than eliminate.

That reframing is the whole chapter. Once you stop asking "how do I catch all the PII" and start asking "how much of this can I make structurally impossible to send," the problem becomes tractable, and you can say something precise about what is left.

The three tiers

Every field, every value, every payload in your system falls into one of three tiers. The architecture's job is to push things upward.

TIER	WHAT IT IS	GUARANTEE	MECHANISM
1. Never sent	Fields the task does not need	**Absolute**	Type system, projection records
2. Deterministically caught	Structured identifiers with checksums	**Near-absolute, per entity class**	Validators, in-process, on every payload
3. Best effort	Free text a human typed	**~0.5 F1, and no better**	Model-based detection

Tier 1 is the only place a guarantee lives. Chapter 12 is entirely about getting fields into it, and it is the cheapest work in the book.

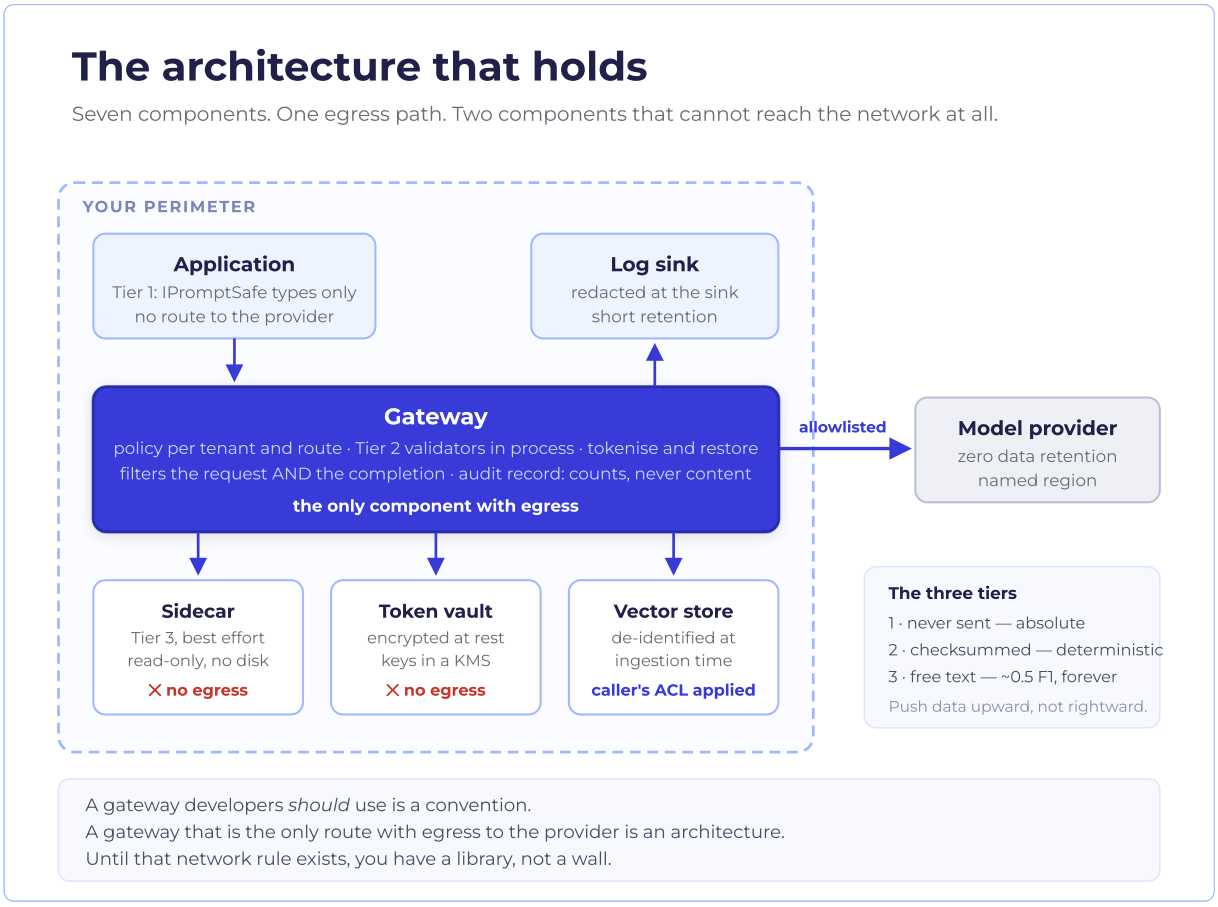
Tier 2 is deterministic but bounded. A Luhn-validated card number will be caught every time it appears in a well-formed state. That is a real guarantee, and its boundary is exact: if the value is malformed, obfuscated, split across lines, or spelled out in words, the validator does not fire. The guarantee is "this class of well-formed identifier, always," not "all payment data, always."

Tier 3 has no guarantee and never will. A customer writing "my wife Karen has the same condition as her mother" in a support ticket has disclosed health data about two identified people, in prose, with no pattern. There is no detector configuration that reliably catches every instance of that, and Chapter 4 is the evidence. This tier is managed by making the blast radius small, not by making the detection good.

Everything below is in service of that table.

The reference architecture

Seven components. Five trust boundaries. One direction of travel.



Four properties make this an architecture rather than a pile of controls.

The application cannot reach the provider. Not by policy, by network rule. This is the single most important line in the diagram, and it is the one most often missing. A gateway that developers *should* use is a convention. A gateway that is the only route with egress to `api.provider.com` is an architecture.

The sidecar has no egress at all. It receives text and returns spans. It does not need the internet, so it does not get the internet.

The vault is reachable only from the gateway, and it holds ciphertext whose key lives in a KMS the vault cannot read.

Filtering happens in both directions at the same chokepoint, which is what closes the output door.

What runs where

```
# Topology, expressed as network zones. The zones are the control.
zones:
  app:
    workloads: [web, api, workers]
    egress: [gateway]                   # and nothing else

  detection:
    workloads: [gateway, sidecar, vault]
    egress:
      gateway: [provider-allowlist, vault, sidecar, log-sink]
      sidecar: []                       # deliberately empty
      vault:   []                       # deliberately empty

  provider-allowlist:
    hosts: ["api.openai.com"]           # one entry, reviewed
```

If you take one thing from this chapter into your next design review, take the two empty lists. A component that cannot make outbound connections cannot exfiltrate, regardless of what a dependency does or what an injected instruction asks for. It is the cheapest hard boundary available to you.

The build order, and what each step buys

Do these in order. The ordering is deliberate: each step reduces the work the next one has to do.

1. Classify (a day). Walk every LLM call. For each field, decide its tier. Write the table down. You cannot design the architecture before you know what is in Tier 3, because Tier 3 sizes everything else.

2. Push everything possible into Tier 1 (days). Projection records, the `IPromptSafe` marker, out-of-band slots. Chapter 12. This is where the absolute guarantee comes from and it costs less than any other step.

3. Build the gateway as a network boundary (a week). Chapter 13. Policy, audit record, failure mode, both-direction filtering. Then change the egress rules so applications cannot reach the provider directly. **Step 3 is not finished until that rule exists**, because until then you have a library, not a wall.

4. Add Tier 2 validators in-process (days). Chapter 5 and Appendix A. They run at 0.1 ms, so they run on every payload, every log line, every tool-call argument. They also become your degraded mode when step 5 is unavailable.

5. Deploy the sidecar, hardened from the start (days). Chapter 14. TLS, no disk, no egress, non-root, pinned digest, logging off. Retrofitting those is a migration.

6. Wire the round trip and the vault (a week). Chapter 9. Stable tokens, scoped restoration, round-trip invariant test, fail closed.

7. Close the remaining doors (days). Log sink redaction and retention, provider contract terms, and if you have RAG, ingestion-time de-identification plus permission-filtered retrieval. Chapters 15 and 16.

8. Measure and prove (days). Golden set in CI, audit record on, task quality comparison. Chapter 17.

A competent team does this in about a month. Chapter 17 lays it out as four weeks.

What this architecture actually guarantees

Precision matters here, because this is the paragraph people will quote back at you in a security review.

It guarantees, absolutely:

- Fields excluded by projection never reach the provider from the paths that route through the gateway.
- The application cannot reach the provider by any other route, if the egress rule is in place.
- The sidecar and vault cannot initiate outbound connections.
- Real values for tokenised entities never leave your perimeter; only tokens do.
- Every model call has an audit record naming the policy version in force.

It guarantees, deterministically but within a bounded class:

- Well-formed checksummed identifiers are caught on every payload the gateway sees.

It does not guarantee, and cannot:

- That personal data in free text is detected. Around half of it will be, on the evidence in Chapter 4.
- That the model does not infer personal data that was never in the input.
- That a person with database access cannot read the same data directly. This architecture addresses accidental egress to a model, not insider access.
- That the provider honours its contract. You have a contractual control, not a mechanism.
- Anything at all about code paths that do not route through the gateway.

That last one is the failure mode in practice. The architecture covers what it covers.

The caveats, stated plainly

The gateway is bypassable until the network says otherwise. A developer with an API key and a `HttpClient` can call the provider from anywhere. Egress control is the only thing that converts intent into enforcement, and it is the step most often skipped because it needs the platform team.

Every new service is a hole until it is routed. Six months from now someone will build a feature, add the SDK, and ship. Your defence is that the API key is not available outside the detection zone, and that the egress rule denies by default. Make obtaining a provider credential a deliberate act.

The vault concentrates risk. You have taken a diffuse problem and put the mapping from token to person in one place. That is easier to defend and worse when it fails. Short TTLs and key separation are what make the trade worth it.

Tier 3 drift is invisible. Your detector's real performance changes when your traffic changes, and nothing alerts. The golden-set test in CI is the only thing that surfaces it.

Latency and availability are now coupled. Your LLM feature depends on a container. Fail closed and you have an availability dependency. Fail open and you have no control. Choose per route, write it in policy, and count the degradations.

None of this makes you compliant. It produces the evidence a compliance process needs. The process is still someone's job.

So what is the actual 100% solution?

You asked, so here it is without hedging.

The only configuration in which no third party receives your personal data is one where no third party is involved. Self-host the model. Run inference inside your perimeter, on hardware you control, with the same egress rule applied to the inference service that you applied to the sidecar. Then the provider row disappears from the diagram, and the whole class of "what does the vendor retain" questions disappears with it.

This is a real option, and for some organisations it is the right one. Open-weight models have closed much of the capability gap for the bounded tasks that most enterprise features actually perform: classification, extraction, summarisation, routing. If your workload is one of those and your data is regulated, self-hosting is worth pricing seriously rather than dismissing.

Three honest caveats, because this is not a free win.

You trade a data risk for an operating cost. GPUs, capacity planning, model updates, evaluation, an on-call rotation for an inference service. That is a standing engineering commitment, not a project.

The capability gap is real at the frontier. For hard reasoning, long context, and tool use, the hosted frontier models remain ahead. If your feature needs that, self-hosting costs you quality.

It closes exactly one door. Look back at Chapter 3. Self-hosting removes the provider from the prompt door. Ingestion, tool calls, logs and output are all still open, and they are all inside your perimeter. Your vector store still holds customer text. Your logs still capture prompts. Your agent can still be injected into calling a tool that emails someone. **Self-hosting is not an alternative to this architecture. It is one substitution inside it**, and every other component stays exactly where it is.

Which is the closing point, and the reason this chapter exists. There is no configuration that gets you to 100% on all five doors, because door three is a human typing prose and prose is unbounded. There is a configuration that gets you to 100% on the data you chose not to send, near-100% on the identifiers you can validate, and a measured, named, survivable number on everything else.

That is the wall. The next five chapters build it, one component at a time.

SOURCES FOR THIS CHAPTER

This chapter is synthesis rather than new evidence. Every quantitative claim restates something sourced elsewhere in the book.

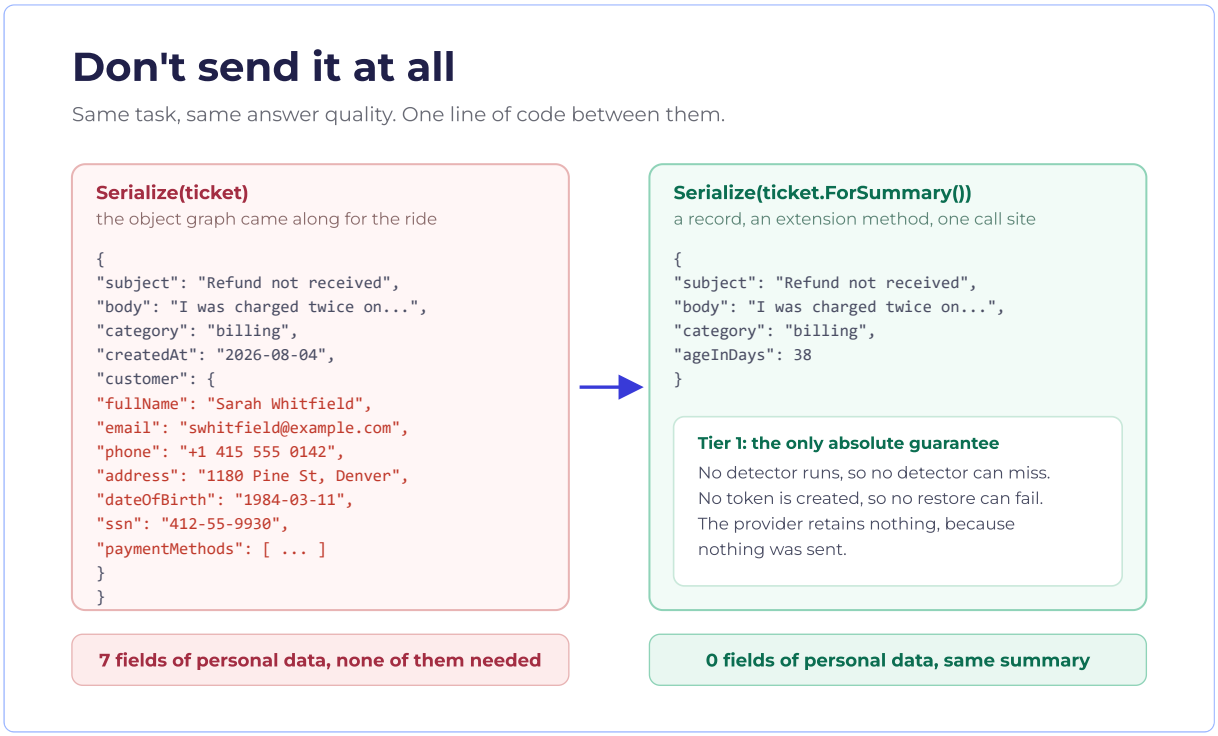
- **Cross-domain detection landing around 0.5 F1, and the Tier 3 ceiling** — Sikkema benchmark and PIIBench, both cited in full in Chapter 4: https://albertsikkema.com/python/security/privacy/2026/06/01/benchmarking-open-source-pii-detection.html · https://arxiv.org/abs/2605.25816
- **Deterministic validator latency (0.1 ms) and the checksum algorithms behind Tier 2** — cited in Chapter 5 and Appendix A.
- **Egress control, capability limitation and provenance-aware deployment as the mitigations that bound blast radius** — named across the agent-security literature cited in Chapter 15: https://arxiv.org/pdf/2506.01055 · https://arxiv.org/pdf/2604.05432
- **The chokepoint-with-versioned-policy pattern, and filtering both directions** — practitioner consensus, sources listed in Chapter 13.
- **Provider zero data retention and regional processing as contractual rather than mechanical controls** — cited in Chapter 16.

Everything structural in this chapter is the author's argument: the three-tier model, the claim that only Tier 1 carries an absolute guarantee, the seven-component reference architecture, the build order, and the guarantee/non-guarantee lists. These are engineering positions, not findings, and they are offered so you can disagree with them specifically rather than generally.

The self-hosting section makes no quantitative claim about the capability gap between open-weight and frontier models, deliberately. Any number there would be stale within a quarter, and Chapter 4's argument about decontextualised benchmark figures applies with equal force to model leaderboards.

Don't Send It At All

The cheapest control in this book, and the one teams reach for last. A field you never send cannot be missed by a detector, restored incorrectly, retained by a provider, or recovered from a vector.



Why the whole object gets sent

Nobody decides to send a postal address to a language model. It arrives like this:

```
var prompt = $"Summarise this support ticket:\n{JsonSerializer.Serialize(ticket)}";
```

One line. It works. It took eight seconds to write. And `ticket` has a `Customer` navigation property, which has an `Address`, and `Customer` has a `PaymentMethods` collection that Entity Framework happened to load because something earlier in the request touched it.

The alternative looked like eleven lines and a new type, so nobody wrote it. That is the entire mechanism. Not negligence, not ignorance, just the relative friction of two options at the moment someone was trying to finish a feature.

The fix has to be cheaper than the leak, and fortunately it is.

The projection

```
public sealed record TicketForSummary(
    string Subject,
    string Body,
    string Category,
    int AgeInDays);

public static TicketForSummary ForSummary(this Ticket t) => new(
    t.Subject,
    t.Body,
    t.Category,
    (int)(DateTime.UtcNow - t.CreatedAt).TotalDays);
```

```
var prompt = $"Summarise this support ticket:\n{JsonSerializer.Serialize(ticket.ForSummary())}";
```

That is the control. A record, an extension method, and a call site that looks almost identical to the one it replaces.

What it bought you: the customer's name, email, phone, address, account number and payment methods are now structurally incapable of reaching the model from this code path. Not filtered, not detected, not masked. Absent. No detector runs, so no detector can miss them. No token is created, so no restore can fail. The provider retains nothing, because nothing was sent.

Compare that to the alternative you were considering, which was a detector averaging 0.48 F1 on cross-domain text, called over a network, with a failure mode you have to design for.

Don't detect what you can refuse to send.

Make it the type system's job

An extension method is a convention, and conventions decay. The next developer writes a new endpoint and serialises the entity again, because the entity is right there.

Make the compiler enforce it.

```
// Marker for types explicitly reviewed and approved for prompt inclusion.
public interface IPromptSafe;

public sealed record TicketForSummary(...) : IPromptSafe;

public sealed class PromptBuilder
{
    public string Build<T>(string instruction, T payload) where T : IPromptSafe =>
        $"{instruction}\n{JsonSerializer.Serialize(payload)}";
}
```

Now `Build(instruction, ticket)` does not compile. The developer has to define a projection and think, for about thirty seconds, about which fields belong in it. Thirty seconds at the right moment is worth more than any amount of runtime filtering.

This is the pattern's real value. It moves the decision from "did anyone remember to redact" to "you cannot express the unsafe version", and it does so without a runtime dependency, a container, or a latency budget.

Out of band

The second half of minimisation. Sometimes the model needs to refer to a person without needing to know who they are.

The instinct is to put the identifier in the prompt so that the answer can reference it. You do not need to.

```
// The model works on a reference. Your code resolves it.
var prompt = $"""
    Draft a reply to the customer for ticket {ticket.Reference}.
    Address them as {{CUSTOMER_NAME}}.
    Issue: {ticket.Body}
    """;

var draft = await _model.CompleteAsync(prompt, ct);

// Resolution happens here, on your side, after the model is done.
var reply = draft.Replace("{{CUSTOMER_NAME}}", customer.FullName);
```

The model composes a letter with a slot in it. Your code fills the slot. The name never enters the prompt, never enters the provider's logs, never enters your prompt cache, and cannot be recovered from anything the model touched.

This generalises well. Any value the model needs to *place* rather than *reason about* can be a slot: names, account numbers, dates of appointments, amounts, links. Values the model needs to *reason about* cannot, and Chapter 10 is about telling the difference.

A caution: the slot marker must be something the model reproduces verbatim and will not helpfully expand. Double-brace tokens work because they are common in templating and models leave them alone. If the model starts filling in slots itself, tighten the instruction and validate the output.

Structure beats prose

A third minimisation that costs nothing. When you have structured data, send it as structure.

The bad version pastes a rendered document, an email chain, or an exported PDF into the prompt. Everything in that document goes, including the signature blocks, the disclaimer footers, the routing headers, the previous six replies, and whatever was in the quoted text at the bottom that nobody has read since March.

The good version extracts the fields the task needs and sends those. It is less convenient and dramatically smaller, which has the pleasant side effect of reducing cost and latency alongside exposure.

If you must send free text, send the relevant span rather than the whole document. The last message in a thread is usually the only one the task needs.

Ask whether the field is needed, out loud

The single most effective habit is embarrassingly low-tech: when adding a field to a prompt, say why.

Keep it in the code, where it stays honest:

```
public sealed record CandidateForScreening(
    string Skills,          // needed: the task scores against required skills
    string Experience,      // needed: years and relevance
    string EducationLevel   // needed: minimum qualification check
    // deliberately excluded: name, email, phone, address, photo, date of birth,
    // nationality, gender. None affects a skills match. Excluding them also
    // removes the most common bias vectors from the screening decision.
);
```

That comment is an artefact. It is your data protection impact assessment in the place where it is most likely to be maintained, it is the answer when an auditor asks how you determined what was necessary, and it is a standing argument with the next developer who wants to add a field for convenience.

What this actually eliminates

Run the exercise across your own application and count. This book will not give you a multiplier, because no study measures one and inventing a number here would undercut the argument of Chapter 4. What the exercise reliably surfaces is a category of finding rather than a ratio: fields that reach the model because an object was serialised, not because the task needed them. Count yours, and the count becomes the business case for the hour it takes to fix.

CONTROL	EFFORT	WHAT IT REMOVES
Field projection	Hours	Every entity in the fields you drop, structurally
`IPromptSafe` marker	Hours, once	Future regressions
Out-of-band slots	Hours per feature	Names and identifiers from the prompt entirely
Structure over prose	Days	Everything incidental in documents
Detector at 0.48 F1	Weeks, plus ongoing operations	About half of what remains

The detector is the most expensive row and the least effective. That is not an argument against it, because the remainder it catches is real and includes the free-text disclosures no projection can prevent. It is an argument for the order of operations. Minimise first, measure what is left, then size the detection layer against the residue rather than against the original problem.

Minimisation also happens to be exactly what GDPR Article 5 asks for. Data must be adequate, relevant, and limited to what is necessary. A field projection with a comment explaining each inclusion is that obligation, expressed in C#.

One weakness: minimisation is a decision each developer makes, hundreds of times, in dozens of places. Conventions decay, marker interfaces get bypassed with a quick `record` that implements the interface and carries everything, and someone always ships a hotfix at five o'clock on a Friday.

Which means you need one place where every model call passes through, whatever the calling code intended.

SOURCES FOR THIS CHAPTER

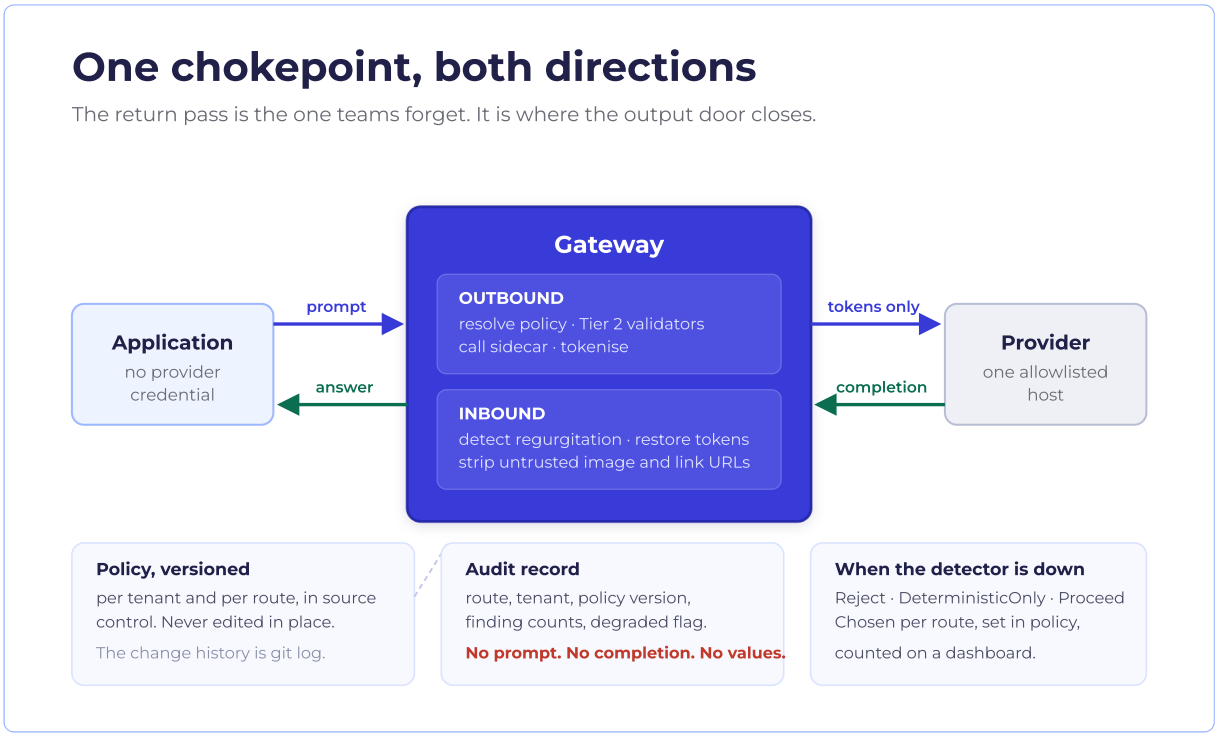
- **Data minimisation: personal data must be adequate, relevant and limited to what is necessary** — GDPR Article 5(1)(c); equivalent purpose-limitation duties under the CCPA and the other state laws cited in Chapter 2.
- **Minimise at source before applying detection** — practitioner consensus across the architecture sources listed in Chapter 13.

This chapter deliberately contains no statistics. An earlier draft carried a multiplier for how much excess personal data a typical LLM feature sends. It was removed because no study measures it, and inventing one would contradict Chapter 4. The `IPromptSafe` marker, out-of-band

slots and the justification comment are patterns offered as engineering arguments.

The Gateway

One chokepoint that every model call passes through, with policy as versioned configuration rather than scattered conditionals. This is where the controls stop being a convention.



Why a chokepoint

Chapter 12's controls are excellent and they are distributed. Every developer, every feature, every code path has to get them right, and the number of code paths only goes up.

A gateway inverts that. Model calls go through one component. That component applies policy. A new feature gets the controls by default, and a developer who wants to bypass them has to work at it visibly rather than accidentally.

It also gives you three things that are hard to retrofit: a single place to change policy without redeploying every service, a single place where every call is observable, and a single answer to "what controls were in force on 3 March?" That last one is Chapter 17's entire subject, and it is almost impossible to produce from scattered per-service logic.

The shape in ASP.NET Core

The gateway is a small service. Applications call it instead of calling the provider.

```
public sealed class ModelGateway(
    IDetectionPipeline detection,
    IPolicyStore policies,
    IModelProvider provider,
    ILogger<ModelGateway> logger)
{
    public async Task<GatewayResult> CompleteAsync(
        GatewayRequest request, CancellationToken ct)
    {
        var policy = await policies.ResolveAsync(
            request.Tenant, request.Route, ct);

        var inbound = await detection.ApplyAsync(
            request.Prompt, policy.Inbound, ct);

        var completion = await provider.CompleteAsync(
            inbound.Text, request.Options, ct);

        var outbound = await detection.ApplyAsync(
            completion, policy.Outbound, ct);

        return new GatewayResult(
            Text: outbound.Text,
            PolicyVersion: policy.Version,
            InboundFindings: inbound.Findings.Count,
            OutboundFindings: outbound.Findings.Count);
    }
}
```

Four things in that method are the chapter.

Policy resolved per tenant and per route

A single global policy forces a bad compromise. The route that summarises internal changelogs and the route that processes patient correspondence do not need the same safeguard, and applying the strict policy everywhere degrades features that did not need it, which is how people end up routing around the gateway.

```
public sealed record DetectionPolicy(
    string Version,
    IReadOnlyList<string> Entities,
    double ScoreThreshold,
    SafeguardKind Safeguard,
    FailureMode OnDetectorUnavailable);

public sealed record GatewayPolicy(
    string Version,
    DetectionPolicy Inbound,
    DetectionPolicy Outbound);
```

Policies live in configuration, versioned in source control, deployed independently of application code. Tightening a threshold becomes a reviewed pull request against a policy file rather than a change to six services.

Give every policy an immutable version string. Never edit a policy in place; publish a new version. The version is what makes the audit trail meaningful.

Both directions

The outbound pass is the one teams forget, and it closes door five from Chapter 3.

Three things can be in a completion that were not in the prompt. The model can **regurgitate** data that retrieval pulled in, which is Chapter 15's problem arriving at your door. It can **infer** something about a person, and an inference about an identified person is personal data you generated. And it can carry **rendered exfiltration**, which is what EchoLeak exploited: markdown that causes a client to fetch an attacker-chosen URL with data in the query string.

The third one is not a PII detection problem and should not be handled by the detector. It is a rendering problem, and it is handled with an allowlist:

```
private static readonly Regex MarkdownImage =
    new(@"!\[[^\]]*\]\(([^)]+)\)", RegexOptions.Compiled);

public static string StripUntrustedImages(string markdown, HashSet<string> allowedHosts)
{
    return MarkdownImage.Replace(markdown, m =>
    {
        var url = m.Groups[1].Value;
        return Uri.TryCreate(url, UriKind.Absolute, out var uri)
               && allowedHosts.Contains(uri.Host)
            ? m.Value
            : "[image removed]";
    });
}
```

Apply the same thinking to links, to HTML if you render any, and to anything else in your output that causes a client to make a request. The rule is that model output is untrusted input to your renderer, because something upstream of the model may have been controlled by someone else.

Failure is a policy decision

Chapter 9 covered failing closed at the call site. At the gateway it becomes explicit and configurable, which is better, because the right answer differs by route.

```
public enum FailureMode
{
    Reject,              // return an error; nothing reaches the provider
    DeterministicOnly,   // proceed with Chapter 5's in-process layer only
    Proceed              // requires an explicit, recorded exception
}
```

```
catch (DetectorUnavailableException)
{
    switch (policy.Inbound.OnDetectorUnavailable)
    {
        case FailureMode.Reject:
            throw;

        case FailureMode.DeterministicOnly:
            logger.LogWarning(
                "Detector unavailable; degraded to deterministic layer on {Route} under policy {Version}",
                request.Route, policy.Version);
            inbound = _deterministic.Apply(request.Prompt, policy.Inbound);
            break;

        case FailureMode.Proceed:
            logger.LogError(
                "Detector unavailable; proceeding UNFILTERED on {Route} under policy {Version}",
                request.Route, policy.Version);
            inbound = DetectionResult.Unfiltered(request.Prompt);
            break;
    }
}
```

`Proceed` exists because some routes genuinely carry no sensitive data and should not have an availability dependency on the detector. Making it a named enum member with an error-level log means choosing it is a decision someone made, recorded in policy, visible in configuration review, and countable in your telemetry. That is very different from a `catch` block that silently does the same thing.

Count the degradations. A dashboard showing "requests processed in degraded mode" is the difference between knowing your control was off for six hours and finding out during an audit.

What the gateway records

This is the artefact Chapter 17 needs, so define it deliberately.

```
public sealed record GatewayAuditRecord(
    Guid RequestId,
    string Tenant,
    string Route,
    string PolicyVersion,
    string ProviderModel,
    int InboundFindingCount,
    IReadOnlyDictionary<string, int> InboundFindingsByType,
    int OutboundFindingCount,
    bool Degraded,
    DateTimeOffset At);
```

Read that list for what is missing. The prompt is not in it, and neither is the completion, the detected values, or the token map. **The record describes what happened to a request without reproducing a single thing the request contained.**

Counts and types, not content. "Route `ticket-summary`, policy `v14`, 3 `PERSON` and 1 `US_SSN` found inbound, 0 outbound, not degraded." That record proves the control ran, proves which policy governed it, and is itself harmless. An audit log that contains the

personal data it is auditing has reproduced the problem with a longer retention period, and it will be the thing you are asked about.

If you truly need content for debugging, put it behind a time-boxed, per-route, off-by-default flag with its own retention, and treat it as the sensitive store it is.

Do not use an LLM as the detector

It comes up in every design review, so here is the answer.

Using a second model call to find personal data in the first prompt is slower by orders of magnitude, non-deterministic, more expensive per request, and unauditable, because you cannot explain why it did or did not fire on a given input. It also sends the sensitive text to a model, which is what you were trying to avoid, with the additional property that the detection call is now itself subject to prompt injection.

The practitioner consensus across every independent source is consistent: detection is NLP and pattern matching, running deterministically and in your perimeter. Use a model for guardrail classification of intent if you like. Do not use one to find an IBAN.

Where it sits

The gateway should be a separate service rather than a library, for one reason: a library can be bypassed by not referencing it, and a network boundary can be enforced. If your applications can only reach the model provider through the gateway, because egress rules say so, then the control is architectural rather than advisory.

That is worth the operational cost for anything handling regulated data. For a small team with one application, a library with the `IPromptSafe` marker from Chapter 12 and a well-tested pipeline is a reasonable place to start, as long as you know which property you gave up.

You have now routed every piece of sensitive text in the organisation through two components you operate: a gateway and a detection container. Those components see everything. The next chapter is about what makes them defensible.

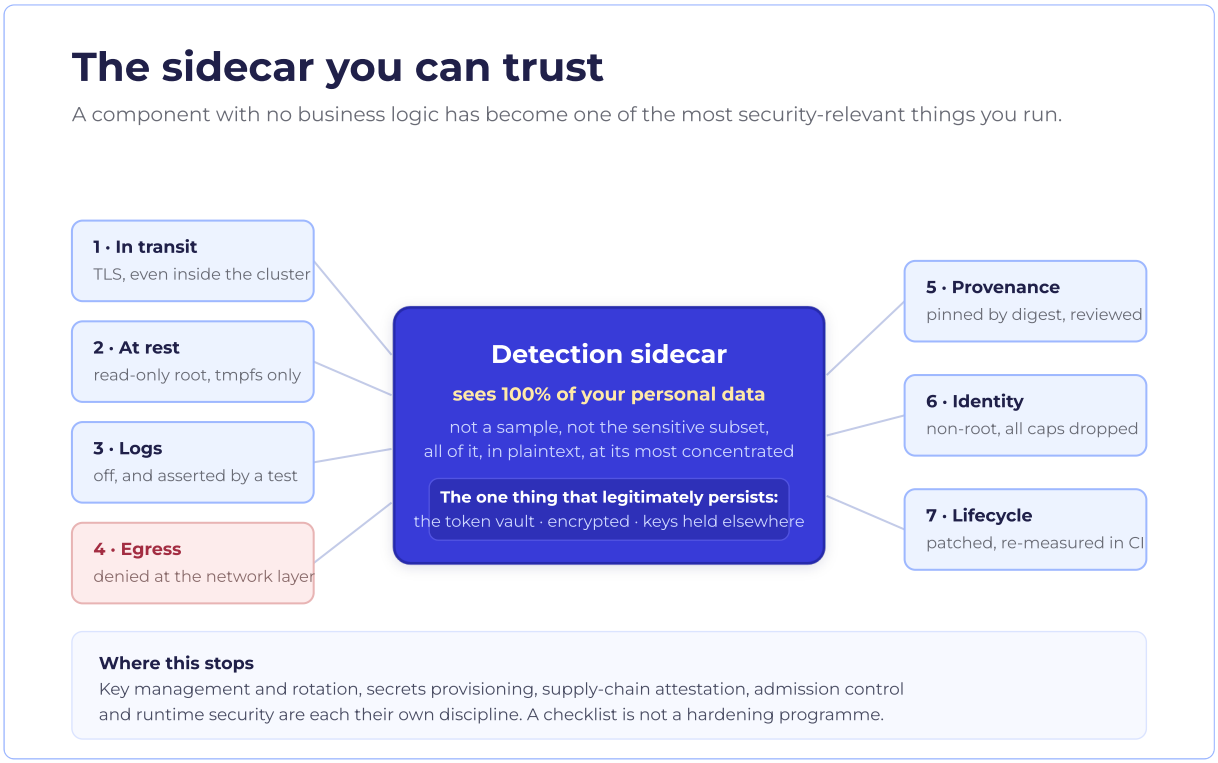
SOURCES FOR THIS CHAPTER

- **Layered detection, redaction on both request and response, self-hosted redaction driven by versioned policy, keeping the vault outside model reach, and not using a second LLM as the detector** — consistent practitioner guidance across independent sources: https://techcommunity.microsoft.com/blog/azuredevcommunityblog/introducing-pii-shield-a-privacy-proxy-for-every-llm-call/4514726 · https://blog.logrocket.com/build-local-ai-proxy-redact-pii-before-llms/ · https://predictionguard.com/blog/pii-detection-redaction-llm-pipelines-regulated-industries · https://www.gravitee.io/blog/how-to-prevent-pii-leaks-in-ai-systems-automated-data-redaction-for-llm-prompt
- **Rendered exfiltration via markdown images, and EchoLeak (CVE-2025-32711)** — https://wraith.sh/learn/markdown-image-exfiltration
- **Immutable audit logs mapped to NIST AI RMF and OWASP** — named in the practitioner guidance above.

The policy shape, the three-member `FailureMode` enum and the content-free audit record are the author's design. The argument that a gateway should be a network boundary rather than a library is a judgement about enforceability.

The Sidecar You Can Trust

You have just centralised every piece of sensitive text in the organisation into one container. This chapter is what makes that defensible, at the level of principle and checklist. It is a summary, not a hardening manual, and the boundary is stated at the end.



The uncomfortable property

The detection service sees **100% of your personal data**. Not a sample, not the sensitive subset, all of it, because it has to read everything to decide what is sensitive.

Every other component in your estate sees a slice. The sidecar sees the whole thing, in plaintext, at the moment it is most concentrated. That inverts the usual risk calculation. A component with no business logic, no database, and no user interface has become one of the most security-relevant things you run.

Seven controls follow from that, in rough order of how much they matter.

1. In transit

Use TLS between the gateway and the sidecar, on a private network, inside the same cluster, without exception. The word doing the work in that sentence is the last one.

The objection is that internal traffic is already isolated, and the objection has been wrong for about a decade. Cluster networks are shared, service meshes route through infrastructure you do not own, cloud provider networking has had cross-tenant issues, and "internal" is a statement about intent rather than about who can observe a packet.

```
builder.Services.AddHttpClient<PresidioAnalyzer>(c =>
{
    c.BaseAddress = new Uri("https://presidio-analyzer:3000");
})
.ConfigurePrimaryHttpMessageHandler(() => new SocketsHttpHandler
{
    SslOptions = new SslClientAuthenticationOptions
    {
        // Pin to your internal CA; do not fall back to the system trust store.
        RemoteCertificateValidationCallback = InternalCa.Validate
    }
});
```

Where your platform offers mutual TLS cheaply, take it, because it also authenticates the caller and control 6 needs that anyway. Where it does not, one-way TLS plus a shared secret is an acceptable stopping point.

2. At rest: persist nothing

The sidecar should write nothing to disk. No request bodies, no temporary files, no spooled uploads, no model cache containing user text.

Enforce it rather than requesting it:

```
# The only compose fragment in this book.
services:
  presidio-analyzer:
    image: presidio-analyzer@sha256:<digest>
    read_only: true
    tmpfs:
      - /tmp:size=64m,mode=1777
    user: "10001:10001"
    cap_drop: [ALL]
    security_opt:
      - no-new-privileges:true
    networks: [detection]
    environment:
      - LOG_LEVEL=WARNING

networks:
  detection:
    internal: true        # no route to the internet
```

A read-only root filesystem turns "we do not write files" from a claim into a property. If something tries, it fails, loudly, in testing, rather than quietly succeeding in production.

The one thing that legitimately persists is the token vault from Chapter 9, and it is the most sensitive store you own, because it holds the mapping from token back to person. Three requirements:

- **Encrypted at rest**, with the key in a key management service rather than in configuration.
- **The service holding the ciphertext does not hold the key.** Separation is what makes a stolen backup useless.
- **A TTL matched to the conversation.** Not ninety days. Expire it and let the map go.

3. Logs: the detector's log is the worst log you have

Follow the logic. The detector reads all your personal data. Default log levels in most frameworks write request bodies at debug or trace. Someone raised the log level during an incident in March and it was never lowered.

You now have a single, centralised, indexed, long-retention store of every piece of personal data in your organisation, sitting in your observability platform, searchable by everyone with a dashboard login, replicated wherever your vendor replicates.

Turn request-body logging off explicitly, and then assert it:

```
[Fact]
public async Task Gateway_NeverLogsPromptContent()
{
    var sink = new CapturingLogSink();
    var gateway = BuildGateway(sink);

    await gateway.CompleteAsync(
        new GatewayRequest(Prompt: "Contact Sarah Whitfield at swhitfield@example.com"), default);

    Assert.DoesNotContain(sink.Entries, e =>
        e.Contains("Sarah Whitfield", StringComparison.OrdinalIgnoreCase)
        || e.Contains("swhitfield@example.com", StringComparison.OrdinalIgnoreCase));
}
```

That test is worth more than a policy document. It fails when someone adds a helpful debug line, and it fails in CI rather than in an audit.

Chapter 16 takes this further, because the problem is not confined to the sidecar.

4. Egress: deny it at the network

The detection container has no legitimate reason to make outbound internet connections. It receives text, runs local models, returns spans.

Deny egress at the network policy, not in application configuration. `internal: true` in the compose fragment above, a `NetworkPolicy` with no egress rule in Kubernetes, a security group with no outbound rules in a cloud VPC.

This single control neutralises a category of outcomes. A compromised dependency that phones home cannot. A malicious model artefact that beacons cannot. Data staged for exfiltration has nowhere to go. It costs one configuration block and it is the highest-value control in this chapter after "do not log the data".

The same applies to the gateway, with one exception: the gateway must reach the model provider. Allowlist that one host and deny the rest.

5. Provenance: pin by digest

Pin images by digest, not by tag, as `image: presidio-analyzer@sha256:<digest>` in the fragment above. A tag is a mutable pointer, so `:latest` and even `:2.2.358` can be repointed at different bytes.

Know who builds the image and from what. This is not hypothetical diligence. **Presidio moved out of the Microsoft GitHub organisation and is transitioning to community ownership under the Data Privacy Stack organisation.** The project is actively maintained and the move is orderly, so this is not an argument against using it. It is a demonstration that the answer to "who maintains the component that sees all our personal data" is not fixed at adoption. Review it on a schedule, the way you review anything else that can change without telling you.

Verify signatures where the project publishes them, and mirror images into your own registry so that a deleted upstream tag does not become an incident.

6. Identity and least privilege

Non-root user. All capabilities dropped. `no-new-privileges`. Its own service identity rather than a shared one. Authenticated callers only, so that anything on the network cannot submit text and read findings.

The compose fragment above does the container half. The authentication half belongs to your platform: mutual TLS, a service mesh identity, or at minimum a bearer token the gateway holds and nothing else does.

Scope the identity tightly. The sidecar needs to accept connections and nothing else. It should not have credentials to your database, your key vault, your object storage, or your cloud account.

7. Lifecycle

The container ages. The NER model inside it ages. The Python dependencies age faster than either.

Two consequences. **Patching is continuous**, and a detection container running an eighteen-month-old base image is a liability regardless of how good its detection is. **Model updates change behaviour**, which is why Chapter 7's golden-set test runs in CI. An upgrade that improves overall accuracy and halves recall on `PERSON` should fail a build.

Put a review date on the component. Who maintains it, what version are we on, when did we last measure it, when did we last patch it. Four questions, quarterly.

The boundary of this chapter

Each of the seven controls above is the shallow end of a discipline with its own literature, its own tooling, and in several cases its own full-time roles.

Out of scope here, named so you know what you are not getting:

- **Key management and rotation.** "Put the key in a KMS" is one sentence covering envelope encryption, rotation schedules, key hierarchies, break-glass procedures, and what happens to a vault encrypted under a key you rotated.
- **Secrets management.** How the sidecar gets its certificate, how it is renewed, what happens when it expires at three in the morning.
- **Supply-chain attestation.** SBOMs, provenance attestation, signature verification in admission control, reproducible builds.
- **Cluster policy and runtime security.** Admission controllers, pod security standards, runtime detection, network segmentation beyond one deny rule.
- **Threat modelling the deployment itself.** Which is worth doing and is a workshop, not a checklist.

This chapter gives you the controls that matter most for this specific component, in a form you can implement this week. Appendix C is the same content as a one-page checklist. A complete hardening programme for a regulated production environment is a different body of work, and treating a checklist as equivalent to that work is the failure mode this section exists to prevent.

The honest position: implement these seven and your detection sidecar is defensible in a security review and materially better than most deployments of its kind. It is not the same thing as having done platform security properly, and you should not tell anyone it is.

Door one is closed, the components that close it are trustworthy. The remaining doors are the ones most teams have never opened, and they hold considerably more data.

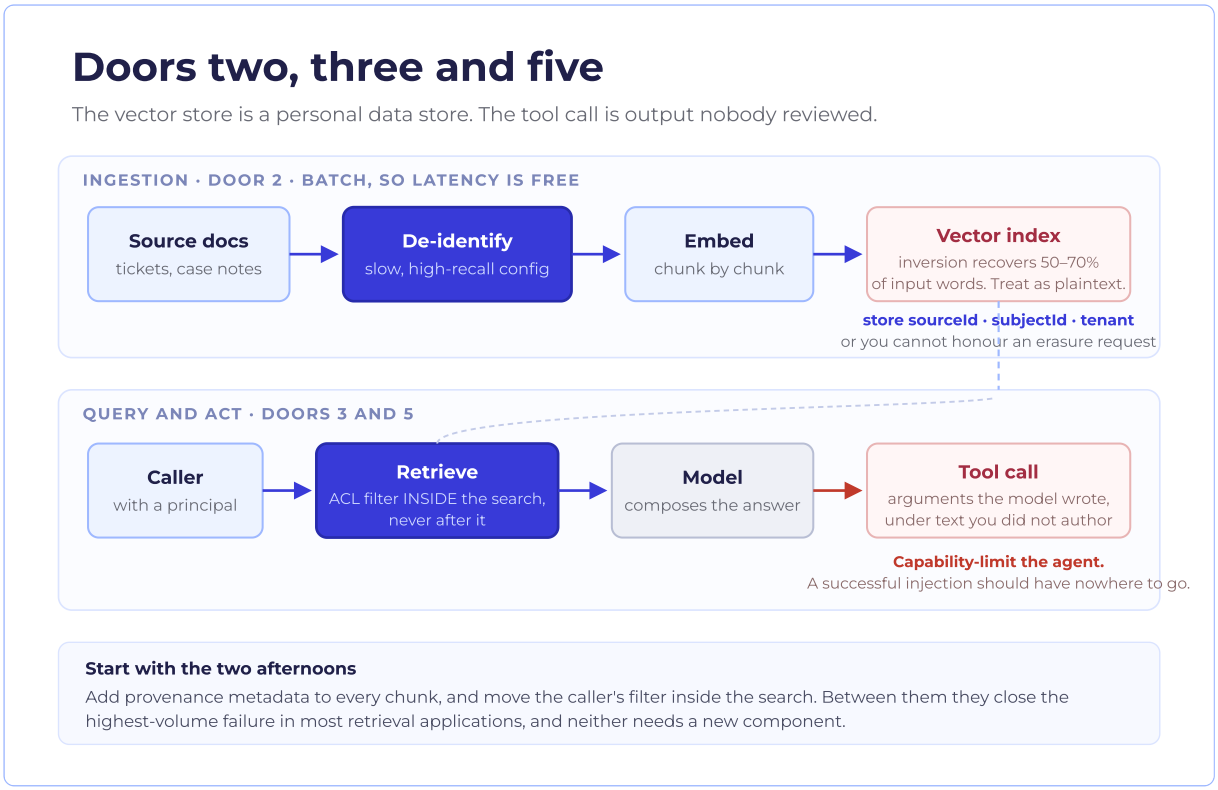
SOURCES FOR THIS CHAPTER

- **Presidio's move out of the Microsoft GitHub organisation to community ownership** — https://presidio.dataprivacystack.org/ · *Used here as the provenance example.*
- **Egress control, provenance-aware deployment and tool-call payload auditing as mitigation directions** — named across the agent-security literature cited in Chapter 15.
- **Container controls (read-only root filesystem, tmpfs, non-root UID, dropped capabilities, `no-new-privileges`, digest pinning, network policy)** — standard container hardening practice; see the Docker and Kubernetes security documentation for each directive.

This chapter is explicitly a summary. The seven controls, their ordering, and the observation that the detector sees 100% of your personal data are the author's framing. Every control named is the shallow end of a discipline with its own literature, which the chapter states at the end and Appendix C repeats.

RAG and Agents

Doors two, three and five. Your vector store is a personal data store that nobody classified, and your agent turns a text problem into an action problem.



The vector store is a database

Ask a team where their personal data lives and they will name the transactional database, probably the data warehouse, occasionally the object storage. Almost nobody names the vector index.

It is a database. It holds a copy of your support history, your document library, your case notes. It was populated by a pipeline nobody classified, it is backed up, it is replicated, and there is a good chance a copy exists in a staging environment because someone needed realistic data for testing retrieval quality.

The reason it escapes classification is that embeddings do not look like data. A vector is a list of floats. It appears to be a lossy, one-way transformation, and lossy one-way transformations feel like anonymisation.

They are not. **Embedding inversion attacks recover 50% to 70% of the original input words** from compromised vectors, and reconstruction approaches near-optimal accuracy with as few as a thousand samples against black-box encoders. OWASP lists vector and embedding weaknesses as a category in its own right: **LLM08:2025**, renumbered **LLM09:2026** in the current edition.

Read that as the operational rule: **treat your vector store as if it contains the plaintext**, because for practical purposes it does. Encrypt it, control access to it, classify it, include it in your data inventory and your deletion processes, and do not copy it to staging.

Deletion, which is where this gets expensive

A person exercises their right to erasure. You delete their row. Where else are they?

In the vector index, as chunks of text they appear in, which may include documents authored by someone else that mention them. In any cached embedding. In a backup taken last Tuesday. In the fine-tuning set, if you built one, where they are now diffused into weights and cannot be removed at all.

The engineering answer is to design the index for deletion from the start:

- **Store provenance on every chunk.** Which source document, which subject, which tenant. A chunk with no lineage cannot be deleted on request.
- **Prefer re-indexing a source to patching the index.** Deleting the source and re-running ingestion is reliable; surgical vector deletion is not always supported and not always complete.
- **Keep fine-tuning sets separate from retrieval corpora**, and think very carefully before putting personal data in one, because that decision is irreversible.

De-identify before you embed

The strongest control for door two is the same as for door one: do not put it in.

Run your detection and de-identification pipeline over documents at **ingestion time**, not at query time. Ingestion is a batch job, so the latency argument disappears entirely. The 118 to 198 millisecond transformer detectors from Chapter 4 are perfectly affordable when you are processing a corpus overnight, and you can afford a lower threshold and a second pass.

```
public async Task IngestAsync(SourceDocument doc, CancellationToken ct)
{
    // Batch context: use the slower, higher-recall configuration.
    var findings = await _detector.AnalyzeAsync(
        doc.Text, policy: IngestionPolicy, ct: ct);

    var treated = Deidentify(doc.Text, findings);

    foreach (var chunk in Chunk(treated.Text))
    {
        await _index.UpsertAsync(new IndexEntry(
            Vector: await _embedder.EmbedAsync(chunk.Text, ct),
            Text: chunk.Text,
            SourceDocumentId: doc.Id,      // for deletion
            SubjectId: doc.SubjectId,      // for erasure requests
            Tenant: doc.Tenant), ct);      // for partitioning
    }
}
```

The three metadata fields at the bottom are not optional. They are how you answer an erasure request, how you scope retrieval, and how you prove which tenant's data is where.

Not everything can be de-identified before embedding, because sometimes the name is what makes the document retrievable. When that is true, the index holds personal data by design, and the controls move to access rather than content.

Retrieval must inherit the caller's permissions

The most common serious flaw in RAG systems, and it is an authorisation bug wearing a machine learning costume.

The indexer runs with broad permissions, because it has to read everything to index it. Retrieval then queries the index. If retrieval does not re-apply the caller's permissions, every user can reach every document, and the similarity search will happily surface the most relevant one regardless of who owns it.

```
public async Task<IReadOnlyList<Chunk>> RetrieveAsync(
    string query, ClaimsPrincipal caller, CancellationToken ct)
{
    var filter = new IndexFilter(
        Tenant: caller.TenantId(),
        VisibleTo: await _authz.VisibleSourceIdsAsync(caller, ct));

    // Filter is applied in the query, not after. Post-filtering
    // leaks through result counts and latency.
    return await _index.SearchAsync(query, filter, topK: 8, ct);
}
```

Apply the filter **inside** the search rather than after it. Post-filtering leaks: the number of results removed, the latency profile, and the ranking behaviour all tell an observer something about documents they cannot see.

In multi-tenant systems, inadequate partitioning causes cross-context retrieval, where one tenant's query surfaces another tenant's embeddings. Where the risk justifies it, separate indexes per tenant are stronger than a shared index with a filter, because a filter is a line of code that can be omitted and an index boundary is not.

Agents: the tool call is unreviewed output

An agent composes the arguments it passes to your tools. That text did not come from your code, it came from a model, and its contents are not predictable.

This is structurally different from every leak so far. A prompt is a string you built. A tool call is a string the model built, possibly under the influence of text an attacker controlled.

The research is unambiguous. Simple prompt injection has been shown to leak personal data that agents observed during ordinary task execution, and data exfiltration via backdoored tool use is a demonstrated technique. Guardrail classifiers placed in front of the model reduce the success rate of malicious prompts materially, and do not eliminate it. This book does not quote a figure for that reduction, because the numbers in circulation trace to vendor testing rather than to a primary result that could be checked.

Three controls, in order of how much they buy you.

Capability design. Limit what the agent can do so that a successful injection has a small blast radius. An agent that can read customer records and send email is an exfiltration tool with extra steps. An agent that can read customer records and draft email for human approval is not. The question is never "can the model be tricked" but "what happens when it is".

Filter the tool-call arguments. Run the same detection pipeline over arguments that you run over prompts. Model-composed text going outward is exactly the case the gateway exists for.

```
public async Task<ToolResult> InvokeAsync(
    ToolCall call, GatewayPolicy policy, CancellationToken ct)
{
    if (_tools[call.Name].HasExternalEffect)
    {
        var inspected = await _detection.ApplyAsync(
            call.SerialisedArguments, policy.Outbound, ct);

        if (inspected.Findings.Count > 0 && policy.BlockPiiInToolCalls)
            throw new ToolCallBlockedException(call.Name, inspected.Findings);
    }

    return await _tools[call.Name].InvokeAsync(call, ct);
}
```

Audit every call. Tool name, argument shape, finding counts, caller, timestamp. Not the arguments themselves, for the Chapter 13 reason. Without this record, exfiltration through a tool leaves no trace in your prompt logs, because the data never appeared in a completion.

Dual-model separation

For agents handling untrusted content, one more pattern is worth knowing.

Split the work between a **privileged model** that sees sensitive data and can call tools, and an **unprivileged model** that processes untrusted external content and cannot. The untrusted content never reaches the model holding the capabilities, so an injection in a scraped web page or an inbound email cannot directly drive a tool call.

It costs an extra call and a more complex orchestration. For an agent that reads inbound email and takes action, it is the difference between a design that can be hijacked by anyone who can send you a message and one that cannot.

The output door, again

Chapter 13 built the markdown image filter. It belongs here too, because retrieval and agents are what make it exploitable.

EchoLeak, CVE-2025-32711, is the worked example. A crafted email reached Microsoft 365 Copilot's context through ordinary ingestion. Reference-style markdown slipped past link redaction. Auto-fetched images carried data out. Zero clicks. Microsoft patched it server-side in June 2025 and reported no exploitation in the wild, so treat it as a proven technique rather than an open hole.

Every element was a normal feature. Ingest email so the assistant is useful. Render markdown so answers look good. Fetch images so they display. The vulnerability lived in the combination, which is why a control on any single component would have missed it, and why the output filter belongs at the gateway where it sees the assembled result.

If your application renders model output and your model reads content you did not author, you have this shape. Allowlist the hosts your renderer will fetch from, and treat every URL in a completion as attacker-controlled until proven otherwise.

What to do first

CONTROL	EFFORT	CLOSES
Add provenance metadata to chunks	Low	Erasure, partitioning
Apply the caller's filter inside the search	Low	Cross-tenant retrieval
De-identify at ingestion	Medium	Bulk exposure in the index
Output host allowlist	Low	Rendered exfiltration
Audit tool calls	Low	Blind exfiltration
Capability-limit agents	Medium	Injection blast radius
Dual-model separation	High	Injection from untrusted content

The first two are afternoons and they close the highest-volume failure in most RAG applications. Start there.

Four doors are shut. The fifth is the widest, the cheapest to close, and the one almost nobody has looked at.

SOURCES FOR THIS CHAPTER

- **Embedding inversion recovering 50–70% of original input words** — arXiv 2406.10280: https://arxiv.org/pdf/2406.10280
- **OWASP Top 10 for LLM Applications: vector and embedding weaknesses, LLM08:2025 / LLM09:2026** — https://genai.owasp.org/initiatives/top-10-for-llm-and-genai/ · 2025 PDF: https://owasp.org/www-project-top-10-for-large-language-model-applications/assets/PDF/OWASP-Top-10-for-LLMs-v2025.pdf
- **Cross-context retrieval in inadequately partitioned multi-tenant vector stores** — https://www.protecto.ai/blog/how-rag-systems-sliently-expose-pii/ · https://christian-schneider.net/blog/rag-security-forgotten-attack-surface/ · *Both practitioner or vendor sources.*
- **Prompt injection leaking personal data observed by agents during ordinary task execution** — arXiv 2506.01055: https://arxiv.org/pdf/2506.01055
- **Data exfiltration via backdoored tool use** — arXiv 2604.05432: https://arxiv.org/pdf/2604.05432
- **Guardrail classifiers reduce malicious-prompt success materially but not completely** — stated qualitatively on purpose. A ">90%" figure circulates widely in secondary reporting; the primary result could not be located, so the number is **not** quoted in the text.
- **EchoLeak, CVE-2025-32711** — arXiv 2509.10540: https://arxiv.org/abs/2509.10540 · Disclosed by Aim Security June 2025, CVSS 9.3, patched server-side June 2025, no exploitation in the wild.
- **Capability-based agent design and dual-model separation** — mitigation directions named across the agent-security literature above.

The deletion and provenance guidance, the argument for filtering inside the search rather than after it, and the priority table at the end are the author's engineering positions.

The Boring Controls

Your logs hold more personal data than your prompts do, and your contract decides what the provider keeps. Neither is interesting. Both matter more than the detector.

Your observability stack is a personal data store

Every LLM integration logs the prompt and the completion. It happens early, for a good reason: when a model returns something strange, the first thing anyone wants is the exact input. So someone adds it during development and it ships.

Follow what that produces. A verbatim copy of every prompt, retained at whatever the platform default is, indexed and searchable, available to everyone with a dashboard login, replicated to wherever your observability vendor replicates, and exported to a data lake for analysis.

Compare that to the prompt itself, which goes to a provider you have a contract with, under retention terms you negotiated, and may not be retained at all. The log is longer-lived, more accessible, and less governed than the thing it is logging.

Then add the adjacent sources. **Exception messages** carry the object being processed. **Traces** capture span attributes that include request payloads. **Analytics events** fired for product metrics carry properties nobody reviewed. **Error reporting tools** attach the full request body by default because that is what makes them useful.

Redact at the sink

The instinct is to fix this at the call site: be careful what you pass to the logger. That fails for the same reason Chapter 12's conventions fail. There are hundreds of call sites and one of them will be wrong.

Fix it where everything converges.

```
public sealed class RedactingEnricher(IDeterministicDetector detector) : ILogEventEnricher
{
    public void Enrich(LogEvent e, ILogEventPropertyFactory factory)
    {
        foreach (var (name, value) in e.Properties.ToList())
        {
            if (value is not ScalarValue { Value: string s }) continue;

            var redacted = detector.RedactAll(s);
            if (!ReferenceEquals(redacted, s))
                e.AddOrUpdateProperty(factory.CreateProperty(name, redacted));
        }
    }
}
```

Use Chapter 5's deterministic layer here rather than the model-based detector. It runs in 0.1 ms, in-process, with no network call. A logging pipeline cannot afford a 15 ms round trip per event, and cannot afford to fail when the container restarts. Accept the narrower coverage: catching every card number, IBAN and national identifier in your logs is a large win on its own.

Three more measures worth the hour they cost.

Deny-list the field names that should never be logged. `password`, `ssn`, `ein`, `routingNumber`, `accountNumber`, `dateOfBirth`, `body`, `prompt`, `completion`. Structural, cheap, and it catches the case where someone logs a whole DTO.

Set retention deliberately. Most teams have never changed it. Thirty days of application logs is plenty for debugging and dramatically better than the two years your platform defaults to.

Test it. The same shape as Chapter 14's assertion:

```
[Fact]
public void Logger_RedactsIdentifiersInStructuredProperties()
{
    var sink = new CapturingLogSink();
    var log = BuildLogger(sink);

    log.Information("Processing {Payload}",
        new { Routing = "021000021", Note = "card 4111111111111111" });

    Assert.DoesNotContain(sink.Entries, e => e.Contains("021000021"));
    Assert.DoesNotContain(sink.Entries, e => e.Contains("4111111111111111"));
}
```

The provider side

Now the half you cannot fix in code.

Three questions, and all three are contract terms rather than engineering problems:

Is the data retained, and for how long? Providers differ, and several offer zero data retention for API traffic where prompts are not stored at all.

Is it used for training? Major providers state they do not train on API traffic by default, and this is usually the easiest term to confirm.

Where is it processed? Regional processing matters for any cross-border transfer analysis. Some providers commit contractually to processing within a region you name; others commit only on particular tiers, or not at all. This is a term to read rather than a property to assume, and it is one of the few where the answer is usually written down plainly.

Two of those three are genuinely negotiable, and the third is usually already answered in the provider's standard terms.

This book deliberately names no provider's retention terms. That is not squeamishness, it is the same standard applied to itself: **provider terms changed at least twice during 2026**, and a figure that cannot be re-verified at the moment of reading is a figure that will mislead someone. Any per-vendor comparison table printed in a book is stale within a quarter.

What does not go stale is the structural point:

> *Retention, residency and training-use are negotiable contract terms. They cost a procurement conversation rather than an engineering quarter, and they are frequently the highest-leverage control available to you.*

A team can spend six weeks building a redaction pipeline that catches half of what it sees, or spend two weeks getting zero data retention and regional processing written into a contract. Do the second one first. Then build the pipeline, because the contract does not help with doors two through five.

Go to the provider's own documentation, not a comparison article. Read the enterprise or commercial terms rather than the consumer privacy policy, because they differ and only one of them governs your API traffic. Record the URL, the date and the exact wording, because "we do not train on your data" and "we do not train on your data by default" are different commitments. Then check whether the term you are relying on is contractual or merely documentary: documentation changes without notice, an agreement does not.

Who is actually enforcing

A word about the risk climate, because it shapes how these projects get funded, and the loudest version of the story is not the accurate one. Every figure below carries the date it was verified, and every one of them moves.

California is the one with a dedicated regulator. The CPRA created the California Privacy Protection Agency, the only privacy-specific regulator in the country. At a **September 2025** board meeting its staff reported hundreds of investigations and enforcement actions in progress, many at a stage where the business did not yet know it was a target.

Civil penalties are CPI-adjusted in odd-numbered years. The amounts in force since **1 January 2025**, and unchanged through 2026, are **$2,663 per violation** and **$7,988 per intentional violation**, or per violation involving a consumer the business knows to be under 16. The next adjustment is due in 2027. Penalties are assessed per violation, which in practice usually means per affected consumer, so totals scale with your user count rather than with the severity of the mistake.

The CCPA also carries the only **private right of action** in a comprehensive US privacy law, for breaches caused by a failure to maintain reasonable security. Statutory damages are likewise CPI-adjusted: **$107 to $799 per consumer per incident** since 1 January 2025, not the $100 to $750 written in the statute. Statutory damages with no requirement to prove harm are what make class actions economic, and that is a different risk shape from a regulator's fine. One material qualifier: a consumer must give **30 days' written notice**, and if the business actually cures the violation within that window and says so in writing, no action for statutory damages may be brought.

New CCPA regulations covering risk assessments, cybersecurity audits, insurance data and **automated decision-making technology** took effect **1 January 2026**. Read the dates carefully, because they are not the same date. The regulations are in force now; the substantive obligation for businesses using ADMT to make significant decisions about consumers begins **1 January 2027**. If your LLM feature contributes to a decision about a person's employment, credit, housing, education or healthcare, that is the deadline in your calendar, and it has not passed.

If you touch health data, OCR is the more likely counterparty. The HHS Office for Civil Rights closed 2025 with **21 settlements and civil monetary penalties**, its second-highest annual total, and has continued at pace through 2026. Penalties are inflation-adjusted annually: the calendar-year cap for all violations of an identical provision rose to **$2,190,294** effective **28 January 2026**. Note the gap between the statutory number and practice, because it cuts both ways: since 2019 OCR has exercised enforcement discretion to apply materially lower annual caps to the three lower culpability tiers, so the headline figure is the ceiling rather than the expectation. Business associates have been directly liable since HITECH, so a vendor position does not insulate you.

One thing frequently reported as imminent is not. The proposed overhaul of the **HIPAA Security Rule**, published as an NPRM in January 2025, **has not been finalised**. As of September 2026 it remains a proposal, and OMB's target for final action has moved to **July 2027**. Plan for it; do not tell your board it is law.

The FTC covers the gap. Mobile health apps, fitness trackers and personal health record vendors sit outside HIPAA and inside the FTC's remit, under Section 5 and the Health Breach Notification Rule.

Now the contrast, and it needs stating carefully because it is widely misreported. The Italian data protection authority fined OpenAI €15 million over ChatGPT, and that fine was cited everywhere as proof that AI privacy enforcement had arrived. **On 18 March 2026 the Court of Rome annulled the decision in its entirety**, setting aside both the fine and the order to run a public awareness campaign.

Read the reason before drawing the lesson. The court annulled on **jurisdictional** grounds: once OpenAI established its Irish establishment in 2024, the GDPR one-stop-shop mechanism made the Irish authority the lead supervisory authority, and the Garante no longer had competence to act. The ruling therefore says almost nothing about whether the underlying processing was lawful. It is not a finding that the conduct was fine. It is a finding about who was entitled to ask.

The useful conclusion is narrower than "enforcement is toothless" and more useful than "a fine was issued". The generative-AI enforcement *record* is thinner than the enforcement *noise*, and a meaningful part of what did happen turned on procedure rather than on substance. Plan for the ordinary kind of enforcement instead. A state attorney general or the CPPA asking what personal information you collected and why. OCR asking whether a business associate agreement was in place. A plaintiff's firm counting consumers after a breach. None of those requires an AI-specific law, and all of them turn on the same question: **was this data necessary?**

That is data minimisation, it is Chapter 12, and it is enforceable today under every regime named above.

If you also serve EU customers, the EU AI Act adds transparency duties rather than new data protection rules. **Article 50 transparency obligations became applicable on 2 August 2026**, catching user-facing chatbots and synthetic content. The **Digital Omnibus on AI, Regulation (EU) 2026/1744**, in force since 27 July 2026, deferred the high-risk regime to **2 December 2027** for standalone Annex III systems and **2 August 2028** for AI embedded in products already covered by EU product-safety law. It left Article 50 alone, apart from a four-month grace period until **2 December 2026** on the content-marking duty in Article 50(2) for systems already on the market. Separately, the Commission's power to fine providers of general-purpose AI models under Article 101 was carved out of the August 2025 enforcement date and applies from 2 August 2026.

If you are building a chatbot, the obligation live today is telling users they are talking to a machine.

Build for minimisation. It is the thing every regulator named here can already enforce, it is the control that actually reduces your exposure, and it is cheaper than everything else in this book.

The boring checklist

CONTROL	EFFORT	REDUCES
Turn off prompt/completion logging	Hours	The largest store of personal data you have
Redact at the log sink	Hours	Everything incidental in telemetry
Field deny-list	Hours	Whole-DTO logging accidents
Shorten log retention	Minutes	Exposure window, storage cost
Zero data retention with your provider	A procurement conversation	Provider-side persistence
Confirm no training on API data	An email	Memorisation risk
Regional processing	A contract term	Transfer risk

Nothing on that list is technically interesting. The top four are an afternoon of work between them and they close the widest door in most organisations. The bottom three are someone else's afternoon and they remove an entire category of risk you cannot otherwise touch.

Everything is in place: minimised inputs, a chokepoint, a trustworthy detector, closed doors, and negotiated terms. None of it counts until you can show it, and showing it is easier than it sounds if you built the previous chapters the way they were described.

SOURCES FOR THIS CHAPTER

All figures below were verified on 14 September 2026. Every one of them is subject to periodic adjustment; the dates are part of the claim.

- **CCPA civil penalties $2,663 / $7,988, CPI-adjusted, in force since 1 January 2025, next adjustment 2027** — California Privacy Protection Agency announcement: https://cppa.ca.gov/announcements/2024/20241217.html
- **CCPA private right of action, $107–$799 per consumer per incident (CPI-adjusted from the statutory $100–$750), and the 30-day cure notice** — Cal. Civ. Code §1798.150; adjusted amounts effective 1 January 2025, next adjustment 1 January 2027: https://databreachcost.com/regulation/ccpa-breach-fine · *The statutory figures are still widely quoted. The adjusted ones are the operative amounts.*
- **CPPA staff reporting hundreds of investigations in progress** — reported at the agency's **September 2025** board meeting. *An earlier draft of this book dated this to "early 2026"; that was wrong.*
- **CCPA regulations on risk assessments, cybersecurity audits, insurance and ADMT effective 1 January 2026; substantive ADMT obligations from 1 January 2027** — approved by the Office of Administrative Law 22 September 2025:

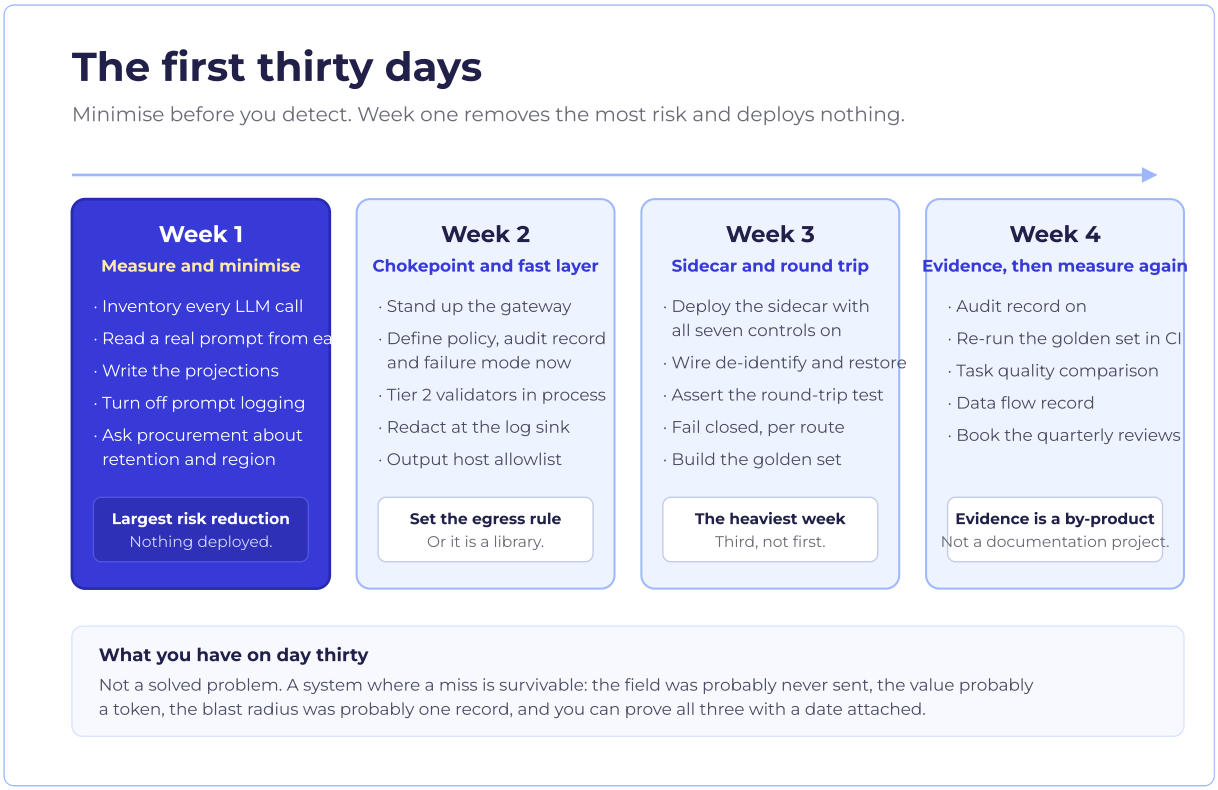
https://cppa.ca.gov/regulations/ccpa_updates.html · Skadden analysis: https://www.skadden.com/insights/publications/2025/10/california-finalizes-cppa-regulations · *The two dates are distinct and are frequently conflated.*

- **Twenty states with a comprehensive privacy law in effect; twenty-four enacted including four with future effective dates** — IAPP US State Privacy Legislation Tracker: https://iapp.org/resources/article/us-state-privacy-legislation-tracker
- **OCR closed 2025 with 21 settlements and civil monetary penalties, its second-highest annual total** — https://www.hipaajournal.com/what-are-the-penalties-for-hipaa-violations-7096/
- **HIPAA calendar-year penalty cap $2,190,294 effective 28 January 2026** (OMB multiplier 1.02598), **and OCR's standing enforcement discretion to apply lower caps to the three lower tiers since 2019** — https://www.mercer.com/insights/law-and-policy/hhs-adjusts-2026-hipaa-certain-aca-and-msp-monetary-penalties/
- **The HIPAA Security Rule overhaul remains a proposal. NPRM published 6 January 2025; no final rule as of September 2026; OMB target for final action moved to July 2027** — HHS NPRM page: https://www.hhs.gov/hipaa/for-professionals/security/hipaa-security-rule-nprm/index.html · status: https://www.hipaajournal.com/hipaa-security-rule-update-postponed/ · *An earlier draft said it was expected to finalise in May 2026. It did not.*
- **FTC jurisdiction over health apps and PHR vendors outside HIPAA** — Section 5 and the Health Breach Notification Rule.
- **Garante €15M fine against OpenAI (2 November 2024), annulled in its entirety by the Court of Rome on 18 March 2026 on jurisdictional grounds under the GDPR one-stop-shop, following OpenAI's 2024 Irish establishment** — Wilson Sonsini: https://www.wsgr.com/en/insights/openai-prevails-in-landmark-italian-ai-and-gdpr-enforcement-case.html · analysis: https://www.europeanlawblog.eu/pub/92oig1ws · https://www.crossborderdataforum.org/generative-ai-and-gdpr-enforcement-in-europe-a-lot-of-noise-one-fine-zero-survivors/ · *The jurisdictional basis matters: the ruling is not a finding that the processing was lawful. An earlier draft of this book omitted the reason and over-read the result.*
- **EU AI Act Article 50 applicable 2 August 2026; Article 101 GPAI fines carved out of the 2 August 2025 date and applicable from 2 August 2026** — Article 113: https://artificialintelligenceact.eu/article/113/
- **Digital Omnibus on AI, Regulation (EU) 2026/1744: published in the Official Journal 24 July 2026, in force 27 July 2026. High-risk deferred to 2 December 2027 (Annex III) and 2 August 2028 (Annex I). Article 50 unamended apart from a grace period to 2 December 2026 on the Article 50(2) content-marking duty for systems already on the market** — Gibson Dunn: https://www.gibsondunn.com/eu-ai-act-omnibus-agreement-postponed-high-risk-deadlines-and-other-key-changes/ · Jones Walker: https://www.joneswalker.com/en/insights/blogs/ai-law-blog/yes-august-2-still-matters-the-eu-approved-a-high-risk-ai-delay-but-most-trans.html · *Caution: the widely-linked artificialintelligenceact.eu Article 113 page still shows the pre-Omnibus 2 August 2027 date for Annex I. The adopted regulation says 2 August 2028.*
- **Provider retention terms: deliberately not quoted.** No vendor's retention, residency or training-use terms are stated in this chapter. They changed at least twice during 2026, OpenAI's own page could not be retrieved at verification time, and a claim this book cannot re-verify is a claim it does not make. The chapter gives the procedure for checking instead.

The claim that your observability stack holds more personal data than your prompts do is the author's assertion from practice, not a measured finding. It is offered as a prompt to go and look at your own log store.

Evidence, and the First Thirty Days

Two halves. What you need to be able to show, and the order to build it in. Both are shorter than you expect, because the evidence is a by-product of doing the work properly rather than a separate project.



What people actually ask for

Three audiences will ask, and they want the same five things in different tones of voice.

An **enterprise customer's security questionnaire** wants to know whether their data reaches a third-party model and under what controls. A **regulator or DPO** wants a lawful basis, a minimisation argument, and a record of processing. An **auditor** wants to see that the control you described was operating on a given date.

The five artefacts:

1. A data flow record. For each LLM feature: what personal data enters the prompt, what enters any corpus, which provider receives it, under what retention terms, in which region. This is Chapter 2's inventory, kept current.

2. The policy, versioned. Which entities are detected, at what threshold, treated how, on which routes, with a version string and a change history. Chapter 13 produced this as configuration in source control, which means the history is `git log`.

3. Detector performance on your own data. Precision and recall per entity type, measured on a golden set drawn from your traffic, with the measurement date and the sample size. Chapter 7 built this and put it in CI, so it is current by construction.

4. Proof the control was running. The gateway audit record: request, tenant, route, policy version, finding counts, whether it degraded. Not the content. This is what answers "was the control in force on 3 March?", and it answers it precisely.

5. A minimisation argument. Why each field in the prompt is necessary. Chapter 12 put this in a comment on the projection record, where it stays maintained.

Notice what is not on the list: a policy document nobody reads, a spreadsheet of controls maintained by hand, a slide deck. Every one of the five is generated by the system doing its job. That is the difference between evidence you have and evidence you assemble in a panic.

The two formal documents

A DPIA is required under GDPR where processing is likely to result in high risk, and an LLM feature processing personal data at scale frequently qualifies. Your DPO decides. What engineering owes it is the substance: the data flow record, the minimisation argument, the residual risk after controls, and the controls themselves. Artefacts 1, 2, 3 and 5 are most of a DPIA already.

Article 50 transparency, applicable since 2 August 2026, requires that people are told when they are interacting with an AI system. If you have a user-facing chatbot, this is a line of interface copy and a decision about where it appears. It is the most easily satisfied obligation in this book and the one most likely to be forgotten, because it is a product task rather than an engineering one.

Week one: measure and minimise

Do not start with the detector. Start with what you are sending.

Inventory every LLM call. Find them all, including the one in the batch job and the one somebody added to the admin tool. For each, capture a real prompt and read it. Teams routinely find a field they did not know they were sending, and the discovery is free to fix.

Write the projections. Chapter 12. For each feature, define the record that contains only what the task needs, with a comment per field saying why. This is the largest risk reduction available to you and it is a day or two of work.

Turn off prompt and completion logging, and add the assertion test. Chapter 16. An afternoon, and it closes the widest door.

Ask procurement to confirm retention, training-use and region with your provider. It runs in the background for the rest of the month.

By Friday you have measurably reduced exposure and you have not deployed a single new component.

Week two: the chokepoint and the fast layer

Stand up the gateway. Chapter 13. Start as a library if you must, but define the policy shape, the audit record, and the failure mode now, because retrofitting those is painful.

Add the deterministic layer. Chapter 5. Luhn, ABA routing, NPI, and the identifiers you actually handle. In-process, 0.1 ms, no dependencies. It catches your highest-consequence entities at near-certainty, and it becomes the degraded mode when the detector is unavailable.

Redact at the log sink using that same layer. Chapter 16.

Add the output filter. Chapter 13's host allowlist for markdown images and links. Small, and it closes the door EchoLeak used.

End of week two, the highest-consequence identifiers are handled deterministically and every model call passes through one place you control.

Week three: the sidecar and the round trip

Deploy the detection container with Chapter 14's controls applied from the start: TLS, read-only filesystem, no egress, non-root, pinned digest, logging off. Applying them later means a migration.

Wire the round trip. Chapter 9. De-identify, call, re-identify, with the round-trip invariant test, because a 64% reversibility pass rate is a real result from a real benchmark.

Fail closed. Chapters 9 and 13. Decide per route, record the decision in policy, count the degradations.

Build the golden set. Chapter 7. Two to five hundred labelled samples from your own traffic, handled with the care Chapter 7 describes. Score it, choose thresholds per entity, commit the floors to CI.

This is the heaviest week. It is third rather than first because everything before it reduces how much this layer has to catch.

Week four: evidence, and measure again

Turn on the audit record. Counts and types, never content.

Re-run the golden set and record the numbers with their date. This is artefact 3 and it now runs on every build.

Run the task quality comparison from Chapter 10. Fifty examples, treated and untreated, read side by side. Find out whether your controls broke the feature before a user does.

Write the data flow record and the minimisation argument. Mostly assembly, because the inputs already exist.

Book the recurring reviews. Re-measure the golden set quarterly. Review the sidecar's provenance and patch level quarterly. Re-verify provider terms quarterly, with dates.

What you have after thirty days

Prompts carry only the fields the task needs. Every call passes through one chokepoint with versioned policy. Structured identifiers are caught deterministically at near-certainty. Names and free text are caught probabilistically by a detector you host, whose actual performance on your data you can state with a number and a date. Real values never reach the provider, and the mapping never leaves your infrastructure. Logs are redacted and short-lived. The output is filtered. The audit trail proves it.

You will not have solved this. Chapter 4 is clear about why: cross-domain detection lands around 0.5, and yours will too. What you will have is a system where a detection miss is survivable, because the field was probably never sent, the value was probably a token, the blast radius was probably one record, and you can prove all three.

That is the achievable goal. Anyone offering you more than that is selling the demo number.

Detection is a net, not a wall.

Keeping it running

The uncomfortable part is that none of this is finished. The detector drifts as your traffic changes. The container ages faster than the model inside it. Provider terms move, twice a year on recent evidence. New features get built by developers who were not in the room for any of this, and the marker interface only protects the paths someone remembered to route through it.

Chapter 14 drew a line and said complete hardening is a different body of work. The same line applies to the whole system. Building this takes a month. Running it takes attention, indefinitely, from people who keep the context.

If that is work your team should own, own it. The book is deliberately specific enough to implement without help.

If it is work you would rather have delivered and maintained by engineers who do this regularly, that is what **You-Source Dev on Demand** is for: subscription engineering, delivered as small, scoped tasks with a cadence and visible accountability, instead of a hiring round or a consultancy engagement. The sensible way to start is a **Proof of Quality**: pick one real task from the thirty-day plan, have it delivered, and judge the work before committing to anything. Details at you-source.com.

Don't detect what you can refuse to send.

SOURCES FOR THIS CHAPTER

- **EU AI Act Article 50 transparency obligations, applicable 2 August 2026** — https://artificialintelligenceact.eu/implementation-timeline/
- **DPIA requirement for high-risk processing** — GDPR Article 35. **Risk assessment and automated decision-making obligations** under CPRA regulations effective 1 January 2026 — see the state trackers cited in Chapter 16.
- **All quantitative claims restated here** (cross-domain detection around 0.5, the demo gap) are sourced in Chapter 4.

The five evidence artefacts and the four-week sequence are the author's recommendation. The ordering follows the book's argument that minimisation is cheaper and more effective than detection; it is not derived from a published framework.

Appendix A — Entity Catalogue and C# Validators

The reusable part. US identifiers you can detect deterministically, with working validators and honest false-positive characteristics.

The catalogue

ENTITY	FORMAT	CHECKSUM	FALSE POSITIVES WITHOUT VALIDATION	CONFIDENCE WITH VALIDATION
Payment card	12–19 digits	Luhn	Very high	Very high
ABA routing number	9 digits	Weighted mod-10	Very high	Very high
NPI	10 digits	Luhn, `80840` prefixed	Very high	Very high
IBAN (international customers)	15–34 alphanumeric	mod-97	Low	Very high
SSN	9 digits	None, range rules only	Very high	Medium
ITIN	9 digits, `9` prefix	None, range rules only	Very high	Medium
EIN	9 digits	None, prefix list only	Very high	Medium
MBI (Medicare)	11 chars, mixed	None, positional rules	Medium	Medium
US bank account	4–17 digits	None	Very high	Low, needs routing context
Driver's licence	Varies by state	None	Very high	Low
Email	RFC 5322	None	Low	High
US phone (NANP)	10 digits	None, NANP rules	High	Medium
ZIP / ZIP+4	5 or 9 digits	None	High	Low
IPv4 / IPv6	Dotted quad / hex	None	Medium	High, often not personal data

Read the last two columns together. **A validator is what separates an identifier from a number.** Where no checksum exists, format and context are all you have, and confidence caps out at medium.

The four rows with a real checksum are worth disproportionate attention. They are also, conveniently, four of the highest-consequence categories: payment, banking, healthcare provider, and international banking.

Luhn, for payment cards

```
public static bool IsLuhnValid(ReadOnlySpan<char> digits)
{
    int sum = 0;
    bool doubling = false;

    for (int i = digits.Length - 1; i >= 0; i--)
    {
        if (!char.IsDigit(digits[i])) return false;

        int d = digits[i] - '0';
        if (doubling)
        {
            d *= 2;
            if (d > 9) d -= 9;
        }

        sum += d;
        doubling = !doubling;
    }

    return sum % 10 == 0;
}
```

Known-valid test values: `4111111111111111`, `5500005555555559`, `378282246310005`.

ABA routing number, weighted mod-10

Nine digits. Weights `3, 7, 1` repeating, and the weighted sum must be a multiple of ten.

```
private static readonly int[] AbaWeights = [3, 7, 1, 3, 7, 1, 3, 7, 1];

public static bool IsAbaRoutingValid(ReadOnlySpan<char> routing)
{
    if (routing.Length != 9) return false;

    int sum = 0;
    for (int i = 0; i < 9; i++)
    {
        if (!char.IsDigit(routing[i])) return false;
        sum += (routing[i] - '0') * AbaWeights[i];
    }

    return sum % 10 == 0;
}
```

Known-valid test values: `021000021`, `011000015`.

A routing number identifies an institution, not a person, so on its own it is not personal data. Paired with an account number it is. Detect the routing number first, since it is the half you can verify, then treat nearby digit runs as the account.

NPI, Luhn with the CMS prefix

The National Provider Identifier issued by CMS. Ten digits, validated by Luhn **after prepending** `80840`. Omit the prefix and every valid NPI fails, which is the single most common implementation error here.

```
public static bool IsNpiValid(string npi)
{
    var digits = new string(npi.Where(char.IsDigit).ToArray());
    if (digits.Length != 10) return false;

    // CMS specifies Luhn over the NPI prefixed with 80840.
    // "80" denotes health applications, "840" denotes the United States.
    return IsLuhnValid("80840" + digits);
}
```

Known-valid test values: `1993999998`, `1234567893`.

An NPI identifies a provider rather than a patient. It is still a strong quasi-identifier in a clinical note, because the treating provider narrows the patient population sharply.

IBAN, mod-97 (ISO 13616)

For customers outside the US. Move the first four characters to the end, convert letters to numbers where `A` is 10, and the remainder against 97 must be 1.

```
public static bool IsIbanValid(string iban)
{
    var s = new string(iban.Where(char.IsLetterOrDigit).ToArray()).ToUpperInvariant();
    if (s.Length is < 15 or > 34) return false;
    if (!char.IsLetter(s[0]) || !char.IsLetter(s[1])) return false;

    var rearranged = s[4..] + s[..4];

    int remainder = 0;
    foreach (char c in rearranged)
    {
        int value = char.IsDigit(c) ? c - '0' : c - 'A' + 10;
        remainder = value > 9
            ? (remainder * 100 + value) % 97
            : (remainder * 10 + value) % 97;
    }

    return remainder == 1;
}
```

Known-valid test values: `GB82WEST12345698765432`, `DE89370400440532013000`.

The first two characters give you the country, which tells you which rules apply to the account holder. Keep it.

SSN, range rules only

No checksum. Exclusions are all you have: area `000`, `666`, and `900` – `999` are never issued; group `00` and serial `0000` are never issued.

```
public static bool IsSsnPlausible(ReadOnlySpan<char> ssn)
{
    if (ssn.Length != 9) return false;
    foreach (var c in ssn) if (!char.IsDigit(c)) return false;

    int area = int.Parse(ssn[..3]);
    int group = int.Parse(ssn[3..5]);
    int serial = int.Parse(ssn[5..]);

    if (area is 0 or 666 || area >= 900) return false;
    return group != 0 && serial != 0;
}
```

Call it `IsPlausible`, not `IsValid`. Nine digits with no checksum means high false positives on any numeric data. Require context: a nearby label, or the `xxx-xx-xxxx` separator format, which is far more reliable than the bare digits.

ITIN, a special case of the SSN shape

Individual Taxpayer Identification Numbers share the nine-digit shape and always begin with `9`, with the group digits falling in defined ranges.

```
public static bool IsItinPlausible(ReadOnlySpan<char> itin)
{
    if (itin.Length != 9) return false;
    foreach (var c in itin) if (!char.IsDigit(c)) return false;

    if (itin[0] != '9') return false;
    int group = int.Parse(itin[3..5]);
    return group is (>= 50 and <= 65) or (>= 70 and <= 88)
                 or (>= 90 and <= 92) or (>= 94 and <= 99);
}
```

Known-valid test value: `912501234`. Because the leading `9` excludes it from valid SSN areas, checking ITIN before SSN avoids double-counting the same span.

EIN, prefix list only

Nine digits, written `XX-XXXXXXX`. No checksum. Validity is a list of issued campus prefixes, which changes, so treat this as a shape check and lean on the separator and the label.

```
private static readonly Regex EinPattern =
    new(@"\b(\d{2})-(\d{7})\b", RegexOptions.Compiled);

public static bool LooksLikeEin(string candidate) =>
    EinPattern.IsMatch(candidate);
```

An EIN identifies a business, so it is personal data only for sole proprietors, where it is frequently the owner's SSN. Detect it, then decide by context.

MBI, positional rules

The Medicare Beneficiary Identifier replaced the SSN-based HICN. Eleven characters in a fixed positional pattern of digits and a restricted letter set that excludes `S`, `L`, `O`, `I`, `B` and `Z` to avoid visual confusion.

```
// C = constrained letter set, N = digit, A = letter or digit
private static readonly Regex MbiPattern = new(
    @"^[1-9][ACDEFGHJKMNPQRTUVWXY][ACDEFGHJKMNPQRTUVWXY0-9]\d" +
    @"[ACDEFGHJKMNPQRTUVWXY][ACDEFGHJKMNPQRTUVWXY0-9]\d" +
    @"[ACDEFGHJKMNPQRTUVWXY]{2}\d{2}$",
    RegexOptions.Compiled | RegexOptions.IgnoreCase);

public static bool IsMbiPlausible(string mbi) =>
    MbiPattern.IsMatch(mbi.Replace("-", ""));
```

Known-valid test value: `1EG4TE5MK73`. The restricted alphabet does real work here. An eleven-character string matching this pattern by accident is unlikely, so despite having no checksum the MBI behaves closer to a validated identifier than the SSN does.

Verify the current pattern against CMS's published specification before relying on it.

Context scoring

For the entities with weak or absent checksums, proximity to a label does most of the work.

```
public static double ContextBoost(
    string text, int matchStart, IReadOnlyList<string> cues, int window = 40)
{
    var from = Math.Max(0, matchStart - window);
    var span = text[from..matchStart].ToLowerInvariant();
    return cues.Any(span.Contains) ? 0.35 : 0.0;
}
```

Cue lists that earn their keep:

```
["ssn", "social security", "soc sec"]
["routing", "aba", "rtn", "wire"]
["account number", "acct", "checking", "savings"]
["npi", "provider id", "rendering provider"]
["ein", "tax id", "federal id", "fein"]
["mbi", "medicare", "beneficiary id"]
["mrn", "medical record", "chart number"]
["dl", "driver's license", "license number"]
```

The HIPAA 18, as a coverage checklist

Not a detection spec. A list to check your entity coverage against, and a reminder that most of these have no pattern at all.

Names · geographic subdivisions smaller than a state · all date elements except year · telephone numbers · fax numbers · email addresses · Social Security numbers · medical record numbers · health plan beneficiary numbers · account numbers · certificate and licence numbers · vehicle identifiers and serial numbers including plates · device identifiers and serial numbers · web URLs · IP addresses · biometric identifiers including finger and voice prints · full-face photographs and comparable images · any other unique identifying number, characteristic, or code.

That last item is deliberately open-ended, and it is where your own internal customer reference belongs.

Removing all eighteen meets the Safe Harbor standard and does not eliminate re-identification risk from combinations of what remains, which is why Expert Determination exists as an alternative route. Chapter 2 covers the distinction.

Two standing rules

Date every validator. Identifier schemes change. EIN prefixes get added, MBI patterns get clarified, and card ranges shift. Put a comment naming the scheme and the year you verified it, plus a golden test with known-valid examples, so a scheme revision fails a build rather than silently returning `false` for every legitimate identifier.

Never loosen for recall. This layer's value is that when it fires it is right. Chase recall in the model layer, where a false positive costs a redacted word rather than a broken validator.

SOURCES FOR THIS APPENDIX

- ABA routing number check digit, weights 3-7-1 mod 10 — Apache Commons Validator `ABANumberCheckDigit` reference implementation: https://commons.apache.org/validator/apidocs/org/apache/commons/validator/routines/checkdigit/ABANumberCheckDigit.html · algorithm description: http://www.brainjar.com/js/validation/
- NPI check digit, Luhn over `80840` + NPI — CMS, *Requirements for National Provider Identifier (NPI) and NPI Check Digit*: https://www.cms.gov/Regulations-and-Guidance/Administrative-Simplification/NationalProvIdentStand/Downloads/NPIcheckdigit.pdf · worked example: https://www.johndcook.com/blog/2024/06/26/npi-number/
- IBAN structure and mod-97 check — ISO 13616.
- Luhn — ISO/IEC 7812-1.
- HIPAA 18 identifiers and the Safe Harbor / Expert Determination distinction — HHS guidance on de-identification; secondary summary at https://www.strac.io/blog/hipaa-de-identification

Every validator in this appendix was executed against the stated known-valid test values before publication. The SSN, ITIN, EIN and MBI rules are **shape and range checks, not checksums**, and the text says so at each one.

Appendix B — Tooling at a Glance

The Chapter 4 and Chapter 6 data in one place. Everything here was accurate in September 2026 and will age; the accuracy figures age slowest, the maintenance status fastest.

Detection accuracy, cross-domain

From an independent benchmark across four datasets in two languages, six shared entity types, June 2026.

DETECTOR	AVG F1	BEST DATASET	WORST DATASET	DEGRADATION	MEDIAN LATENCY (CPU)
Piiranha (86M DeBERTa-v3)	0.542	0.780	0.169	−78%	118.5 ms
GLiNER v1 (209M)	0.535	0.607	0.455	0%	161.3 ms
Presidio	0.481	0.780	0.298	−17%	15.1 ms
GLiNER v2 (205M)	0.478	0.558	0.373	−20%	197.7 ms
Regex only	0.171	0.297	0.000	n/a	0.1 ms

A paired t-test found no statistically significant difference between the top three (p well above 0.10). Do not choose by decimal places.

From PIIBench, 82 entity types across ten source datasets, May 2026: a directly fine-tuned DeBERTa reached **F1 0.6476**; the strongest previously published comparator on that benchmark reached **0.1723**.

Per-entity spread

Aggregate F1 hides this, which is why Chapter 7 insists on per-entity scoring.

ENTITY	BEST OBSERVED	WORST OBSERVED	COMMENT
Email	0.996	0.553	Pattern-friendly; Presidio strongest
Phone	0.874	0.606	GLiNER v1 most consistent
Person name	0.787	0.139	The hardest category by a distance
IBAN, card	High with validation	Low without	Appendix A applies

Choosing a route from .NET

	PRESIDIO SIDECAR	AZURE AI LANGUAGE	NATIVE C# RULES
.NET access	REST via `HttpClient`	Official SDK (`Azure.AI.TextAnalytics`)	In-process
Text crosses your perimeter	No	Yes	No
Entity coverage	Broad, extensible	Broad, fixed	Checksummed identifiers only
Custom recognisers	Yes	No	Yes
Reversible pipeline	Built in	Redaction policies	Build it yourself
Latency	~15 ms plus local hop	Network round trip	~0.1 ms
You operate	A container	A subscription	Nothing
Failure mode	Container down	API error, quota	None

Presidio, in detail

Components: Analyzer (detection), Anonymizer (de-identification operators), Image Redactor (OCR), Structured (tabular and semi-structured data).

Detection method: spaCy or HuggingFace NER, plus regex recognisers, checksum validation, and contextual scoring.

Deployment: Python packages, Docker images, Kubernetes, direct Python and PySpark integration, Azure App Service, Databricks, Spark.

Governance, as of September 2026: no longer a Microsoft project. Transitioning to community ownership under the Data Privacy Stack organisation. Documentation at `presidio.dataprivacystack.org` ; the old Microsoft URL redirects twice. Much published material still calls it "Microsoft Presidio" and links to the retired address.

.NET client: none official. REST is the route.

Unofficial client: `Presidio.SDK` by StefH. Version 0.0.2, June 2025, ~5.6K downloads, 14 commits, targets .NET 6.0 and .NET Standard 2.1. Analyzer demonstrated, anonymizer not evidenced. Companion `Presidio.SDK.Extensions` adds a handful of locale-specific pattern recognisers (~970 downloads). Useful to know about, too thin to build on.

Known issue: 64% reversibility pass rate in the benchmark, from character offset misalignment. Fixable in post-processing, and a reason to assert the round-trip invariant (Chapter 9).

Azure AI Language PII, in detail

Feature types: Text PII (synchronous, string payloads), Conversation PII (turn-based transcripts, asynchronous), Document PII (native `.pdf` , `.docx` , `.txt` , asynchronous, preserves structure and emits JSON metadata).

SDKs: C#, Java, JavaScript, Python, plus REST.

Redaction: `redactionPolicies` parameter from API version 2025-11-15-preview, multiple policies per request.

Output: entity categories with confidence scores. No customisation of the model on your data.

Guidance: use GA API versions in production; do not mix payload examples across versions.

Data handling (verified 14 September 2026): no storage or processing outside your deployment region; all content encrypted at rest; up to 48 hours temporary storage for catastrophic-failure debugging, governed by `LoggingOptOut` — which **defaults to true on the PII and health endpoints**, so it does not apply to PII calls.

Choosing a safeguard (Chapter 8, condensed)

#	SAFEGUARD	REVERSIBLE	PRESERVES COREFERENCE	TYPICAL USE
1	Remove	No	n/a	Default; the field was not needed
2	Redact	No	No	One-shot classification
3	Mask	No	No	Human-facing confirmation
4	Generalise	No	No	Analytics, banded reasoning
5	Pseudonymise	Yes	Yes	Conversational work, the default
6	Surrogate	Yes	Yes	Where natural text matters
7	Format-preserving encrypt	Yes	Yes	Downstream format validation
8	Synthesise	No	n/a	Test and demo data

Effort versus reduction

CONTROL	EFFORT	REDUCES
Field projection	Hours	Every entity in the fields you drop, structurally
Turn off prompt logging	Hours	The largest personal data store most teams have
Deterministic layer	Days	Highest-consequence identifiers, near-certainty
Log sink redaction	Hours	Incidental telemetry exposure
Output host allowlist	Hours	Rendered exfiltration
Provider zero data retention	A procurement conversation	Provider-side persistence
Retrieval permission filter	Hours	Cross-tenant retrieval
De-identify at ingestion	Days	Bulk exposure in the vector index
Model-based detector	Weeks, plus operations	About half of what remains
Round trip with vault	Weeks	Real values at the provider

The ordering is the argument of the book. The most expensive row is the least effective, and it should be sized against what is left after the cheap rows, not against the original problem.

Ageing notes

Fastest to age: provider retention terms (changed at least twice during 2026), maintenance status, API preview version names, package download counts.

Slower: accuracy figures, latency characteristics, the shape of the trade-offs.

Effectively stable: checksum algorithms, the ladder of safeguards, the five doors, the fact that in-distribution scores do not predict out-of-distribution performance.

Re-verify anything in the first group before you rely on it, and print the date you checked.

Appendix C — Container Trust Checklist

Chapter 14 as a one-page checklist. Take it to a review. It is a checklist, not a hardening programme, and the difference is stated at the bottom.

The premise

The detection sidecar reads **100% of your personal data**, in plaintext, at its most concentrated. Size the controls for that, not for "it is only an internal service."

In transit

- [ ] TLS between gateway and sidecar, including on private networks and inside a single cluster.
- [ ] Certificate validation pinned to your internal CA rather than falling back to the system trust store.
- [ ] Mutual TLS where the platform provides it cheaply, since it also satisfies caller authentication below.
- [ ] No plaintext HTTP listener, even bound to localhost, even "temporarily for debugging".

At rest

- [ ] Read-only root filesystem. `read_only: true`.
- [ ] `tmpfs` for scratch, with a size limit, so a failure is a failure rather than a disk filling with user text.
- [ ] No request bodies written to disk. No spooled uploads. No model cache containing user text.
- [ ] Token vault **encrypted at rest**, with the key in a KMS.
- [ ] The service holding the vault ciphertext does **not** hold the key.
- [ ] Vault TTL matched to the conversation lifetime, not to a default retention policy.
- [ ] Backups of the vault covered by the same key separation.

Logs

- [ ] Request and response body logging explicitly **off**, not merely absent from the current config.
- [ ] Log level set in the image, not inherited from a debugging session.
- [ ] A test asserting that known personal data never reaches the log sink.
- [ ] Field deny-list at the sink: `prompt`, `completion`, `body`, `password`, `ssn`, `ein`, `routingNumber`, `accountNumber`, `dateOfBirth`.
- [ ] Deterministic redaction (Appendix A) applied at the sink, in-process.
- [ ] Log retention set deliberately. Thirty days is usually plenty.

Egress

- [ ] Outbound internet **denied at the network layer**, not in application configuration. `internal: true`, a `NetworkPolicy` with no egress rule, or a security group with no outbound rules.
- [ ] Gateway allowlisted to the model provider host only, everything else denied.
- [ ] DNS egress considered, since exfiltration over DNS does not need an HTTP route.

Provenance

- [ ] Images pinned by **digest**, not tag. `image: name@sha256:<digest>`.
- [ ] Signatures verified where the project publishes them.

- [ ] Images mirrored into your own registry.
- [ ] The maintaining organisation recorded, and **reviewed quarterly**. Presidio moved out of the Microsoft GitHub organisation to community ownership; ownership is not fixed at adoption.

Identity and least privilege

- [ ] Non-root user with an explicit UID.
- [ ] `cap_drop: [ALL]`.
- [ ] `no-new-privileges: true`.
- [ ] Its own service identity, not a shared one.
- [ ] Authenticated callers only. Network reachability is not authorisation.
- [ ] No credentials to your database, key vault, object storage, or cloud account.

Lifecycle

- [ ] Patch cadence defined and met. A detection container on an eighteen-month-old base image is a liability.
- [ ] Golden-set regression test in CI, so a model upgrade that halves recall on `PERSON` fails a build.
- [ ] Quarterly review: who maintains it, what version, when last measured, when last patched.

Gateway specifics

- [ ] Policy versioned in source control, never edited in place.
- [ ] Failure mode set per route and recorded in policy, not implied by a `catch` block.
- [ ] Degraded-mode operation counted and alerted on.
- [ ] Audit record carries **counts and types only**, never content.
- [ ] Output host allowlist applied to markdown images, links, and anything else that causes a client fetch.

The boundary

Everything above is the shallow end of a discipline with its own literature and, in several cases, its own full-time roles.

Not covered here: key management and rotation, secrets provisioning and renewal, supply-chain attestation and SBOMs, admission control and pod security standards, runtime detection, network segmentation beyond one deny rule, and threat modelling the deployment itself.

Working through this checklist gives you a detection sidecar that is defensible in a security review and materially better than most deployments of its kind. It is **not** the same as having done platform security properly, and the failure mode this page exists to prevent is telling anyone that it is.

Appendix D — Sources

Every substantive claim in this book, with its source and the date it was checked. Where a source is vendor telemetry or secondary reporting, it says so.

All URLs accessed **11 September 2026** unless stated otherwise.

Detection accuracy

Sikkema, A. "Benchmarking Open-Source PII Detection Across Domains", June 2026. https://albertsikkema.com/python/security/privacy/2026/06/01/benchmarking-open-source-pii-detection.html Source of the cross-domain F1 table, the latency figures, the in-distribution versus out-of-distribution degradation percentages, the hybrid regex gain, and the 64% Presidio reversibility result. Methodology: five approaches, four datasets (AI4Privacy EN+NL, Gretel Finance EN+NL, Nemotron-PII, CoNLL-2002 NL), six shared entity types, ±5 character span tolerance. Caveats carried in Chapter 4: three of four datasets synthetic, AI4Privacy is Piiranha's training distribution, CoNLL-2002 covers only two of six entity types, and a paired t-test found no significant difference between the top three models.

Jha, P. "Fine-Tuning Over Architectural Complexity: Broad-Coverage PII Detection on PIIBench with DeBERTa", arXiv 2605.25816, 25 May 2026. https://arxiv.org/abs/2605.25816 82 entity types across ten source datasets. Direct fine-tuned DeBERTa F1 0.6476; source-conditioned hierarchical 0.5899; curriculum variant 0.2772; strongest published comparator 0.1723. Conclusion: diverse training data and a simple weighted cross-entropy objective outperform architectural sophistication.

GLiNER2-PII: A Multilingual Model for Personally Identifiable Information Extraction, arXiv 2605.09973. https://arxiv.org/pdf/2605.09973

Vendor-claimed F1 figures of 0.92–0.99, and the reported 0.96 to 0.18 out-of-distribution drop for one commercial privacy filter, appear in secondary reporting and are labelled as such in Chapter 4. They are used to illustrate the gap between published and achieved, not as measurements this book verified.

Volume and exposure

Cyberhaven, 2026 AI Adoption & Risk Report. https://www.cyberhaven.com/blog/sensitive-data-flowing-into-ai-tools 39.7% of enterprise AI interactions expose sensitive data; 34.8% of corporate data entering AI tools is sensitive, against 27.4% and 10.7% in the two prior years; nearly 40% of uploaded files contain PII or PCI data; 22% of pasted text contains regulated information. **This is vendor endpoint telemetry, not a survey or a census**, drawn from organisations that had already deployed monitoring. Chapter 1 states this caveat in the text.

77% of employees paste into GenAI prompts; 82% of those pastes come from personal accounts. https://www.esecurityplanet.com/news/shadow-ai-chatgpt-dlp/ (secondary reporting)

Presidio

Presidio documentation, Data Privacy Stack. https://presidio.dataprivacystack.org/ Source for the four components, deployment options, detection method, and the statement that Presidio is transitioning to a community-owned project. The former `microsoft.github.io/presidio` address redirects via `data-privacy-stack.github.io/presidio` to the above.

Presidio.SDK (unofficial C# client). https://www.nuget.org/packages/Presidio.SDK · https://github.com/StefH/Presidio.SDK Version 0.0.2, published 10 June 2025. ~5.6K total downloads (4.4K on current version). 14 commits on main. Targets .NET 6.0 and .NET Standard 2.1. Companion `Presidio.SDK.Extensions` (~970 downloads) adds a handful of locale-specific pattern recognisers.

Azure AI Language

PII detection overview, Microsoft Learn. Page dated 30 June 2026, updated 3 August 2026. https://learn.microsoft.com/en-us/azure/ai-services/language-service/personally-identifiable-information/overview Three feature types, official client libraries for

C#, Java, JavaScript and Python, `redactionPolicies` from API version 2025-11-15-preview, confidence-scored entity categories, no model customisation on customer data.

Transparency note for PII detection. https://learn.microsoft.com/en-us/azure/ai-foundry/responsible-ai/language-service/transparency-note-personally-identifiable-information

Data, privacy and security for Azure Language. https://learn.microsoft.com/en-us/azure/ai-foundry/responsible-ai/language-service/data-privacy No storage or processing outside the customer's deployment region. All content encrypted at rest. Request data may be temporarily stored up to 48 hours for catastrophic-failure debugging, governed by the `LoggingOptOut` query parameter, which **defaults to true on the PII and health endpoints** and false on Language Detection, Key Phrase Extraction, Sentiment Analysis and NER. Page dated 1 April 2026, updated 17 August 2026. Verified 14 September 2026.

RAG, embeddings and agents

Embedding inversion recovering 50–70% of original input words; near-optimal reconstruction with ~1,000 samples against black-box encoders. https://www.promptfoo.dev/lm-security-db/vuln/embedding-inversion-privacy-leak-bd566a31/ Primary: *Transferable Embedding Inversion Attack*, arXiv 2406.10280. https://arxiv.org/pdf/2406.10280

OWASP Top 10 for LLM Applications — vector and embedding weaknesses. https://genai.owasp.org/initiatives/top-10-for-llm-and-genai/ · 2025 edition PDF: https://owasp.org/www-project-top-10-for-large-language-model-applications/assets/PDF/OWASP-Top-10-for-LLMs-v2025.pdf Numbered **LLM08:2025** and renumbered **LLM09:2026**. The apparent disagreement between sources is an edition change, not an error; the book gives both.

Cross-context retrieval in multi-tenant vector stores. https://www.protecto.ai/blog/how-rag-systems-sliently-expose-pii/ · https://christian-schneider.net/blog/rag-security-forgotten-attack-surface/ (both practitioner/vendor)

Prompt injection leaking personal data observed by agents during task execution, arXiv 2506.01055. https://arxiv.org/pdf/2506.01055 **Data exfiltration via backdoored tool use**, arXiv 2604.05432. https://arxiv.org/pdf/2604.05432

Guardrail classifiers and malicious-prompt success rates — a ">90% reduction" figure circulates widely in secondary reporting. The primary result could not be located at verification, so **no figure is quoted anywhere in this book**. Chapter 15 makes the point qualitatively and says why.

EchoLeak, CVE-2025-32711 — zero-click data exfiltration from Microsoft 365 Copilot via crafted email, reference-style markdown bypassing link redaction, and auto-fetched images. https://wraith.sh/learn/markdown-image-exfiltration (narrative account; cite the CVE record itself for the vulnerability)

Memorisation

Extracting memorized training data via decomposition, arXiv 2409.12367. https://arxiv.org/pdf/2409.12367 **Towards more realistic extraction attacks: an adversarial perspective**, TACL. https://direct.mit.edu/tacl/article/doi/10.1162/TACL.a.62/134536/ Thousands of email addresses, phone numbers and URLs extracted; combining attacks roughly doubles extraction risk, persisting under deduplication. Treated in Chapter 3 as a training-time risk and therefore a contract question for API consumers.

Regulation

GDPR — Article 5(1)(c) data minimisation; Recital 26 on pseudonymised versus anonymous data. Statutory.

Italian Garante €15M fine against OpenAI (decision November 2024, published December 2024). https://www.lewissilkin.com/en/insights/2025/01/14/openai-faces-15-million-fine-as-the-italian-garante-strikes-again-102jtqc

Court of Rome annulled that fine, 18 March 2026. https://www.crossborderdataforum.org/generative-ai-and-gdpr-enforcement-in-europe-a-lot-of-noise-one-fine-zero-survivors/ Chapter 16 uses this pairing deliberately. The original fine is widely cited; the annulment rarely is.

EDPB opinion on training AI models using personal data. https://www.dataprotectionreport.com/2025/01/the-edpb-opinion-on-training-ai-models-using-personal-data-and-recent-garante-fine-lawful-deployment-of-llms/

EU AI Act — application dates. Article 113: https://artificialintelligenceact.eu/article/113/ Chapters I–II from 2 February 2025. Chapter III §4, Chapter V, Chapter VII, Chapter XII and Article 78 from 2 August 2025, **expressly excluding Article 101** (fines on providers of general-purpose AI models), which therefore applies from the general date of 2 August 2026. Article 50 transparency duties applicable **2 August 2026**. GPAI models already on the market transition to 2 August 2027.

Digital Omnibus on AI — Regulation (EU) 2026/1744. ADOPTED, not a proposal. European Parliament approved 16 June 2026; Council adopted 29 June 2026; signed 8 July 2026; published in the Official Journal 24 July 2026; **in force 27 July 2026**. High-risk regime deferred to **2 December 2027** (standalone Annex III) and **2 August 2028** (Annex I, AI embedded in products already covered by EU product-safety law). Article 50 **not amended**, apart from a four-month grace period to **2 December 2026** on the Article 50(2) content-marking duty for systems placed on the market before 2 August 2026. https://www.gibsondunn.com/eu-ai-act-omnibus-agreement-postponed-high-risk-deadlines-and-other-key-changes/ · https://www.joneswalker.com/en/insights/blogs/ai-law-blog/yes-august-2-still-matters-the-eu-approved-a-high-risk-ai-delay-but-most-trans.html ⚠️ **The widely-linked** `artificialintelligenceact.eu` **Article 113 page still shows the pre-Omnibus 2 August 2027 date for Annex I.** The adopted regulation says 2 August 2028. Verified 14 September 2026.

HIPAA de-identification: Safe Harbor and Expert Determination. HHS, *Guidance Regarding Methods for De-identification of Protected Health Information in Accordance with the HIPAA Privacy Rule*: https://www.hhs.gov/hipaa/for-professionals/special-topics/de-identification/index.html Regulatory text: 45 CFR §164.514(b) — Expert Determination at (b)(1), Safe Harbor at (b)(2). HHS sets **no numeric threshold** for the "very small" risk standard; the expert defines it for the dataset and release environment and retains the justification. The 18 identifiers in Appendix A are reproduced from the standard.

Uniqueness of simple demographics. L. Sweeney, *Simple Demographics Often Identify People Uniquely*, Carnegie Mellon Data Privacy Working Paper 3, 2000: https://dataprivacylab.org/projects/identifiability/paper1.pdf — **87%** unique on ZIP + gender + full date of birth, **1990 Census data**. P. Golle, *Revisiting the Uniqueness of Simple Demographics in the US Population*, ACM WPES 2006: https://crypto.stanford.edu/~pgolle/papers/census.pdf — **63%** on the same combination using **2000 Census data**. Chapter 2 quotes the lower, more recent figure in the prose and names both. The 87% number is one of the most-repeated statistics in privacy and is almost always cited without its 1990 date, which is exactly the failure mode Chapter 4 is about.

Provider retention

OpenAI zero data retention for frontier models. https://openai.com/index/offering-zero-data-retention-for-frontier-models/
OpenAI platform data controls. https://developers.openai.com/api/docs/guides/your-data

Every claim in this section is dated September 2026 and should be re-verified before use. Provider terms changed at least twice during 2026.

HOW TO RE-VERIFY

The procedure takes about twenty minutes per provider and should run quarterly.

Go to the provider's own documentation rather than a summary, a comparison article, or an analyst note, all of which lag. Find four things: the **retention period** for API traffic, whether **API data is used for training** by default, which **regions** process the request, and whether **zero data retention** is available and on what terms. Read the enterprise or commercial terms rather than the consumer privacy policy, because they frequently differ and only one of them governs your API traffic.

Record the URL, the date, and the exact wording for each of the four. Wording matters more than summary here: "we do not train on your data" and "we do not train on your data by default" are different commitments, and so are "retained for 30 days" and "retained for up to 30 days for abuse monitoring". Put the record next to your data flow record from Chapter 17, so an auditor asking about provider terms gets an answer with a date attached.

Finally, check whether the terms you rely on are contractual or documentary. A statement in documentation can change without notice. A term in your agreement cannot.

Architecture patterns

Consistent across independent practitioner sources, presented in Chapters 12 to 13 as accepted practice rather than as a cited finding: layered detection, redaction on both request and response, self-hosted redaction driven by versioned policy, avoiding a second LLM as the detector, keeping the token vault outside model reach, and minimising at source first.

https://techcommunity.microsoft.com/blog/azuredevcommunityblog/introducing-pii-shield-a-privacy-proxy-for-every-llm-call/4514726 https://blog.logrocket.com/build-local-ai-proxy-redact-pii-before-llms/ https://predictionguard.com/blog/pii-detection-redaction-llm-pipelines-regulated-industries https://www.gravitee.io/blog/how-to-prevent-pii-leaks-in-ai-systems-automated-data-redaction-for-llm-prompt

Standards referenced

ISO 13616 — IBAN structure and the mod-97 check. **NIST SP 800-38G** — format-preserving encryption modes, with documented small-domain caveats. **NIST AI RMF** — referenced in practitioner guidance as the mapping target for audit logs.

Deliberately not cited

Two categories were excluded rather than softened.

Vendor accuracy claims presented without a dataset. A figure such as "near-99% precision" from vendor testing contradicts the independent benchmarks in this appendix and is exactly the kind of claim Chapter 4 exists to examine. Where such figures appear in the book, they are labelled as vendor claims and used as the subject of the argument.

Per-vendor retention comparison tables. These were stale within a quarter throughout 2026. Chapter 16 gives the structural point and one named example instead.