

THE CIO'S HANDBOOK

# Sovereign AI Engineering

How to own the agent stack, from the model to the browser

---

You own the code. You own the data, in place. You choose the model. One identity, every client. A harness you can inspect.

**Miguel Amigot**

MEMORARE PRESS · 2026

IBL.AI

# Sovereign AI Engineering

How to Own the Agent Stack, from the Model to the Browser

Miguel Amigot

MEMORARE PRESS · FIRST EDITION · 2026  
IBL.AI/BOOK/SOVEREIGN-AI-ENGINEERING

# Contents

Preface

## PART I — THE BILL COMES DUE

Chapter 1: Two Problems with Every AI Quote

Chapter 2: The SaaSocalypse

Chapter 3: What Sovereign Means in Engineering Terms

Chapter 4: The Lock-In You Did Not Sign

## PART II — THE MODEL LAYER

Chapter 5: Models Below the Frontier Are Commodities

Chapter 6: Routing, Not Choosing

Chapter 7: Hardware You Can Put Your Hand On

Chapter 8: The Air-Gapped Deployment

## PART III — THE DATA LAYER

Chapter 9: The Migration That Never Ends

Chapter 10: Connectors Built on MCP

Chapter 11: The MCP Broker

Chapter 12: The Ontology over Your Systems

## PART IV — THE CLIENT LAYER

Chapter 13: One Backend, Every Surface

Chapter 14: Identity Without Secrets in the App

Chapter 15: The Agent in the Browser

Chapter 16: Shared Components as Policy

## PART V — THE AGENT LAYER

Chapter 17: What a Harness Is

Chapter 18: The Five Layers of a Production Agent

Chapter 19: Headless Applications

## PART VI — RUNNING IT

Chapter 20: The Sovereign Operating Model

## FRAMEWORK AND APPENDICES

Framework: Framework: the five properties of a sovereign AI stack

Appendix A: Sovereignty Maturity Model

Appendix B: The CIO's Readiness Assessment

Appendix C: Reference Architecture

Appendix D: Hardware Sizing Tables

Appendix E: Connector and Broker Policy Catalogue

Appendix F: A 90-Day Sovereign Pilot Plan

Appendix G: Glossary of Sovereign AI Engineering Terms

## BACK MATTER

Bibliography

About the Author

About Memorare Press

About ibl.ai

# Preface

## Why a CIO Needs an Engineering Book

Every AI quote that reaches a chief information officer has two problems, and they are the same problem seen from two sides. The first is the price: a number per person per month that scales with headcount whether or not anyone uses the thing. The second is where the data goes: into a system the institution does not run, through a model it did not choose, under terms it cannot audit. Most of the industry treats these as separate conversations, one for procurement and one for security. They are not separate. Both are consequences of a single architectural decision, made by the vendor, about who owns the stack.

This book is about making that decision yourself.

I have spent the last several years building and deploying AI platforms inside universities, companies and government agencies, and the pattern I see is consistent. The organizations that get the most out of AI are not the ones with the largest budgets or the earliest pilots. They are the ones whose leaders understood, early, that the model is a commodity, that the data must stay where it lives, and that the applications their people use should be built on an identity and agent layer the institution controls. Everything else follows from those three convictions.

I call the discipline of acting on them sovereign AI engineering. Sovereign because the institution holds the code, the data, the model choice and the keys. Engineering because sovereignty is not a policy statement or a procurement clause. It is a set of design decisions about where tokens live, how a connector reaches a system of record, what a harness is allowed to do, and how a new application gets built without anyone handling a secret.

The audience is the CIO and the people who report to them: the enterprise architect, the CISO, the head of data, the director of infrastructure. I assume you have signed software contracts and regretted some of them. I assume you have been asked by a board or a cabinet what happens to the institution's data when it is sent to an AI vendor, and that you would like a better answer than the one in the vendor's trust center. I do not assume you write code, but I do assume you want to know what the engineers are deciding on your behalf.

The book is organized by layer. Part I makes the case that the current model is breaking, and defines sovereignty in terms you can verify. Part II is about models: why the frontier is not where your value is, how to route among providers, and what hardware you can actually buy and put in a rack. Part III is about data: why migration projects fail and how connectors built on the Model Context Protocol read live data through a broker that

holds the credentials. Part IV is about clients: how one backend serves the web, a Chrome side panel, desktop and mobile apps, and the terminal, all sharing one identity system that never puts a secret in an application. Part V is about agents: what a harness is, which attributes make it trustworthy, and what a headless application looks like once the subscription software it replaces is gone. Part VI is the operating model, from the first pilot to the exit clause you will never need to use.

Each chapter ends in numbers where numbers exist. Where they do not, I say so, because the second most expensive thing in this field, after lock-in, is a confident figure that nobody measured.

I have tried to write a book that will still be useful when the model names in it are forgotten. The specific products will change. The architecture, if it is built on ownership, will not have to.

Miguel Amigot New York, 2026

## CHAPTER 1

# Two Problems with Every AI Quote

*The price and the data location are not two line items. They are one decision, made by someone else.*

A quote for enterprise AI arrives on a CIO's desk looking like every other software quote of the last fifteen years. There is a per-user price, a minimum seat count, a term, and a discount schedule that rewards buying more seats than you need. Somewhere in the appendix is a data processing addendum. The two documents are reviewed by two different people, and the architecture that produced both is never discussed at all.

This chapter is about reading that quote the way an engineer reads it.

## The per-seat arithmetic

Take a mid-sized university with 14,000 students and staff who would plausibly use an AI assistant. At a list price of \$20 per person per month, the annual bill is \$3.36 million. That number does not change if half the people never log in, because the pricing unit is the seat, not the work. It does not change in the summer. It does not change if the institution's own IT team could have served the same workload for a fraction of the cost, because the institution was never offered that option.

Enterprise tiers from the largest vendors sit between \$30 and \$60 per user per month. Vertical products for specific professions run far higher; legal assistants have been quoted at several hundred dollars per lawyer per month. The structure is identical in every case: a fixed monthly fee for every person you onboard, independent of what they consume.

Now look at what the same work costs when it is priced the way the underlying technology is actually sold. Model providers charge developers per million tokens, a few dollars for input and somewhat more for output on current frontier models, and a small fraction of that for models a few months behind the frontier. A typical knowledge-worker conversation consumes a few thousand tokens. Even a heavy user, running dozens of substantial conversations a day, consumes a few dollars of tokens a month. A light user consumes cents.

The gap between those two numbers is not a margin. It is a structural difference in what is being sold. The per-seat vendor is selling access to a model behind an interface it

controls, priced to capture the value of the institution's headcount. The usage-priced alternative is selling the work actually done. Above a hundred users the two diverge by an order of magnitude; above a few thousand, by two. A SUNY team that understood this turned a six-figure multi-vendor quote into a deployment of roughly thirty thousand dollars. Syracuse University, deploying against model APIs inside its own cloud rather than buying per-seat plans, reported costs 85 percent lower than the per-seat alternative.

## **Where the data goes**

The second problem is less visible on the quote and more visible in the incident report.

When a person types a question into a hosted AI assistant, the question, any document they attach, and often the content of the page or file they are looking at, leave the institution's network and arrive at the vendor's. The vendor's model processes it. The vendor's logging stores some version of it. The vendor's terms describe, in language reviewed by the vendor's lawyers, what the vendor may do with it. For a consumer product this is a fair exchange. For a hospital, a bank, a public agency or a university holding student records under federal privacy law, it is a transfer of custody that the institution did not design and cannot inspect.

The usual answer is a contract: a business associate agreement, a data processing addendum, a certification. These documents are necessary. They are not architecture. A contract promises that data will be handled a certain way. An architecture makes it impossible to handle it any other way. A CIO who has lived through a vendor breach knows which of the two they would rather be holding.

## **Why the two problems are one**

Here is the connection that the quote obscures. The vendor can charge per seat because the vendor runs the system. The data goes to the vendor because the vendor runs the system. Both facts flow from a single architectural decision, made before the institution was ever in the room: the platform is hosted by the seller, the institution gets access, and access is what is priced.

Change that one decision and both problems change with it. If the institution runs the platform, on its own cloud account or its own hardware, the data never leaves. And because the institution runs it, there is no seat to sell; the institution pays the model providers for tokens, or runs open-weight models on its own GPUs and pays for electricity. The price becomes a function of work, and the data location becomes a function of the institution's own network diagram.

This is why I say the two problems are one decision. It is also why fixing them is an engineering project rather than a negotiation. No discount schedule turns a hosted

product into one you run. No addendum moves the data back inside your firewall. Only the architecture does that.

## **What usage pricing changes**

Usage pricing is not merely cheaper. It changes behaviour in ways that matter to a CIO.

It makes adoption free. Under per-seat pricing, every new user is a cost, so institutions ration access, run pilots with fifty people, and spend a year deciding who deserves a license. Under usage pricing, a user who never logs in costs nothing, so the platform can be offered to everyone on day one and the bill follows real use. The pilot becomes a question of what to connect, not whom to admit.

It makes the budget cap real. A pooled, rechargeable credit balance with a ceiling the institution sets cannot be exceeded. A per-seat contract can only be exceeded, because the next hire is the next seat.

It makes model choice an operating decision. When the institution pays the provider directly, switching from one model to a cheaper or better one is a configuration change with an immediate effect on the bill. When the model is bundled into a seat price, the vendor captures the savings from every improvement in the underlying technology and the institution sees none of it.

And it exposes the real cost of the work, which is the first step toward managing it. A CIO who knows that an advising conversation costs a few cents on one model and a fraction of a cent on another is in a position to make decisions. A CIO who knows only that the seat costs twenty dollars is not.

## **The question to ask every vendor**

The rest of this book builds up a definition of sovereignty that can be verified line by line. But the first question is simple, and it can be asked of any vendor before the second meeting.

Who runs it?

If the answer is "we do, and you get access," then every other promise in the quote, about data, about price, about the roadmap, about what happens at the end of the term, depends on the vendor's continued goodwill and solvency. If the answer is "you do, and we help," then the institution is buying something it can keep.

Everything that follows is an elaboration of what "you do" has to mean in practice, layer by layer, for the answer to be true rather than merely marketable.

## CHAPTER 2

# The SaaSpocalypse

*When the people who used to buy applications can build them, the application stops being the asset.*

In the trade press the argument has a name, the SaaSpocalypse, and a familiar shape: enterprises are beginning to ask whether they should keep paying for software subscriptions when AI lets them build applications themselves, and nobody is sure what happens to per-seat pricing when they stop. Like most arguments with dramatic names, the interesting part is not the prediction but the mechanism. This chapter is about the mechanism, and about the conclusion I draw from it for anyone who runs an institution's technology portfolio.

## What the debate is actually about

A software-as-a-service product bundles four things. A user interface. A data store, usually the vendor's. Business logic, encoded by the vendor's engineers. And an operating commitment: the vendor keeps it running, patched and compliant. For two decades the bundle was efficient, because building and running any one of those four was expensive, and buying all four for a monthly fee per user was cheaper than hiring the people to do it.

AI changes the cost of exactly one of the four: the business logic and the interface around it. A competent team with a coding agent can now produce a working internal application in days, where it once took a quarter. A less technical team with a capable agent platform can produce a useful one in an afternoon. The question the SaaSpocalypse asks is what happens to the price of the bundle when one of its components approaches zero.

The naive answer is that subscriptions collapse. The more careful answer is that the bundle unbundles, and the price moves to whatever is still scarce.

## What a subscription was paying for

Consider what an institution actually receives from a typical departmental SaaS product. A ticketing tool, say, or an advising platform, or a contract tracker. The interface is pleasant but generic. The logic is a set of forms and workflows that encode a reasonable default process, which the institution then spends months customizing. The data is the

institution's own, exported from its systems of record, synchronized on a schedule, and held in the vendor's database where it is at best a copy and at worst a divergent one.

The operating commitment is the part that is genuinely hard to replace. Uptime, security patches, compliance evidence, backups, an on-call rotation. But notice that this commitment attaches to the data store and the runtime, not to the forms. If the forms can be regenerated in an afternoon, what the institution needs to keep paying for is a place to run them and a place to keep the data, under the institution's own governance.

That is the first half of the conclusion. The second half concerns what the regenerated applications need in order to be useful.

## **The three things that remain scarce**

An application generated by an agent is only as good as what it can reach. Three things determine that, and none of them is produced by the agent.

The first is live access to the systems of record. An advising app that cannot read the student information system is a demo. The connector that reads it, with the right permissions, through an audited path, is infrastructure, and it is reusable across every application the institution will ever build.

The second is identity. Every application needs to know who the user is, which organization they belong to, and what they are allowed to see. If each generated app solves this on its own, the institution has a hundred login systems and a hundred places where a token can leak. If identity is a shared service that applications consume, a new app inherits single sign-on, roles and session handling without writing any of it.

The third is the agents themselves, with their memory, their tools and their guardrails. An application is increasingly a thin surface over an agent that does the work. The agent, configured and governed once, is the asset. The surface is disposable.

Data access, identity and agents. These are what remain scarce when the application itself becomes cheap, and they are exactly the things a CIO should want to own rather than rent.

## **The headless conclusion**

Put the two halves together and the conclusion is what I call the headless application. The institution owns a platform that provides the three scarce things: connectors to its systems, a shared identity layer, and governed agents. Applications are generated on top of it as needed, for a department, for a semester, for a single process, and discarded when they are no longer useful. Some have a full interface. Many have almost none: an agent reached through a chat panel, a browser extension, a terminal, or an API call from another system. The head is optional. The body is permanent.

This inverts the SaaS economics. The institution stops paying per seat for a hundred vendors' forms and pays instead, on usage, for one platform it controls, plus the model tokens the agents consume. New capability costs engineering hours, or increasingly agent hours, rather than new subscriptions. And the data stays in the systems of record where it was all along, because the headless application never needed a copy.

I do not claim this happens overnight, or that every SaaS product is replaceable. Systems of record, the ERP, the SIS, the EHR, are not applications in this sense; they are the data layer the headless applications read. Specialized tools with deep domain logic will survive, at least for a while. But the long tail of departmental software, the tools that are mostly forms over the institution's own data, is where the SaaS apocalypse argument is simply correct, and that tail is where most of a CIO's subscription spend lives.

## **What it means for a CIO's portfolio**

Three practical consequences follow for anyone managing an application portfolio today.

Stop evaluating AI vendors as if they were SaaS vendors. The question is not which product has the best features. It is which platform leaves the institution owning the connectors, the identity layer and the agents when the contract ends. A hosted assistant with a superb interface and no way to take the configuration with you is a liability dressed as a product.

Start treating connectors and identity as capital investments. Every system of record connected through an owned, audited path is reusable by every future application. Every application built on the shared sign-in is one less place for credentials to live. These compound; subscriptions do not.

And plan the application estate for churn. If generating a departmental tool takes days, the estate will have more tools, shorter-lived, built closer to the people who use them. Governance has to move down into the platform, into the broker that scopes data access and the harness that constrains what agents may do, because it cannot keep up at the level of individual application reviews. The rest of this book is largely about how to build that platform so that the churn above it is safe.

## CHAPTER 3

# What Sovereign Means in Engineering Terms

*Sovereignty is not a region on a cloud console. It is what you can still do when the vendor says no.*

The word sovereign has been stretched to cover almost anything. Cloud providers sell sovereign regions. Vendors sell sovereign tiers of their hosted product. Governments publish sovereignty strategies that amount to a preference for domestic suppliers. Each of these means something, and none of them is what a CIO needs the word to mean when deciding how to run artificial intelligence inside an institution that holds other people's data.

I want to replace the adjective with a test. Four questions, each of which can be answered by an engineer in an afternoon with access to the system, and each of which has a yes or no answer.

### Four questions you can verify

**Can you read the source?** Not a summary, not an architecture diagram, not an escrow arrangement that releases code to a lawyer if the vendor goes bankrupt. The complete source of the platform, including the backend services and the infrastructure definitions, available to your engineers now, under a license that lets them modify and run it. If the answer is no, every other sovereignty claim rests on trust.

**Can you move the data?** All of it, in a documented format, without a professional services engagement, and without losing the structure that makes it useful. More importantly, does the platform hold a copy of your data at all, or does it read your systems of record in place? A platform that never took custody has nothing to hand back.

**Can you switch the model?** Today, without a migration, by changing a configuration value. To a different commercial provider. To an open-weight model running on your own hardware. To a different model for a different data classification. If a model is wired into the product so deeply that replacing it is a project, you do not have a platform; you have a reseller.

**Can you run it with the cable unplugged?** Literally: can the entire system operate on a network with no route to the public internet? This is the strongest test because it subsumes the others. A system that runs air-gapped necessarily carries its own code, its own data path and its own models. Many institutions will never run that way. All of them should know whether they could.

Four yeses is sovereignty. Three is a negotiation. Fewer is a hosted product with a sovereignty section in the brochure.

## The deployment tiers

In practice sovereign systems run in one of four ways, and the choice is a business decision rather than an architectural one, provided the architecture supports all four.

**The vendor's cloud.** The platform runs on infrastructure the vendor operates, but the institution owns the code and can leave with its configuration and data at any time. This is the fastest start and the right choice for a pilot, a small institution, or a department proving value before the central IT organization commits. The institution gives up nothing structurally; it defers the operating commitment.

**The institution's own cloud account.** The same platform deployed into the institution's AWS, Google Cloud or Azure tenancy, including government regions where required. Data never leaves an account the institution controls. The institution's security team applies its own controls, logging and identity policies. This is where most universities and companies end up, and where Syracuse, among others, runs today.

**On-premise.** Physical servers in the institution's data centre, usually chosen because of data classification, existing investment in hardware, or regulatory requirements that treat any cloud as a transfer. The GPU question becomes concrete here, and Chapter 7 addresses it directly.

**Air-gapped.** No outbound network at all. Models run locally, updates arrive as signed bundles, and nothing phones home. Defence, intelligence, some financial desks, some research facilities. Chapter 8 is a walk-through.

## One codebase, four tiers

The engineering requirement that makes the tiers a business decision rather than four products is that they share one codebase. The platform that runs in the vendor's cloud must be, byte for byte, the platform that runs in the institution's account, on its servers, and behind its air gap. The differences are configuration: which model endpoints to call, which identity provider to trust, which network routes exist.

This matters for three reasons. The institution can start on the vendor's cloud and move inward without a migration, because nothing about the application changes. The vendor

cannot quietly degrade the self-hosted version, because it is the same version. And the air-gapped tier, which almost nobody uses, keeps everybody honest, because any dependency on a vendor service would break it, and so no such dependency is allowed to exist.

A platform with a "cloud edition" and an "enterprise edition" that differ in code has already failed this test. The institution is being sold two products and told they are one.

## **What sovereignty is not**

Sovereignty is not a data residency promise. Keeping data in a particular country's region of a hyperscaler tells you where the disks are. It tells you nothing about who can read them, which model processes the data, or whether you can leave.

It is not an escrow clause. Code released from escrow on a vendor's bankruptcy is code nobody on your team has ever run, delivered at the moment you are least able to learn it.

It is not open source by itself. Open source tells you that you may read and modify the code. It does not tell you whether the system can run without the vendor's hosted services, whether the model can be switched, or whether the data was ever yours. Many open-source AI products are a free client for a paid cloud.

And it is not a preference for domestic vendors. A foreign vendor whose platform passes all four questions leaves you more sovereign than a domestic one whose platform passes none. Where the company is headquartered matters, and I will say later why I think it does, but it is a separate question from whether you own the system.

## **A worked example**

Consider an institution deciding between two offers for a campus-wide AI assistant.

Offer A is a hosted product from a large vendor. The interface is excellent. The data processing addendum is thorough. The price is \$40 per user per month with a 5,000-seat minimum, \$2.4 million a year. The vendor's model is the only model. The vendor's cloud is the only cloud. Source is not available. Export is a support ticket.

Offer B is a platform the institution deploys into its own cloud account. The source is delivered under a perpetual license. The institution connects its own model providers, pays them directly for tokens, and can add an open-weight model on its own GPUs for sensitive workloads. Connectors read the student information system in place. The price is a platform fee that does not depend on headcount, plus the tokens actually consumed, which for 5,000 real users of a campus assistant amounts to a small fraction of Offer A's figure.

Offer A fails all four questions. Offer B passes them. That is the whole evaluation, and it took one paragraph. Everything else, the feature comparison, the demo, the references, is detail. Useful detail, but detail. The architectural decision is the one that will still matter in year five, and it is the one the quote was designed to keep off the table.

## CHAPTER 4

# The Lock-In You Did Not Sign

*An exit clause is a promise about the future. An exit architecture is a fact about the present.*

Lock-in used to be something you could see coming. A proprietary database, a closed file format, a licensing scheme that punished migration. CIOs learned to spot it and negotiate around it. AI platforms introduce four new kinds, and three of them are invisible on a contract because they are not contractual at all. They are consequences of how the system was built, and they bind the institution whether or not anyone agreed to them.

## Model lock-in

The most obvious and the least discussed. A platform that calls one provider's models, through that provider's proprietary API, with prompts tuned to that provider's quirks, cannot change providers without re-engineering. The institution is exposed to that provider's pricing, its outages, its policy changes, and its decisions about which models to retire. When a better or cheaper model appears elsewhere, and in this market one appears every few months, the institution cannot take advantage of it.

The engineering fix is well understood and widely ignored: every model behind one interface, every prompt written against that interface, and the specific provider a configuration value. Most of the industry has converged on a common request format, so a platform that speaks it can address dozens of providers and any open-weight model served by any of the standard inference engines. If your platform does this, switching models is a change to a settings page. If it does not, ask why.

The subtler form of model lock-in is capability drift. A platform that has quietly come to depend on one provider's particular strengths, its long context, its tool-calling style, its image handling, will work worse on any other model even if the interface is standard. The defence is to test. A platform that routinely runs its evaluation suite against several models, and publishes the results to its administrators, is one whose model independence is real rather than nominal.

## Identity lock-in

Every AI product has to know who the user is. The hosted ones solve this by becoming an identity provider: users get accounts in the vendor's system, and the institution's directory is synchronized into it. Over time, roles, groups, permissions and sharing settings accumulate in the vendor's identity store. The agents people have built, the conversations they have had, the memory the system holds about them, are all keyed to identities the vendor owns.

Leaving means re-establishing all of that somewhere else. In practice institutions do not leave, and the vendor knows it.

The sovereign alternative is an identity layer the institution runs, which every application consumes. Users sign in once, through the institution's own single sign-on if it has one, and receive tokens scoped to themselves and their organization. Applications, whether built by the vendor, by the institution, or by an agent in an afternoon, trust that layer and hold no identity of their own. Chapter 14 describes how this is engineered so that applications never touch a secret. The point here is that identity is the layer that makes everything else portable, and it must be the first one the institution owns.

## **Connector lock-in**

Connecting an AI system to an institution's data is expensive. Someone has to understand the student information system's data model, or the ERP's, or the clinical system's, write the integration, handle authentication, map the fields, and keep it working through upgrades. Vendors know this, and a growing number offer their connectors as the product's main attraction.

The trap is that connectors built inside a vendor's platform belong to the vendor's platform. They encode months of institutional knowledge about the institution's own systems in a form that cannot be taken anywhere else. The second AI product the institution adopts starts the integration work from zero.

The engineering escape is a connector standard. The Model Context Protocol, introduced by Anthropic in late 2024, adopted by the other major model providers through 2025 and since placed under the Linux Foundation's stewardship, defines how an agent discovers and calls tools, including tools that read and write institutional systems. A connector written to the protocol works with any agent platform that speaks it. The institution's investment in understanding its own systems becomes portable. Part III is about building on this foundation and, crucially, about the broker that sits between the agents and the connectors so that credentials and policy never live in the agent itself.

## **Format and memory lock-in**

Two quieter forms deserve a paragraph each.

Conversations, documents, agent configurations and evaluation results accumulate in whatever format the platform chose. If that format is proprietary, exporting it yields something that is technically complete and practically useless. The sovereign requirement is that every artefact the platform holds is available in a documented, open format: agent definitions as readable files, conversations as structured records, evaluations as data a spreadsheet can open.

Memory is the newest form and potentially the stickiest. As agents remember what each person has told them, which preferences they hold, which problems they are working on, the memory store becomes the thing that makes the system feel personal. A vendor that holds the memory holds the relationship. A sovereign platform keeps memory in the institution's own database, in rows an administrator can read, correct and delete, scoped per person and per agent, and exportable like everything else.

## **Writing the exit into the architecture**

A good contract has an exit clause: notice periods, data return, transition assistance. These are worth negotiating. But a clause is a promise about future behaviour, enforced by lawyers after the relationship has already soured. The sovereign approach is to make the exit a fact about the present, verifiable at any time, by building it into the architecture.

Concretely: the institution runs the platform, or could tomorrow, because it has the code and the deployment definitions. The identity layer is the institution's. The connectors are written to a standard and the broker that governs them is the institution's. The models are configuration. The data was never copied. Memory lives in the institution's database.

An institution built this way can end a vendor relationship by, at most, replacing the people who help run the platform. Nothing else has to change. That is the only exit clause that cannot be litigated, because there is nothing to litigate. The system simply keeps running.

The following parts of this book are about each layer in turn, because each of the four kinds of lock-in is defeated by a specific engineering decision at a specific layer, and a CIO who knows which decision to look for can tell in a meeting whether a vendor has made it.

## CHAPTER 5

# Models Below the Frontier Are Commodities

*The frontier is where the headlines are. The work is done a few months behind it, at a hundredth of the price.*

Every quarter a new model arrives that is, by some benchmark, the best in the world. The announcement is covered as if it mattered to the institution running an advising assistant or a contract-review agent. Usually it does not. This chapter is about why, and about the single architectural decision that follows from it.

## What the frontier is for

Frontier models earn their name by doing things that were previously impossible: long multi-step reasoning, research-grade synthesis, code that compiles on the first try across an unfamiliar codebase. A small number of workloads in any institution genuinely need that. A research group pushing on a hard problem. An engineering team using an agent to refactor a system. A one-off analysis where the cost of being wrong is high and the volume is low.

The overwhelming majority of institutional AI work is not that. It is answering a student's question about a degree requirement from a document the institution already has. It is drafting a first version of a letter. It is extracting fields from a form, summarizing a meeting, classifying a ticket, translating a page. These tasks were solved, to a standard most people cannot distinguish from the frontier, by models that are now two or three generations old. The frontier matters to them only in the sense that it keeps pushing the floor upward.

## The six-month rule

A rule of thumb has held for several years now and shows no sign of breaking: whatever the best model can do today, a model costing a tenth as much will do adequately in about six months, and an open-weight model you can run on your own hardware will do adequately in about a year.

The mechanism is competitive. There are now at least a dozen serious model developers, commercial and open, and each new capability is matched by the others within months.

Prices for a given level of capability have fallen by roughly an order of magnitude a year since 2023. Open-weight releases from Meta, Mistral, DeepSeek, Qwen and others have repeatedly reached the capability of commercial models from a year earlier, and they can be downloaded and run by anyone with the hardware.

For a CIO the consequence is that the question "which model should we standardize on" is malformed. Any answer is wrong within two quarters. The right question is "how do we make sure we can use whichever model is good enough and cheapest for each task, as that changes."

## Pricing power and where it goes

When a model is a commodity, pricing power leaves the model and moves to whoever controls the interface to the user. That is the economic logic behind every per-seat AI product. The vendor buys tokens at commodity prices, or trains its own model, and sells access to it at a price unrelated to token cost, because the institution is not buying tokens. It is buying a chat window, a set of connectors, an identity store and a brand, and the model inside is interchangeable.

Follow the money and the lesson is plain. If the institution owns the interface, the connectors and the identity store, then the institution captures the falling price of models. Every time a cheaper or better model appears, the institution's bill goes down or its capability goes up, by changing a setting. If the vendor owns those things, the vendor captures the savings and the institution's bill stays flat.

This is not a small effect. An institution paying a per-seat price negotiated in 2024 is, in 2026, paying for a model that costs the vendor a small fraction of what it cost two years earlier, at the same price. The difference is margin. A sovereign deployment, paying providers directly, would have seen most of that decline on its own invoice.

## The design consequence

The architecture that captures commodity pricing has one rule: the model is a setting, not a dependency.

Every call to a language model passes through one interface inside the platform. That interface speaks a standard request format, which almost every commercial provider and every mainstream open-weight inference engine now accepts. Prompts are written against that interface, not against a specific model's quirks. The provider, the model name, the endpoint and the credentials are configuration values an administrator can change for the whole platform, for one agent, or for one class of request, without touching code.

Three practices make the rule real rather than nominal.

The platform maintains a catalogue of available models with their prices, context limits and capabilities, refreshed from the providers, so an administrator chooses from facts rather than memory. Each agent carries an evaluation set, a few dozen representative prompts with expected properties of a good answer, that can be run against any model on demand. And the default model for routine work is chosen deliberately as the cheapest one that passes the evaluations, with the frontier reserved for the agents that have shown they need it.

A platform built this way treats a new model release the way a procurement office treats a new supplier quote: run the evaluations, compare the price, switch if it wins. A platform built around one provider treats a new release as news.

## **How to test the claim in your own house**

A CIO does not have to take the commodity argument on faith. It can be tested in a week with the institution's own workloads.

Take the three agents or use cases that generate the most volume. Collect fifty real prompts for each, with the answers people considered good. Run those prompts through the frontier model the institution is using or being sold, through a model one generation older from the same provider, through the cheapest capable commercial model from a different provider, and through a leading open-weight model served on a rented GPU. Have the people who own the use case grade the answers blind.

In my experience the result is consistent: for routine institutional work, the graders cannot reliably tell the frontier from the tier below it, and the open-weight model is acceptable for most tasks and clearly worse on a few. The price difference across the four is typically fifty to a hundred times.

That exercise does two things. It tells the institution what it actually needs, which is usually less than it is being sold. And it produces the evaluation set that makes model switching a routine operation thereafter. An institution that has done it once is in a different negotiating position with every vendor it meets, because it knows what the model inside is worth.

The next chapter is about what to do with that knowledge: routing among models continuously, rather than choosing one.

## CHAPTER 6

# Routing, Not Choosing

*The best model is a different model for each request. The platform's job is to know which.*

The previous chapter argued that models are commodities and that the architecture must treat them as settings. This one takes the next step. Once switching is free, there is no reason to switch everything at once. A sovereign platform routes each request to the model that suits it, and changes the routing as the market changes. This chapter is about what that routing considers, and what it looks like from an administrator's chair.

## Four dimensions of a routing decision

Every request an agent makes to a model can be scored on four axes, and a routing policy is a rule over those axes.

**Cost.** Providers price per million tokens, input and output separately, and the spread between the cheapest capable model and the most expensive is two orders of magnitude. A platform that knows the price of each model and the expected size of each request can estimate the cost of a decision before making it.

**Latency.** A model behind a chat window has a few seconds; a model behind a voice call has a few hundred milliseconds; a model running a nightly batch job has all night. Smaller models are faster. Models hosted nearby are faster than models across an ocean. The right choice for a voice agent is wrong for a research summary.

**Capability.** Some requests need long context, tool calling, image understanding, or reasoning across many steps. Most need none of these. The routing policy sends the hard requests to models that can handle them and the easy requests to models that are cheap.

**Data classification.** This is the axis the other platforms do not have, and the one that makes routing a sovereignty tool rather than a cost optimization. A request that contains student records, patient data, privileged legal material or classified information may be allowed to reach only certain models: those running on the institution's own hardware, or those under a contract that permits it, or none at all outside the perimeter.

A routing rule is a sentence like: "Requests from the admissions agents go to the institution's own open-weight model if they contain applicant data, otherwise to the

cheapest commercial model that passes the admissions evaluation set, with the frontier model as a fallback if the first attempt fails its own check."

## Routing by data classification

The fourth axis deserves its own section because it is where policy becomes enforceable.

Most institutions already classify data. Public, internal, confidential, restricted; or some local equivalent. Most AI deployments ignore that classification entirely, because the model is fixed and the only choice is whether to use it. A routing layer changes that. The connector that fetches data can tag it with its classification. The agent's context carries the tag. The router refuses to send restricted-tagged context to any model not on the restricted-approved list.

This turns a governance document into a system behaviour. The policy that says "patient data does not leave the network" is no longer a training slide; it is a rule the router applies to every request, and an audit line records every decision. When the CISO asks whether any protected data has reached an external model in the last quarter, the answer is a query, not an investigation.

The approved list for restricted data will usually be short: models running on the institution's own GPUs, which Chapter 7 covers, and perhaps a commercial provider under a specific contract in a specific region. That is fine. Restricted workloads are a minority, and the six-month rule means the open-weight models that can run locally are good enough for most of them.

## Fallbacks and outages

Commercial model providers have outages, rate limits and quiet deprecations. A platform with one provider inherits all of them. A routing layer with a fallback chain does not.

The simplest useful policy is a primary model, a secondary from a different provider, and a tertiary running locally. A request that fails at the primary, with a timeout, a rate-limit response or a content refusal that the platform recognizes as a provider-side policy rather than a real problem, retries at the secondary without the user noticing. If every external provider is down, the local model answers, perhaps less well, but the institution's services do not stop.

The same chain handles deprecation. When a provider retires a model, the platform's catalogue marks it, the evaluation suite runs against the proposed replacement, and the routing rule is updated. Agents built against the standard interface never know.

## Open-weight models in the mix

Open-weight models change the routing calculation in two ways beyond data classification.

They have no per-token price. The cost is hardware and electricity, which means the marginal cost of one more request is close to zero once the hardware is bought. For high-volume, low-stakes workloads, a classifier running on every incoming email, a summarizer running on every meeting, a local model can be dramatically cheaper than any commercial API, and the routing policy should send that traffic there by default.

They are stable. A commercial model can change behaviour overnight when the provider updates it. An open-weight model pinned to a specific version behaves identically until the institution decides otherwise. For workloads where consistency matters more than peak capability, regulated reporting, anything a human has signed off on, that stability is worth more than a few benchmark points.

The practical pattern in most sovereign deployments is a mix: one or two commercial providers for the hard problems and the long tail, one or two open-weight models on local hardware for the volume, the restricted data and the fallback. The routing layer makes the mix invisible to the people using the agents and visible to the people paying for them.

## **What the administrator sees**

None of this matters if it requires an engineer to operate. The routing layer has to be a surface an administrator can use.

In practice that means a model catalogue with live prices and capabilities, per provider. A per-agent setting that chooses the default model, with the evaluation results beside each option. A classification policy table that maps data tags to allowed models. A fallback chain, in order. A spend view that shows tokens and cost by agent, by model and by day, against the budget cap the institution set. And an audit log, filterable by classification tag, that shows which model saw which request.

An administrator with that surface can run the institution's model strategy as an operating function: review the catalogue monthly, re-run the evaluations when something new appears, move traffic when the numbers justify it, and answer the governance questions with data. That is what "model-agnostic" means when it is a property of the system rather than a line in a brochure. The next chapter turns to the hardware the local half of that mix runs on.

## CHAPTER 7

# Hardware You Can Put Your Hand On

*Dedicated compute means your rack, not a GPU under someone's desk.*

The previous two chapters assumed that an institution can run open-weight models on hardware it controls. This chapter is about that hardware: what to buy for which scale, what the honest limits are, and how to read a specification sheet without being misled. I will name products because a CIO has to buy specific things, and I will give the numbers that determine fit rather than the ones that sell.

## Why memory, not FLOPS, is the number

Marketing for AI hardware leads with floating-point operations per second. For running a language model to serve people, the number that matters first is memory, and the number that matters second is memory bandwidth.

A model has to fit in memory to run at all. A model with 70 billion parameters needs roughly 140 gigabytes at 16-bit precision, 70 at 8-bit, and about 35 to 40 at the 4-bit quantizations that have become standard for serving, with a modest quality cost. A 400-billion-parameter model needs five to six times that. If the machine does not have the memory, no amount of compute helps; the model does not load.

Once the model fits, the speed at which it produces each token is bounded, for a single conversation, by how fast the machine can read the whole model from memory once per token. Divide memory bandwidth by model size in bytes and you have a ceiling on tokens per second for one user. A machine with 800 gigabytes per second of bandwidth serving a 40-gigabyte model cannot exceed about 20 tokens per second for one conversation, regardless of how many FLOPS it advertises. Serving many users at once amortizes this, which is why the throughput figures for data-centre systems look so different from the figures for a desktop.

With those two numbers in hand, the hardware market sorts itself into three tiers.

## The department tier: Apple silicon

Apple's desktop machines have become a serious, slightly accidental, option for local inference because they put a large pool of fast unified memory next to a capable GPU in a quiet box that plugs into a wall socket.

A Mac mini with an M4 Pro processor can be configured with 64 gigabytes of unified memory and offers memory bandwidth in the high two hundreds of gigabytes per second. That is enough to run quantized models in the 30-billion-parameter class comfortably, for a handful of simultaneous users, at conversational speed. It costs about what a well-equipped laptop costs and fits on a shelf.

A Mac Studio with an M3 Ultra can be configured with up to 512 gigabytes of unified memory and roughly 800 gigabytes per second of bandwidth. That memory ceiling is the remarkable part: it will load quantized models in the several-hundred-billion-parameter class that otherwise require a multi-GPU server. Single-conversation speed on the largest models is modest, a handful of tokens per second, but for a department that wants a capable private model for a few dozen people, with no data leaving the building, it is a credible answer at a price that does not require a capital committee.

Honest limits: Apple's platform is not where the inference-serving ecosystem is most mature, concurrency is limited compared with data-centre GPUs, and there is no path to scale a single machine beyond its box. It is a department tier, and a good one.

## **The team tier: DGX Spark and DGX Station**

NVIDIA's desk-side systems bring the data-centre software stack, CUDA and everything built on it, to a form factor a team can own.

DGX Spark is a compact system built around the GB10 Grace Blackwell chip with 128 gigabytes of unified memory. NVIDIA positions it for models up to about 200 billion parameters at reduced precision on one unit, and two units can be linked to roughly double that. It draws ordinary wall power. For a team that wants to develop, fine-tune and serve mid-sized models on the same software it will later deploy in a data centre, it is the natural first purchase.

DGX Station is the larger desk-side system, built around the GB300 Grace Blackwell Ultra, with roughly 784 gigabytes of coherent memory across the CPU and GPU. It is a workstation in the sense that it sits in an office, and a server in every other sense. It will load the largest open-weight models at useful precision and serve a team with real concurrency.

The team tier is where I would point most institutions beginning a sovereign deployment that needs local inference for restricted data. It is enough compute to matter, it runs the same software as the production systems, and it can be bought, installed and returned without a procurement cycle measured in quarters.

## The institution tier: DGX in the data centre

Above the desk sit the rack-scale systems: multi-GPU DGX servers and the larger configurations built from them, in the institution's own data centre or a colocation facility it controls. This is where an institution serves thousands of concurrent users from open-weight models, fine-tunes on its own data at scale, and runs the restricted-classification workloads for a whole enterprise.

The economics at this tier are a straightforward capital-versus-operating comparison. A rack of GPUs is a large purchase with a multi-year life and a predictable power bill. Against it stands the commercial token price for the same volume of work, which is falling, and the per-seat price, which is not. For institutions with the volume, the data classification requirements, or both, the rack pays for itself; for institutions without them, the team tier plus commercial APIs is the better spend. The routing layer from Chapter 6 is what lets an institution start at one tier and move to another without changing anything above the hardware.

## Routers across machines

Institutions rarely have one machine. They have a Mac Studio in one department, a Spark in another, a GPU workstation in a lab, and increasingly a desk-side system or two. Software now exists to treat them as one pool.

NVIDIA's Personal AI Router, released as open source in September 2026, is a representative example. It is a router rather than a cluster: it sends each independent request to one eligible machine and never splits a model across machines or pools their memory. Nodes can be RTX-class GPUs, DGX Spark, or Apple M4-class systems. Its value is not raw speed; a reference three-device group was measured in minutes for a heavy workload, not seconds. Its value is utilization: idle machines across an institution become an inference pool for the volume workloads that do not need the data centre, and the routing policy decides what goes there.

## What to budget and what not to believe

Three rules for reading a hardware proposal.

Ask for memory capacity and bandwidth first, then concurrency at a stated model size and precision, then price. A proposal that leads with FLOPS and a benchmark headline is selling something other than serving capacity.

Do not believe single-user token rates as a measure of what a machine will do for a team. Ask for throughput at ten, fifty and two hundred concurrent conversations at the model you intend to run, and ask how it was measured. If the vendor cannot answer, run the test yourself before buying; the models and serving software are free.

And budget for the software and the people, not just the box. A machine serving models needs an inference server, monitoring, a model catalogue, access control and someone who updates it. A sovereign platform supplies most of that; the institution supplies the person. Appendix D sets out sizing tables for the three tiers against users, model class and latency, with the arithmetic shown so it can be redone when the next generation of hardware arrives, which it will.

## CHAPTER 8

# The Air-Gapped Deployment

*Almost nobody will run it with the cable unplugged. Everybody should know that they could.*

An air-gapped deployment is a sovereign AI platform running on a network with no route to the public internet. No model API calls leave. No telemetry leaves. Updates arrive on media or across a controlled transfer point, signed and verified. It is the strictest of the four deployment tiers from Chapter 3 and the one that tests every claim a platform makes about itself. This chapter walks through one, so that a CIO who will never run it understands what it proves about the platform that can.

## Who needs it

Defence and intelligence environments, obviously. Beyond them: certain financial trading and risk desks whose regulators treat any external connection as a reportable event. Research facilities working with export-controlled material. Hospitals or public-health agencies with specific data sets they have chosen never to put on a connected network. Critical-infrastructure operators whose control networks are segregated by design. Law firms running matters for clients who require it.

The common thread is that the data classification rules out connected operation, not that the organization is unusual in any other way. The people inside still want an agent that answers questions from their documents, drafts, summarizes and reasons. They want the same platform as everyone else, minus the cable.

## What has to be inside the perimeter

Everything. That is the discipline, and it is useful to enumerate it, because every item is something a connected deployment might have quietly outsourced.

**The models.** Open-weight models on local GPUs, served by an inference engine inside the network. The model catalogue from Chapter 6 lists only these. There is no commercial provider in the routing table, because there is no route to one.

**The embeddings.** Retrieval over the institution's documents depends on an embedding model that turns text into vectors. In a connected deployment this is often a

cheap API call. Inside the gap it is another local model, smaller than the chat models, running on the same hardware.

**The vector store and the database.** Memory, documents, conversations, agent configurations and audit logs live in databases the institution runs. A platform that depends on a managed vector database or a hosted analytics service fails here.

**Identity.** The single sign-on and the identity layer from Part IV must work against the institution's own directory with no external identity provider in the chain.

**The connectors.** The systems of record the agents reach are inside the perimeter by definition, so the connectors and the broker that governs them run there too.

**The clients.** Web, desktop, browser extension and terminal all point at the internal platform. For a browser extension this means building the extension against the internal host rather than the public one, which is a single configuration value at build time in a platform designed for it, and impossible in one that was not.

**The build itself.** The platform's source, its dependencies and its container images have to be available inside the network to be built, patched and redeployed. A platform that pulls dependencies from the internet at deploy time cannot be deployed.

## Updates by signed bundle

A sealed network still needs new versions. The pattern that works is a release bundle: the platform's images, the model weights, the dependency archives and a manifest, cryptographically signed by the vendor and verified by the institution before anything is loaded. The bundle crosses the gap on approved media or through a one-way transfer device. Nothing inside the network ever reaches out to check for updates; the institution decides when a bundle enters and verifies what it contains.

This sounds exotic and is not. It is how software has reached secure facilities for decades. What is new is that the bundle now includes model weights measured in tens or hundreds of gigabytes, and that the platform has to be able to swap a model in and out, through the same catalogue and evaluation process as a connected deployment, without any external dependency.

## What you give up

Honesty about the costs matters more here than anywhere.

You give up the frontier. The best commercial models are not available inside the gap, and the open-weight models that are will lag them by six months to a year on the hardest tasks. For most institutional work this is invisible; for the small set of workloads that

genuinely need the frontier, the air-gapped environment will be slower or worse, and the institution should decide knowingly.

You give up convenience. Every model, every update and every new connector arrives through a controlled process rather than a download. The people running the platform need more skill and more time.

You give up some breadth. Connectors to external services, web search, cloud-hosted tools of any kind, do not exist. The agents work with what is inside.

You pay for hardware. The GPU tiers from Chapter 7 are not optional; they are the only place a model can run.

What you keep is total certainty about where the data is. For the organizations on the list above, that is the entire point, and it is worth every item on the list.

## **Why the same codebase must run both ways**

The air-gapped deployment is rare. Its importance is out of proportion to its frequency, because it is the proof that the platform has no hidden dependencies.

A platform whose code is identical in the vendor's cloud and behind an air gap cannot be secretly sending telemetry, because there is nowhere for it to go. It cannot depend on a managed service, because the service would be unreachable. It cannot have a model baked in, because the model would have to be swappable for one that exists inside. Every property that makes the sealed deployment possible is a property the connected deployments inherit for free.

That is why I advise CIOs who will never unplug the cable to ask the air-gap question anyway, and to ask for a demonstration rather than a slide. Watch the platform start on a machine with networking disabled. Watch an agent answer from a local document through a local model. Watch an administrator switch that model to another local one. A vendor that can do this has built a sovereign system. A vendor that cannot has built a hosted one with an on-premise brochure, and the institution will find out which at the worst possible moment.

Part III turns from the model to the data: how agents reach systems of record where they live, through connectors and a broker, without the migration projects that have consumed institutional IT budgets for a generation.

## CHAPTER 9

# The Migration That Never Ends

*Every AI data platform project becomes an ETL project, and every ETL project is stale the day it ships.*

Ask a vendor how their AI system will know about your students, your customers, your contracts or your patients, and the answer almost always involves a word that should alarm a CIO: ingest. Your data will be ingested. Exported, transformed, loaded, indexed and embedded into the vendor's store, where the model can reach it. The pitch frames this as a one-time step. It is never one time. This chapter is about why, and about the alternative that the Model Context Protocol has made practical.

## How the migration starts

The first conversation is reasonable. The AI needs context. The context is in the student information system, the ERP, the CRM, the document management system, the ticketing tool. None of those systems was designed to be queried by a language model. So someone proposes a data platform: a central store into which the relevant data is copied, cleaned and shaped so that the AI can use it.

A project is scoped. Six systems, four months, a team of three, plus the vendor's professional services. The budget is approved because the alternative, an AI that knows nothing, is clearly useless.

Then the real work begins. Each source system has a data model that nobody fully documents. Each has permissions that the copy must somehow preserve. Each changes with every upgrade. The four months become nine. The team of three becomes five. And on the day the pipeline finally runs clean, it begins to decay, because the source systems keep changing and the copy does not know.

## What it actually costs

Institutions rarely total the cost of these projects, because it is spread across budgets and years. When they do, the pattern is consistent.

The build is the smaller part. The larger part is maintenance: every schema change upstream breaks something downstream; every new field someone needs requires a pipeline change; every permission model drift creates a compliance question about who

can now see what in the copy. Teams that set out to build an AI capability end up running a data integration function, and the AI work waits.

Then there is the second copy. The institution already has a data warehouse for reporting. The AI platform wants its own store, shaped differently. Now there are two copies, two pipelines, two sets of permissions to reconcile, and two answers to any question about what the institution's data says.

And there is the exit cost from Chapter 4. The pipelines and the shaped data live in the vendor's platform. Leaving means rebuilding them.

## **Why the copy is always wrong**

Even a well-maintained copy is wrong in three ways that matter to an agent acting on behalf of a person.

It is late. A nightly sync means an agent answering a question at 9 a.m. is reading yesterday's data. For a student asking whether a payment posted, a nurse asking about a medication order, or a customer asking about an order status, yesterday is wrong. Streaming pipelines reduce the lag at the cost of complexity, but they do not eliminate it.

It is permissionless. The source system knows that this advisor may see these students and not those. The copy, almost always, flattens that. Either the AI sees everything in the copy, which is a breach waiting for a prompt, or someone reimplements the source system's permission model in the copy, which is a second system to keep correct.

It is partial. The pipeline copied the fields someone thought the AI would need. The question a real person asks needs a field nobody anticipated. The answer is a pipeline change, a ticket and a wait, and the agent says it does not know in the meantime.

## **The case for reading in place**

The alternative is to leave the data where it is and let the agent reach it at the moment of the question, through the source system's own interface, with the asking person's own permissions.

This was impractical when every integration had to be hand-built for every AI tool. It became practical when the industry converged on a standard for how an agent discovers and calls tools. The Model Context Protocol, introduced by Anthropic in November 2024, adopted by the other major model developers over the following year and now stewarded by the Linux Foundation, gives a connector a standard shape. A connector to the student information system, written once, can be used by any agent on any platform that speaks the protocol. Chapter 10 describes the connectors in detail and Chapter 11 the broker that governs them.

What reading in place buys, concretely:

The data is current, because the agent asks the source system now. The data is permissioned, because the connector calls the source system as the person who asked, or with a scope the broker derives from their role, and the source system applies its own rules. The data is complete, because any field the source exposes is reachable without a pipeline change. And there is no copy to govern, secure, reconcile or hand back when the vendor relationship ends.

## **What changes when the data stays put**

A CIO who adopts reading in place as the default will notice four changes in how AI projects run.

The first project gets faster. There is no ingestion phase. The team connects one system, through a connector that may already exist, and the agent is useful against live data within days. Syracuse University's first agent started with easily accessible materials and connected additional systems phase by phase, useful from the first week rather than after a migration.

The data team gets its time back. Connectors are maintained against the source system's interface, which the source vendor keeps stable because their own clients depend on it, rather than against a bespoke pipeline.

Governance gets simpler, not harder. The question "what can this agent see" has the same answer as "what can this person see," enforced by the systems that already enforce it, with an audit log of every call at the broker.

And the institution keeps its leverage. Connectors written to a standard belong to the institution and work with the next platform too. The vendor that owns no copy of the data has no hold over the institution that the contract did not grant.

There is one caveat. Reading in place depends on the source systems having usable interfaces, and some do not. A legacy system with no API still needs a copy, or an interface built in front of it. The sovereign approach is to make that the exception, scoped to the one system that needs it, rather than the default architecture for everything. The next two chapters describe how the default works.

## CHAPTER 10

# Connectors Built on MCP

*A connector is a protocol, not an integration. That difference is the whole investment thesis.*

The last chapter made the case for reading data in place. This one is about the mechanism: connectors built on the Model Context Protocol, what they are, why the protocol matters more than any individual connector, and what a CIO should expect a sovereign platform to ship.

## What a connector is

A connector is a small service that stands between an agent and a system of record. It exposes the system as a set of tools the agent can call: look up a student by ID, list a customer's open invoices, search contracts by clause, create a ticket. Each tool has a name, a description written for a language model to read, a typed set of inputs, and a typed output. The agent sees the descriptions, decides which tool answers the question in front of it, calls it with the right inputs, and reads the result.

The connector translates between the model's world and the system's. On one side it speaks the protocol; on the other it speaks whatever the system speaks: a REST API, a SQL interface, a vendor SDK, an LTI launch. It holds the knowledge of the system's data model so that the agent does not have to.

In Miguel Amigot's phrase from a panel at Harvard, the connector lets an agent translate plain English into API calls against the systems where the data actually lives. That is the entire function, and it is enough.

## Why a protocol and not an integration

Before the Model Context Protocol, every AI tool integrated with every system in its own way. A vendor would build "our Salesforce integration" and "our Canvas integration," and they belonged to that vendor. An institution with three AI tools had three Salesforce integrations, each maintained separately, each encoding institutional knowledge in a form that could not be shared.

A protocol changes the ownership. A connector written to the protocol presents the same shape to every client. The institution writes, buys or adopts one connector per

system, runs it on its own infrastructure, and any agent platform that speaks the protocol can use it. The institution's investment in understanding its own systems becomes an asset that survives any single vendor.

The protocol has three other properties that matter here. It is open, with a published specification and reference implementations in several languages, so an institution's own engineers can write a connector for a system nobody else has. It is widely adopted: by 2025 the major model developers supported it, which is why a connector written for one platform works with another. And it is simple enough that a connector for a well-documented system is a few hundred lines of code, not a project.

## **The catalogue a CIO should expect**

A sovereign platform should arrive with connectors for the systems its market actually runs, and a way to add the rest.

For higher education: the student information system (Banner, PeopleSoft, Jenzabar, Colleague), the learning management system (Canvas, Blackboard, D2L, Moodle), the CRM for recruitment and advancement (Slate, Salesforce), the identity directory, the ticketing tool, the document stores.

For enterprises: the CRM, the ERP, the HR system, the support desk, the project tools, the collaboration suites, the data warehouse.

For healthcare: the electronic health record through its sanctioned interfaces, scheduling, the clinical document store, with the data classification rules from Chapter 6 applied at the broker.

For legal and financial services: matter and case management, document management, the billing system, the research databases, market and reference data.

Across all of them: the generic connectors for file systems, databases, web pages, calendars and email that most real tasks touch somewhere.

The number matters less than the pattern. A platform that ships a few dozen connectors and a clear way to write the next one, with the connectors running inside the institution's perimeter, is in a different category from one that ships a hundred integrations running in its own cloud.

## **Scoping to the person**

The property that turns a connector from a convenience into a governance tool is how it authenticates to the source system.

The wrong way, and the common way, is a single service account with broad read access. The agent can then see everything the service account can see, and the only thing

standing between a curious user and another person's record is the model's judgement about what to retrieve. That is not a control.

The right way is to call the source system as the person asking, or with a scope derived from their role. A student asking an advising agent about their own transcript triggers a call to the student information system under that student's identity, and the system returns that student's transcript and nothing else. An advisor asking about their caseload gets their caseload. The source system's own permission model does the work it already does for the system's native interface.

This requires that the identity flowing from the client, through the agent, to the connector is the real user's identity, which Part IV covers, and that something between the agent and the connector enforces it, which is the broker in the next chapter. A connector catalogue without those two pieces is a liability with good documentation.

## **Plain-English skills on top of connectors**

Connectors give an agent the ability to reach a system. They do not tell it when, in what order, or what to do with the result. That knowledge lives in skills: instructions, written in plain language, that describe how to accomplish a task using the tools available.

A skill for "prepare a student's registration hold summary" might say: look up the student, list their holds, for each hold find the office responsible, draft a summary in this format, flag anything older than thirty days. It names the connectors' tools and the steps. It is written by someone who knows the process, not by an engineer, and it can be read and corrected by anyone.

The sovereign platform I work on ships over a hundred and sixty such skills, each a readable file, covering platform operations as well as institutional tasks, and institutions add their own. The practical effect is that the people who understand the work can shape what the agents do without a development cycle, and the connectors underneath stay generic and reusable.

## **What to ask a vendor about their connectors**

Five questions sort the market quickly.

Do the connectors speak the Model Context Protocol, so they work with other platforms, or are they proprietary to yours? Where do they run: inside my perimeter, or in your cloud? Do they call my systems as the asking user, or as a service account? Can my engineers write one for a system you have never seen, with your documentation and no professional services engagement? And when a connector calls a system, where is the credential, and who can see the log?

The last question is the one that leads to the broker, and the next chapter answers it.

## CHAPTER 11

# The MCP Broker

*Credentials reach tools through the broker, never through the agent. Everything else about governance follows from that sentence.*

Connectors give agents reach. The broker gives the institution control over that reach. It is the single component in a sovereign data layer that most AI products do not have, and its absence is why so many deployments either lock agents out of real data or let them see too much. This chapter is about what a broker is, what it holds, and why it must be the institution's.

## The piece most vendors skip

In a typical agent product, the agent holds its own credentials. The connector to the CRM is configured with an API key; the key lives in the agent's environment; when the agent decides to call the CRM, it does so directly. This is simple, and it fails in every way that matters.

The credential is wherever the agent runs, which may be a cloud service, a user's browser, or a desk-side machine, and it is as safe as that environment. The agent calls the system with the credential's full authority, not the user's. There is no central place to say "this agent may read but not write," or "this role may not call this tool at all." And the log of what was called, by whom, on whose behalf, is scattered across connectors or does not exist.

A broker fixes all four problems by introducing one component between every agent and every tool.

## What the broker holds

The broker is a service, inside the institution's perimeter, that every tool call passes through. It holds three things.

**The credentials.** The API keys, service identities and OAuth grants that connectors need to reach systems of record live in the broker's secret store, not in any agent. An agent that wants to call a tool asks the broker; the broker, if the call is allowed, attaches the credential and forwards it. The agent never sees the key. A compromised agent, a

leaked prompt, a stolen laptop running the browser extension: none of them yields a credential, because none of them ever held one.

**The policy.** A table that says which roles may call which tools, with what scopes, on which agents. Read access to the student information system for advising roles. Write access to the ticketing system for support agents only. No access to the payroll connector for anyone except the HR agent, and then only to specific fields. The policy is data, editable by an administrator, versioned and auditable.

**The audit log.** Every call: timestamp, agent, user, role, tool, parameters, result size, decision. Written to a store that the broker itself cannot alter after the fact.

## Deny by default

The policy evaluation has one rule that every CIO should insist on: deny by default. A tool call that no policy explicitly allows is refused.

The alternative, allow by default with exceptions, is how most integration platforms work, and it means that every new connector, every new agent and every new role starts with maximum access until someone remembers to restrict it. Deny by default inverts the burden. A new connector exposes nothing until an administrator grants a role access to specific tools. A new agent can call nothing until it is placed in the permission tree. The institution's exposure grows only by deliberate acts, each of which is logged.

The practical cost is a little friction at setup, which the platform should make easy with templates. The practical benefit is that the answer to "what can this agent reach" is always "exactly what we said, and here is the table."

## Scoping every call to the person

Chapter 10 argued that connectors should call systems as the asking person. The broker is where that happens.

When a user asks an agent a question and the agent decides to call a tool, the request that reaches the broker carries the user's identity, verified through the identity layer from Part IV, and the agent's identity. The broker looks up the user's role, checks the policy for that role, agent and tool, and if the call is allowed, decides how to authenticate to the source system: as the user, by exchanging the user's token for one the source system accepts; or with a scoped service credential that the policy limits to what that role may see.

Either way, the source system's own permission model applies, and the broker's policy applies on top. A student cannot see another student's record because the student information system will not return it under their identity, and because the advising

agent's policy restricts the lookup tool to the caller's own ID. Two independent controls, neither of which depends on the model behaving.

## **The audit line**

The audit log is the artefact that turns a sovereign deployment from a security posture into evidence.

Because every tool call passes through the broker, the log is complete by construction. There is no path from an agent to a system of record that bypasses it. When a regulator, an auditor or a CISO asks what the AI has touched, the answer is a query over one table: which agents read patient records last month, under which users, how many calls, what was returned. When something goes wrong, the investigation starts with a filter, not with a search through connector logs that may or may not exist.

The log also feeds operations. Spend by tool and by agent. Which connectors are busy and which are dead. Which roles are hitting the policy wall, suggesting a missing grant or a misuse attempt. A broker with a good log is the control room for the data layer.

## **Why the broker belongs to the institution**

A vendor can run a broker in its own cloud, and some do. That recreates the problem the broker solved. The credentials to every institutional system would live in the vendor's secret store; the policy would be enforced on infrastructure the institution does not control; the audit log would be the vendor's to retain or lose.

The sovereign requirement is that the broker runs inside the institution's perimeter, on the same tier as everything else from Chapter 3: the institution's cloud account, its servers, or behind its air gap. The credentials never leave. The policy is the institution's. The log is the institution's. And when the vendor relationship ends, the broker, the connectors and the policy table keep running, because they were never the vendor's to take.

A CIO evaluating a platform can test this in one question: where is the secret store that holds the credentials your connectors use to reach my systems? If the answer is anywhere other than infrastructure the institution controls, the data layer is not sovereign, whatever the connectors can do. The next chapter adds the last piece of the layer, the ontology that gives agents a shared vocabulary for the institution without copying it.

## CHAPTER 12

# The Ontology over Your Systems

*An ontology that stores nothing is the only one you never have to migrate.*

Connectors reach systems. The broker governs the reach. One problem remains: each system calls things by its own names, and an agent that has to know that a "person" in the directory is a "student" in one system, a "constituent" in another and an "account" in a third is an agent that will get it wrong. The traditional answer is a data warehouse with a unified schema, which is the migration from Chapter 9 under a grander name. The sovereign answer is an ontology that describes the institution's objects and where they live, and stores nothing itself.

## The vocabulary problem

Institutions are federations of systems, each built by a different vendor for a different purpose, each with its own model of the world. The registrar's system thinks in terms of enrolments and sections. The advancement CRM thinks in donors and campaigns. The HR system thinks in positions and appointments. The same human being appears in all three under different identifiers, with overlapping but inconsistent attributes.

A person asking an agent a question does not think in any of those terms. They ask about "Maria," "the biology department," "last term's cohort." For the agent to answer, something has to map those everyday concepts to the specific systems, identifiers and fields where the facts are. Without that mapping, every skill has to encode it from scratch, and every agent reinvents the institution's data model badly.

## A pass-through, not a warehouse

An ontology, in the sense I mean here, is a description of the institution's concepts, the relationships among them, and for each concept, the system of record and the connector tool that answers questions about it. A Student has a Program, Enrolments, Holds and an Advisor. Student attributes come from the SIS through this tool; the advising relationship comes from the advising platform through that tool; the directory supplies contact details.

The crucial property is that the ontology holds the map and not the territory. It stores no student records, no enrolments, no holds. When an agent asks about Maria's holds, the ontology resolves "Maria" to an identifier, resolves "holds" to the SIS connector's hold-

lookup tool, and the broker makes the call under the asking user's identity. The answer comes from the SIS, now, with the SIS's permissions. Nothing was copied.

This is why I describe it as a pass-through. It is a layer of meaning over the systems, not a layer of data beside them. It can be versioned, reviewed and extended like any configuration. It is never stale, because it contains no facts that can go stale. And it never has to be migrated, because there is nothing in it to move.

## **How identity and authorization flow through it**

Because the ontology sits above the connectors and the broker, it inherits their guarantees rather than weakening them.

A query that arrives at the ontology carries the user's verified identity. The ontology's resolution of concepts to tools does not change that identity or add any authority to it. When the resolved call reaches the broker, the broker applies its deny-by-default policy for that user's role, agent and tool, exactly as if the agent had named the tool directly. And the source system applies its own permissions last.

What the ontology adds is consistency. Two agents asking about the same concept resolve to the same tool with the same scoping, because the mapping lives in one place. An administrator who changes which system is authoritative for a concept, say, moving advising relationships from a spreadsheet to a real platform, changes one mapping and every agent follows.

Some platforms also use the ontology to enforce authorization rules that no single source system can express, such as "an advisor may see enrolments only for students in their assigned caseload across both the SIS and the LMS." Those rules live beside the mapping, are evaluated at query time, and are logged like any other decision. They are policy over live data, not a filtered copy of it.

## **What belongs in the ontology and what does not**

The discipline that keeps an ontology from becoming a warehouse is a short list of what it may contain.

It may contain concept definitions, relationships, and the mapping from each concept to the system and tool that serves it. It may contain identity resolution rules: how a person in one system is matched to the same person in another. It may contain authorization rules that span systems. It may contain descriptive metadata that helps the agent choose: this field is the preferred name, that one is legal name; this date is in the local time zone.

It may not contain records. No cached results, no nightly snapshots, no "just this one table for performance." The moment it holds data, it has a freshness problem, a

permissions problem and an exit problem, and it has become the thing it was built to avoid.

Where performance genuinely requires a copy, a search index over documents for instance, that copy is a separate, explicitly governed component with its own classification and its own refresh policy, reached through a connector like any other system. The ontology points at it; it does not become it.

## **Building it incrementally**

Institutions sometimes hear "ontology" and imagine a two-year enterprise modelling exercise. The sovereign approach is the opposite. The ontology starts with the concepts the first agent needs, three or four, mapped to one or two connectors, and grows with each agent that follows. Because it stores nothing, there is no backfill, no migration and no cutover. Adding a concept is adding a definition and a mapping.

In practice the growth follows the connectors. Connect the SIS and the ontology learns about students, courses and enrolments. Connect the CRM and it learns about prospects and donors, and about the identity rules that link them to students and alumni. Connect the ticketing system and it learns about cases and the people who file them. Each step makes every existing agent a little more capable, because the vocabulary they share has grown.

Appendix E lays out a starting catalogue of concepts and connectors for the sectors this book addresses. The point of the catalogue is not completeness. It is to show how small the first step is, and how much of the institution an agent can reach, live and governed, once the data layer is built this way instead of by copying.

Part IV turns to the clients: how the agents built on this data layer reach people on the web, in the browser, on the desktop and on the phone, through one identity system that never hands an application a secret.

## CHAPTER 13

# One Backend, Every Surface

*A platform that only exists as a web app has decided where its users may work. That is not the platform's decision to make.*

The data layer described in Part III is invisible to the people it serves. What they see is a client: a chat window on the web, a side panel in the browser, an app on a laptop or a phone, a command in a terminal. This chapter is about why a sovereign platform has to show up on all of those surfaces from one backend, and what that requires of the engineering.

## Where the work actually happens

A CIO buying an AI platform is told to imagine users opening a web application, typing a question, and reading an answer. Some work happens that way. Most does not.

An advisor spends the day inside the student information system and the advising platform. A recruiter lives in the CRM and LinkedIn. A clinician works in the electronic health record. A lawyer works in a document editor and a matter management system. A finance analyst works in spreadsheets and the ERP. An engineer works in a terminal and an editor. The common property is that the work has a surface already, and the surface is not the AI vendor's web page.

An AI platform that requires people to leave their work, open a separate application, paste context into it and carry the answer back has a ceiling on its usefulness that no model improvement will lift. The platform has to meet people where they are: in the browser tab they have open, on the desktop they use, on the phone in their hand, in the terminal where the automation runs.

## Seven surfaces, one codebase

The platform I work on ships to seven surfaces, and the list is a reasonable specification for what a CIO should expect.

**The web.** The full application, in any browser, with nothing to install. The default for most people most of the time.

**The browser.** A Chrome side panel that puts the same agents on whatever page the person is on, reads the tab as context, and, when asked, acts on the page. Chapter 15 is about it.

**macOS and Windows.** Native desktop applications, signed and notarized, with the same interface and the additional abilities a desktop grants: local files, system integration, a runtime that can drive other applications under the user's supervision.

**iOS and Android.** Phone and tablet applications in the two stores, for the person who is not at a desk.

**Linux.** Built from source, because the people who run Linux desktops generally prefer it that way, and because the source is theirs to build.

**The terminal.** A headless surface: the platform's operations exposed as skills that a coding agent or a script can call, so an administrator can create agents, configure models, manage users and pull analytics with no interface at all. Chapter 19 returns to this.

What matters is not the count but the word "one." These are one backend, one identity system, one set of agents and one memory, presented seven ways. A conversation started in the side panel is waiting in the desktop app. A model switched by an administrator changes for every surface at once. A connector added to the broker is reachable from the phone.

## What a native shell adds

It is fair to ask why a desktop or mobile application is worth building when the web version works. Three reasons, in order of importance.

**Presence.** An application in the dock or on the home screen is used; a bookmark is forgotten. For an institution trying to make an AI platform the default place its people turn, the difference in adoption is large.

**Capability.** A desktop application can read local files, watch a folder, drive other applications on the person's behalf through the operating system's own automation interfaces, and run a local model when the hardware allows it. A browser tab can do none of that. For the agents in Part V, which act rather than merely answer, the desktop is where the acting happens.

**Control.** A native application can be built against the institution's own deployment, locked to its organization, and distributed through the institution's device management. An air-gapped institution builds its own, pointed at its own host. A web page cannot be handed to a device-management system.

The engineering that makes this affordable is a shell: a thin native wrapper around the same web application, so that the interface is written once and the native layer adds only what the platform cannot do in a browser. The sovereign platform ships that shell and the tooling to build and release it, so that an institution's team can produce signed desktop and mobile builds against their own deployment without becoming a mobile development shop.

## **Shared components, not shared screenshots**

The temptation, when building for many surfaces, is to build many applications that look alike. The sovereign requirement is stronger: they must be the same components.

The chat interface, the agent picker, the memory panel, the voice controls, the embed widget, and above all the sign-in flow, exist once as a library, and every surface mounts them. When the memory panel gains a feature, every surface gains it. When a security fix lands in the sign-in flow, it lands everywhere at once. And when an institution builds an application of its own, the subject of Chapter 16, it mounts the same components and inherits the same behaviour.

This is a property a CIO can check in a demo. Ask to see the same feature on the web and in the desktop application. If they differ in anything but the window around them, there are two codebases, and the institution will be living with their divergence for years.

## **The cost of the alternative**

An institution that adopts a web-only AI platform pays three costs that do not appear on the invoice.

It pays in adoption, because the platform is one more tab, and tabs lose to the tools people already have open.

It pays in reach, because an agent that cannot see the page the person is working on, or the file on their desk, has to be told everything, and most people will not tell it.

And it pays in governance, because when the web application turns out to be insufficient, departments buy or build their own clients against whatever API they can find, each with its own sign-in, its own credential and its own idea of what the user may see. The next chapter is about the identity layer that makes the alternative, many clients on one governed backend, safe to build.

## CHAPTER 14

# Identity Without Secrets in the App

*The application that never held a secret is the application that cannot leak one.*

Every application an institution builds or buys has to answer the same question: who is this, and what are they allowed to do? Answer it badly and the institution ends up with credentials scattered across code, configuration files and laptops. Answer it once, in a shared identity layer, and a new application can be built in a day by someone who never touches a secret. This chapter describes the design that makes the second outcome the default.

## Two kinds of credential

Start by separating two things that are usually confused.

A **session token** represents a signed-in person. It carries exactly that person's permissions in exactly their organization. It is short-lived, it expires, and it is refreshed by signing in again. If it leaks, the attacker is that one person for a short time.

An **organization credential** represents the institution itself. It has administrative authority over an organization's agents, users and data. It lives until someone deletes it. If it leaks, the attacker is the institution.

The entire design follows from where each of these may live. Session tokens live with the person, in the client they are using. Organization credentials live on a server the institution controls, and nowhere else. No client, web, browser extension, desktop or mobile, ever holds an organization credential. A platform that hands a client an organization key, even "just for the demo," has already failed.

## The hosted sign-in hand-off

The mechanism that delivers a session token to a client without the client ever seeing a password or a secret is a hand-off through a hosted sign-in service.

An application that needs a signed-in user does not show a password form of its own. It redirects the person to the platform's sign-in service, carrying one piece of information: where to return to, an origin the organization's administrator has placed on an allow-list. The sign-in service authenticates the person, by password, by the institution's single

sign-on, by Google or Apple, however the organization has configured it, and returns the person to the application with a bundle of session material: user-scoped tokens with expiry times, the person's profile, and the list of organizations they belong to.

The application stores that bundle locally, scoped to its own origin, and uses the tokens on every request. When they expire, the application sends the person back through the same hand-off. The application never saw a password. It never held anything with more authority than the person in front of it. And the sign-in flow, the one piece of security-sensitive code in the whole system, exists in one place, maintained by the platform, and shared by every client.

The practical consequence is the one the chapter title promises. A team building a new application, for a department, for a semester, for an experiment, mounts the platform's sign-in component and a session provider, lists their application's origin with the administrator, and has working single sign-on with roles and organization scoping in an afternoon. They wrote no authentication code and they hold no secret.

## **Why the organization's key never reaches the client**

Some applications need to act as the organization rather than as a person: a nightly job that reads every agent's analytics, a service that provisions users, an integration that another system calls. Those need the organization credential.

The rule is that such code runs on a server. The credential lives in the server's environment, is read by the server, and is never sent to a browser or a device. The platform's own client library enforces this by construction: the function that returns the organization key returns an empty value when it runs in a browser, so a developer who tries to use it in client code gets nothing rather than a leak.

A server route therefore acts in one of two ways. As the person, by forwarding the session token the browser sent and letting the platform verify it. Or as the organization, with the credential from its own environment. Each path is explicit; neither leaks into the other.

## **Organizations, switching and visiting**

Real institutions are not one organization. A university system has campuses; a company has subsidiaries; a consortium has members. A person may belong to several and may visit others.

The identity layer carries, with each session, the list of organizations the person belongs to and which one is current. An application that needs to operate in a different organization re-authenticates for it, receiving a fresh token pair scoped to that organization. A person opening an agent that belongs to an organization they are not a

member of is a visitor: they may reach it if its visibility allows, and organization-scoped calls that the agent would normally make are refused. All of this is handled by the shared session provider; an application built on it inherits the behaviour.

This matters to a CIO for a specific reason. Multi-organization support is where home-grown identity goes wrong, and it is the area where a leaked token does the most damage. Having it solved once, in a component every client uses, removes an entire category of incident.

## **Native shells and the system browser**

Mobile and desktop shells complicate the hand-off, because single sign-on providers refuse to run inside an embedded web view. The platform's answer is to open the sign-in service in the system browser and return to the application through a registered scheme, the same pattern the operating systems themselves recommend. A desktop build can also be locked to one organization at build time, so that an institution's own build of the application never shows anyone another organization's sign-in.

The browser extension from the next chapter uses the browser's own identity API for the same hand-off, which is why it needs no password form and holds no secret either.

## **The threat model, honestly**

Session tokens in a client are exposed to one thing: code running on that client's origin. A cross-site scripting flaw in an application built on the platform would let an attacker read that application's tokens and act as the signed-in person until they expire. This is the standard exposure of every web application that keeps a session, and the mitigations are standard: a strict content security policy, short token lifetimes, no third-party scripts on sensitive origins, and the one rule this chapter has repeated, that nothing with more authority than the person ever reaches the client.

What the design removes is the catastrophic case. There is no organization key in any client to steal. There is no password form in any application to phish. There is one sign-in flow to audit instead of fifty. And the application a department built last week is exactly as secure as the platform itself, because it reused the platform's identity rather than inventing its own.

The next chapter takes the hardest client, the browser extension, and shows how these guarantees hold when the agent is reading and acting on whatever page a person happens to have open.

## CHAPTER 15

# The Agent in the Browser

*Most AI browser extensions send your screen to someone else's server. Ours sends your question to your own platform.*

The browser is where most knowledge work now happens, and it is where the AI industry has placed its most invasive products: extensions that read every page, send it to a vendor's server, and bill per seat for the privilege. A sovereign platform has to be in the browser too, because that is where the people are. This chapter describes how it can be there without any of the invasion, using the platform's own Chrome extension as the worked example.

## What the side panel is

The extension opens a side panel next to whatever page is open. The panel is the platform's chat and nothing else: the same agents the person uses on the web and the desktop, with the same memory, the same model and the same connectors. It has no settings of its own and no second sign-in; it uses the browser's identity API to complete the hand-off from Chapter 14, and from then on every request carries the person's own session token.

The extension itself is small. The published version is a few hundred kilobytes, built from source that lives in the platform's open repository, and published to the Chrome Web Store automatically whenever its version number changes. An institution can install it from the store, pointed at the platform's hosted service, or build it against its own deployment with one configuration value, which is what an on-premise or air-gapped institution does.

## Reading the page

The first thing the panel does is feed the text of the active tab to the agent as context. A person on a vendor's documentation page can ask what a paragraph means. A person on an internal portal can ask the agent to summarize a case. The page content goes to the person's own platform, through their own session, to the model their administrator chose, and nowhere else.

The second thing is more capable. When the person turns on the agent's browsing mode, the extension's background process reads the page as a structured list: every interactive

element, numbered, across every frame and every open component, with what is on screen listed first. For models that can read images, it also takes a picture of the visible tab with the numbers drawn on, so the model can see what the list describes. That representation is what lets the agent act.

## **Acting on the page**

Given a goal, the agent runs a loop: read the page, decide one action, perform it, read again, until the goal is met. The actions are the ones a person has: click, double-click, type, select, scroll, open a URL, go back, wait. They are delivered as real input events through the browser's debugging interface, so pages that ignore scripted input, menus that open on a press, editors that need a real keystroke, behave as they would for a human. The browser shows its standard notice that the tab is being driven for exactly as long as the run lasts.

A run has limits. It stops after a fixed number of steps or when its token budget is spent, says which, and offers to continue. A stop control ends it at any time. Closing the panel ends it. Every action is narrated in the chat as it happens, so the person watches the agent work rather than discovering afterwards what it did.

The demonstration our team published shows the whole pattern in one scene: a colleague opens LinkedIn, opens the panel, and the LinkedIn agent she built on the platform drafts her post and publishes it. The agent knew her voice because its memory is hers; it knew the page because the panel read it; it could post because the browsing mode gave it real clicks; and nothing about the exchange touched any server but her own organization's.

## **The refusals and the approvals**

An agent that can act on any page needs rules that do not depend on the model's judgement, and the extension has two kinds.

Hard refusals. The agent never types into a password field or a payment field, in any mode, and never reads such a field's contents back. This is enforced in the page, by the field's own type, not by asking the model to be careful. Only ordinary web pages can be driven at all; browser-internal pages are out of reach by construction.

Approval modes. An administrator or the person chooses between automatic operation, where the agent acts without asking, and an ask-first mode that raises an approve-or-deny card before any submit, send, payment or delete. The card is the same one the desktop application uses for the same purpose, so the behaviour is consistent across surfaces.

These are the properties a CISO should look for in any browser agent. The question is not whether the model is well-behaved. It is what the system refuses to do regardless of the model, and whether a human can be placed in the loop at the moments that matter.

## **The trust boundary**

The reason this chapter belongs in a book on sovereignty is the boundary around the data.

Model calls go from the extension to the person's own platform, through the platform's standard model interface, with the person's own session. The platform applies the routing policy from Chapter 6, including data classification, so a page from a restricted system can be confined to a local model. The page content is never sent to a third party by the extension; the only server it talks to is the institution's platform, and the manifest's security policy names those hosts and no others.

Compare that with the common design, where the extension is the vendor's product, the page goes to the vendor's server, the model is the vendor's, and the person is a seat on an invoice. The two designs look identical from the outside. They differ in every property a CIO is responsible for.

## **What the demo proves**

A browser agent is the most demonstrable thing a sovereign platform ships and the easiest to explain: the agent reads the page you are on and acts on it, with model calls on your own session and nothing else leaving the browser. That sentence is the product's whole argument.

It also proves the client-layer claims from the last two chapters in a way a slide cannot. The same agent, memory and model as the web application: one backend. No password form, no secret in the extension: the identity hand-off works in the hardest client. One build against the hosted service, another against an internal host: the same code on every deployment tier. A CIO who watches the panel work on an internal page, under their own deployment, has seen the architecture, not a description of it.

## CHAPTER 16

# Shared Components as Policy

*A governance rule that lives in a document is advice. One that lives in the component every app mounts is behaviour.*

Part IV has argued that one backend should serve every surface, that identity should be a shared service, and that the browser agent shows both working. This chapter closes the part with the organizational consequence. When the pieces every application needs are shared libraries rather than per-application code, governance stops being a review process and becomes a property of the platform. That is what lets an institution build applications quickly without losing control of them.

## Where governance usually fails

Institutions do not lack AI governance policies. They lack a way to make the policies true across a hundred applications built by forty teams. The policy says that AI applications must use institutional single sign-on; a department's prototype has a hard-coded user list. The policy says that agents must not retain personal data without consent; a vendor's tool keeps every conversation. The policy says that actions with financial effect require human approval; a script someone wrote last month does not know what a financial effect is.

Review boards try to catch these. They catch the ones that ask for review. The applications that matter most, the ones built quickly because somebody needed them, never ask.

The structural answer is to put the policy where the applications already have to go: into the components they mount to get anything done.

## The components that carry policy

A sovereign platform's client library is a set of components that every surface, and every application an institution builds, uses because building without them is harder than building with them. Each one carries a piece of governance.

**The sign-in component** carries identity policy. An application that mounts it uses institutional single sign-on, receives roles and organization scope, and holds no secret, because that is the only way the component works. Chapter 14 described it.

**The chat component** carries disclosure and safety policy. It shows the person which agent they are talking to and that it is an AI. It applies the agent's configured guardrails, content rules, PII handling, the prompts the administrator wrote, because those live in the agent's configuration on the platform, not in the application that mounts the component.

**The memory component** carries data policy. What the agent remembers about a person, how it is categorized, who may see it, and the person's own switches to turn capture and recall off, are the platform's rules, enforced in the component and the backend, visible and editable by the person and the administrator.

**The approval card** carries action policy. When an agent is about to submit, send, pay or delete, the same card appears in the browser extension, the desktop application and any application that mounts the chat, because it is one component. An institution that decides such actions always require approval sets that once.

**The embed widget** carries exposure policy. When an agent is placed on a public page or inside another system, who may reach it, whether it allows anonymous use, and what it may see, come from its configuration, not from the page that embedded it.

An application built from these components cannot opt out of the policies they carry. That is not a restriction on the developer; it is the reason the developer can ship without a six-week review.

## Building an application in an afternoon

Here is what building an internal application on a sovereign platform looks like, concretely, when the components do the governance.

A department wants a tool for a specific process: say, preparing a summary for each student with a registration hold. A developer, or increasingly a coding agent working with a developer, starts from the platform's application template. The template already includes the sign-in component, the session provider and the organization handling. The developer registers the application's origin with the administrator so the sign-in hand-off will return to it.

The developer mounts the chat component, pointed at an agent the department has configured on the platform with the right connectors, the right skill and the right memory settings. The agent reaches the student information system through the broker, under the asking advisor's identity, as Part III described. The developer adds whatever interface the process needs around the chat: a list of students, a button, a summary view.

That is the application. It has institutional single sign-on, role scoping, audited data access, the agent's guardrails, memory policy and approval behaviour, and it holds no credential. The developer wrote the parts that are specific to the process and inherited

everything else. Deployed through the platform's hosting or the institution's own, it is live the same day.

## **Skills that make a coding agent a platform developer**

The afternoon above is shorter still when the developer's coding agent already knows the platform. The sovereign platform I work on ships its developer knowledge as skills: more than a hundred and sixty readable files, one per task, that a coding agent loads and follows. Set up sign-in. Mount the chat for a specific agent. Add the memory tab. Deploy to hosting. Release to the app stores. Create an agent through the API. Configure its model. Wire a connector.

A developer who adds those skills to their coding agent gets an assistant that builds correctly on the platform the first time, mounting the shared components rather than reinventing them, because the skills say so. The institution's governance rides along, not because the developer studied the policy, but because the skill the agent followed used the component that enforces it.

## **What the CIO gets**

The result, from the CIO's chair, is a change in what the application estate is.

New applications appear faster, because the hard parts are inherited. They are safer than the ones that used to appear slowly, because they cannot skip the identity layer or the guardrails. They are consistent across surfaces, because the components are. And they are disposable in the sense Chapter 2 described: when a process changes, the application around it can be rebuilt in a day, and the agent, the connectors and the identity underneath it do not change at all.

Governance, in that estate, is the work of configuring the platform well: the agents, their guardrails, the broker's policy table, the memory rules, the approval settings. It is a small number of decisions, made by the people responsible for them, and enforced everywhere by construction. Part V turns to the agents themselves: what the harness that runs them must be able to do, and what a headless application looks like when the harness is the product.

## CHAPTER 17

# What a Harness Is

*The model decides. The harness decides what the model is allowed to decide.*

An agent is a model running inside a loop, and the loop is called a harness. The model supplies judgement; the harness supplies everything else: the tools the model may call, the memory it may consult, the limits on what it may do, the record of what it did. When people say an AI product "has agents," the quality of those agents is mostly the quality of the harness. This chapter defines the harness and lists the attributes a CIO should demand of one.

## The loop

Strip away the vocabulary and every agent does the same thing. It reads its situation: the conversation so far, the page in front of it, the results of its last action. It decides one next step: answer, or call a tool. If it calls a tool, the harness executes the call and feeds the result back. The agent reads again. This repeats until it answers or runs out of allowance.

The browser extension in Chapter 15 runs exactly this loop against a web page. A research agent runs it against a document store. An operations agent runs it against the systems of record through the broker. The loop is identical; what differs is the tools and the limits. That is why the harness, not the model, is where an institution's control lives.

## Nine attributes of a flexible harness

A harness that an institution can trust across many agents, surfaces and years has these properties. I list them because vendors rarely do.

**Model-agnostic.** The harness calls the model through the standard interface from Chapter 5, so the model is a setting. A harness welded to one model inherits that model's lock-in.

**Tool protocol, not hard-wired tools.** Tools arrive through the Model Context Protocol, so any connector from Part III is available to any agent, and a new connector needs no change to the harness.

**Memory with scopes.** The agent can remember, and what it remembers is organized by whose memory it is and who may see it. The next section expands this.

**A sandbox with a network boundary.** When the agent runs code, it runs in an isolated environment on infrastructure the institution controls, with explicit rules about what the code may reach.

**Approval modes.** The harness can be set to act automatically or to pause for a human before consequential actions, per agent, per action class.

**Budgets.** A run has a maximum number of steps and a maximum token spend. When it hits either, it stops and says so.

**A stop button.** A human can end any run at any time, and the harness honours it immediately.

**An audit trail.** Every step, every tool call and its parameters, every approval, written to a log the institution owns.

**The same runtime everywhere.** The harness behaves identically whether it is driving a browser tab, running on a desktop, or executing on a server, because it is one implementation with different drivers underneath.

An agent product missing more than one or two of these is a chat interface with function calling, and the institution should price it as such.

## Memory in three scopes

Memory is what makes an agent feel like a colleague rather than a search box, and it is the attribute most likely to become a governance problem if it is designed casually. A sovereign harness keeps memory in three scopes, each with its own owner and rules.

**Global memory** belongs to a person and follows them across every agent in the organization: their role, their preferences, the context they have shared. It is captured from conversations when the person allows it, and the person can read, edit and delete it.

**Agent memory** belongs to a person and one agent: what this agent has learned about this person, filed by category, with an extraction prompt per category that an administrator can tune. Same rights: the person sees it, the administrator can manage it on their behalf.

**Shared memory** belongs to the agent and to no person: standing facts, rules and background that every conversation with that agent should start from. Our refund window is thirty days. Always cite the policy handbook. Shared memory is written by hand, by someone accountable for it, and is never extracted from a conversation,

because anything placed there reaches every person who talks to the agent. The harness enforces that by having no path to write it automatically.

Three gates control whether any of this reaches a conversation, and they cascade: the organization's switch, the agent's switch, and the person's own. A person who turns memory off for themselves is off, whatever the organization decided. All three stores are rows in the institution's own database, inside its perimeter, behind its access controls, exportable like everything else.

## **The sandbox and the network boundary**

Agents that only answer can be governed by the broker. Agents that act, by running code, transforming files, calling systems in sequence, need a place to act that cannot become a liability.

The sandbox is that place: an isolated execution environment, started for a run and discarded afterward, on infrastructure the institution controls. Inside it the agent can write and run a script against data too large for any context window, process a batch of documents, or orchestrate a multi-step task. The boundary around it is a network policy: by default nothing, then explicit allowances for the systems the task needs, through the broker, under the user's identity. The code the agent wrote cannot reach the internet, the institution's other systems, or the harness's own credentials unless the policy says so.

Guardrail frameworks of the kind NVIDIA ships with its agent tooling add a layer inside the loop: programmable rails that constrain the model's inputs and outputs, injection defences, PII redaction. Those are valuable. They complement the sandbox; they do not replace it, because a rail is advice to the model and a network boundary is a fact about the machine.

## **The same harness on every surface**

The attribute that ties the client layer to the agent layer is that the harness is one implementation. The browser extension drives a tab with it. The desktop application drives local applications with it. The server runs scheduled tasks with it. An agent configured once, with its memory, tools, approvals and budgets, behaves the same way wherever the person meets it, and an administrator reasons about one thing rather than three.

This is also what makes the demonstrations honest. When a CIO watches the browser agent refuse a password field, that refusal is the harness's, and it holds on the desktop and the server too.

## **What to demand**

In an evaluation, ask to see the harness, not the model. Ask to see the loop's limits set and hit. Ask to see memory in its three scopes, and a person turning theirs off. Ask to see the broker refuse a tool the agent's role does not have. Ask to see the sandbox's network policy and what happens when code inside it tries to reach somewhere it should not. Ask for the audit log of the demonstration you just watched.

A vendor whose harness has the nine attributes can show you all of this in an hour. A vendor whose product is a chat window will show you the model being clever. The difference is the whole of Part V, and the next chapter puts it in the frame most buyers will recognize.

## CHAPTER 18

# The Five Layers of a Production Agent

*What ships is layer one. What you are paying for is the other four.*

Chapter 17 described the harness from the inside. This chapter describes a production agent from the buyer's side, as five layers, because that framing makes it obvious what most products on the market are missing and what a CIO is actually paying for.

## The five layers

Each layer does work the layer above it cannot.

**Interface.** How a person meets the agent: a chat window, a side panel, an embedded widget, an API, a place inside an application they already use. This is the visible layer, and it is the one that demos.

**Orchestration.** Multi-step planning: breaking a request into steps, calling tools in sequence, handling a step that fails, deciding when the task is done. Without it, an agent is a chat interface that can call one function.

**Knowledge.** Live, role-scoped access to the systems of record, read at the moment of the question rather than uploaded in advance. Part III of this book.

**Memory.** Context that persists across sessions: what this person is working on, what was decided before, what the institution knows. Chapter 17's three scopes.

**Governance.** Role-based access, audit trails, spend limits and data-handling rules, enforced where the model cannot route around them. The broker, the sandbox boundary, the approval modes, the routing policy.

A product with all five is a platform. A product with the first is a demo that costs per seat.

The layers also explain a pattern CIOs notice and rarely name: the agent that impressed everyone in the pilot and disappointed everyone in production. The pilot exercised the interface. Production exercised orchestration against real failures, knowledge against real permissions, memory across real weeks, and governance against real people trying

to get around it. The pilot could not have revealed what was missing, because the missing layers are invisible until the work is real.

## **Why most products ship only the first**

The economics explain it. The interface is cheap to build and easy to sell; a polished chat window with a strong model behind it impresses in a thirty-minute meeting. Orchestration is harder and shows its value only on real tasks. Knowledge requires connectors into the customer's systems, which means engineering per customer and a broker nobody wants to build. Memory requires a data model, storage the customer trusts, and privacy controls. Governance requires all of the above plus the discipline to enforce rules in infrastructure rather than in prompts.

A vendor optimizing for the first meeting ships layer one and describes the other four in a roadmap. A vendor optimizing for the fifth year ships all five and has a worse demo. A CIO who knows the layers can tell which they are talking to in the first ten minutes.

## **Orchestration and memory are not a longer prompt**

Two of the layers are commonly faked, and it is worth knowing how.

Orchestration is faked with a long system prompt that tells the model to "break the task into steps and use the tools available." A model given that instruction will produce something that looks like a plan. It will not reliably recover from a failed tool call, will not know when it has finished, and will not stop when it should. Real orchestration lives in the harness: a loop with state, budgets, retries and a definition of done that the model participates in but does not own.

Memory is faked by stuffing prior conversations into the context window. That works for one session and collapses after a week: the context fills, the cost climbs, and nothing is organized by whose memory it is or who may see it. Real memory is a store with scopes, extraction rules, retrieval by relevance, and the switches that let a person and an administrator govern it.

The tell, in both cases, is to ask where the capability lives. If the answer is "in the prompt," it is layer one wearing a costume.

## **Governance belongs in infrastructure**

The fifth layer is the one this book cares about most, and the one where the difference between prompt and infrastructure is a compliance finding.

A rule enforced in a prompt, "do not reveal other students' records," "always ask before sending," is a request to the model. Models comply with requests most of the time, and

most of the time is not a standard any auditor accepts. A rule enforced in infrastructure holds regardless of what the model decides: the broker refuses the call the role does not have; the sandbox blocks the network route; the harness pauses for approval before the send because the action class requires it; the router confines restricted data to the local model.

The practical test is to try to break the rule from the chat window. Ask the agent, in a dozen creative ways, to do the thing the policy forbids. If the policy lives in a prompt, one of the dozen will work. If it lives in infrastructure, none will, and the audit log will show the twelve attempts.

## The questions to ask about each layer

For a CIO in an evaluation, five questions, one per layer, separate platforms from demos.

**Interface.** On which surfaces does the agent appear, from one backend? Web only is a warning.

**Orchestration.** Show me a task that takes six tool calls, one of which fails. What happens, and where is the logic that handles it?

**Knowledge.** Which of my systems can the agent read today, through what, running where, as whom? Where are the credentials?

**Memory.** Show me what the agent remembers about a person, in which scopes, and show me that person turning it off. Where is the store?

**Governance.** Show me the policy table, the audit log of this demonstration, and an attempt to violate a rule from the chat window.

A platform built the way this book describes answers all five with a screen rather than a slide. Our own answers are, in brief: seven surfaces from one backend; a harness with budgets, retries and approvals; connectors on the Model Context Protocol through an institution-run broker; three memory scopes in the institution's database; and governance in the broker, the router, the sandbox and the approval card. The point of the chapter, though, is not our answers. It is that a CIO should insist on answers at every layer, because the layer that ships is never the one that costs the most to be without.

## CHAPTER 19

# Headless Applications

*The head is optional. The body, the agent, the skills and the API, is permanent.*

Chapter 2 ended with a conclusion: when applications become cheap to generate, the durable assets are the data access, the identity and the agents, and the application becomes headless. This chapter is about what that looks like in practice, now that the platform underneath has been described.

## The post-SaaSpocalypse application

A headless application is an agent, a set of skills and an API, with a user interface only where a human genuinely needs one. The agent carries the capability. The skills tell it how to do the institution's specific work. The API lets other systems, scripts and coding agents invoke it. The interface, when there is one, is one of the shared components from Part IV mounted wherever the person already is: the side panel, a chat widget on an internal page, the desktop application, a message in a collaboration tool.

Consider the departmental tools the SaaSpocalypse argument says are at risk: the forms-over-data products that each cost a subscription. A request-tracking tool becomes an agent that reads the ticketing connector, a skill that encodes the department's triage rules, and a chat widget on the intranet page where requests used to be filed. A reporting tool becomes an agent with the warehouse connector and a skill per standard report, invoked by a scheduled task that posts the result where the readers are. A contract intake tool becomes an agent with the document connector, a skill that knows which clauses matter, and the browser side panel open on the document itself.

None of these has a dashboard. None needs one. The person asks, the agent acts through the broker under the person's identity, and the audit log records it.

## Managing a platform from a terminal

The clearest demonstration of headlessness is the platform managing itself. A sovereign platform exposes every administrative operation, creating and configuring agents, choosing models, writing prompts and safety rules, adding connectors, managing users, roles and tokens, reading analytics, as a documented API, and ships each operation as a skill a coding agent or a script can run.

An administrator with a coding agent and those skills can stand up a new agent, connect it to a system, set its model and its guardrails, invite its users and pull its first week of analytics without opening the web application once. The platform I work on ships more than a hundred and sixty such skills; an administrator adds them to their coding agent with one command and authenticates with an organization credential that, per Chapter 14, lives on their machine's server side and nowhere in a client.

This is not a convenience for engineers. It is the mechanism by which the institution's platform operations become automatable, reviewable and reproducible. A configuration applied by a skill can be applied again, to a second organization, to a test environment, to a disaster-recovery deployment, exactly. A configuration clicked through a web interface cannot.

## **Skills instead of screens**

The unit of institutional knowledge in a headless estate is the skill: a plain-language document that tells an agent how to accomplish a task with the tools it has. Chapter 10 introduced skills on top of connectors. Here they replace the application.

A screen encodes a process in forms, buttons and validation rules that a developer built and only a developer can change. A skill encodes the same process in prose that the person who owns the process can read and correct. "To prepare the weekly enrolment report: query enrolments by program for the current term through the SIS connector, compare to the same week last year, flag any program down more than ten percent, format as the standard table, post to the enrolment channel, and file the result in the reports folder." That is a skill. It is also the specification, the documentation and the audit reference for the task, in one readable file.

Because skills are files, they are versioned, reviewed and shared like code, without being code. A department's skills are its institutional memory of how it works. They outlast the person who wrote them, the application that used to do the work, and the vendor whose platform runs them, since they are written against the Model Context Protocol tools and the platform's documented API rather than against anyone's proprietary interface.

## **Where a head is still needed**

Headless does not mean faceless. Some work needs a rich interface: reviewing a complex document, comparing options visually, entering structured data at volume, anything a person does for hours. For those tasks the shared components provide the interface, and an application built from them, as Chapter 16 described, takes a day rather than a quarter.

The distinction is where the logic lives. In a headless estate the logic is in the agent and its skills; the interface, when it exists, is a view onto them. The same agent that answers

in the side panel powers the rich application, so the application can be rebuilt, moved to a new surface or retired without losing anything the institution knows how to do.

## **What it does to the budget and the org chart**

Two consequences for a CIO.

The software budget shifts from subscriptions to usage and people. The long tail of per-seat departmental tools shrinks as agents and skills replace them. What grows is model usage, priced per token and capped by the institution, and the time of the people who write skills and connectors. The second is the better investment: it compounds, it is portable, and it is owned.

The organization shifts too. The people who understand the institution's processes become the authors of its automation, because skills are written in their language. The engineering team's work moves to connectors, the broker, the identity layer and the harness, the durable infrastructure, and away from the forms that used to consume it. The platform team runs the model strategy, the policy table and the audit, as an operating function.

That is the sovereign operating model, and Part VI closes the book by describing how an institution gets there from a first pilot.

## CHAPTER 20

# The Sovereign Operating Model

*Bring us a hard problem. Keep the code.*

The previous nineteen chapters described an architecture. This one describes how an institution adopts it: the first pilot, the operating function that grows out of it, and the posture that keeps the institution in control for a decade. It is the chapter for the CIO who agrees with everything so far and has to decide what to do on Monday.

## Start with a hard, multi-system problem

Most AI pilots are designed to succeed, which is why they teach nothing. Fifty volunteers, a chat window, a general-purpose model, no connection to anything the institution actually runs. The pilot succeeds, the report is written, and the institution is no closer to knowing whether the platform can do real work.

A sovereign pilot inverts this. Choose a problem that is hard because it spans systems: a question whose answer requires the student information system and the advising platform and the ticketing tool; a process that today involves three departments passing spreadsheets. Connect those systems through the broker. Build the agent and its skills against live data. Put it in front of the people who do the work, on the surfaces where they do it.

The pilot that succeeds at a hard, multi-system problem has proven the data layer, the identity layer, the harness and the operating process at once. Syracuse University's first agent did this: it began with the materials that were easy to reach and connected more systems phase by phase, useful from the first week and more useful every week after. The pilot that merely proves the model can chat has proven what everyone already knew.

## Engineers embedded, code owned

The sovereign model of professional services is different from the one institutions are used to, and the difference is contractual as well as practical.

The vendor's engineers work inside the institution's team on the pilot problem: they build the connectors, shape the ontology, write the first skills and tune the harness alongside the institution's own people. That is forward-deployed engineering, and it is

how the institution's staff learn the platform by building on it rather than by reading about it.

The contractual difference is that everything built inside the pilot belongs to the institution: the connectors, the skills, the configuration and the agent. There is no separate line for the intellectual property, no clause that keeps the integration work on the vendor's side of the fence. When the pilot ends, the institution holds a working deployment it can run, extend and, if it chooses, run without the vendor. A pilot priced to leave the institution with nothing it can keep is a sales exercise, whatever it is called.

## **Evaluations before users**

Chapter 5 described evaluation sets as the mechanism for switching models safely. In the operating model they do more: they are the gate between building an agent and giving it to people.

Every agent carries a set of representative prompts with the properties a good answer must have. Before the agent reaches users, it is run against that set on its intended model, and the results are reviewed by the people who own the process. When the model changes, the set runs again. When a skill is edited, the set runs again. The results live in the platform where an administrator can see them, not in a spreadsheet on someone's laptop.

An institution that does this has an answer to the question every board eventually asks, "how do you know it works?", that is better than "people seem to like it." It has a test, a record of the test, and a record of every change since.

## **A budget cap you set**

The financial posture of a sovereign deployment is a cap, not a seat count. Model usage is paid per token, to the providers directly or through the platform's pooled credits, and the institution sets the ceiling. When the ceiling is reached, the platform stops spending; it does not invoice. Open-weight models on the institution's own hardware have no per-token cost at all, so the volume workloads routed there cost electricity.

For planning, the arithmetic from Chapter 1 holds. A campus of fourteen thousand that would pay \$3.36 million a year per seat spends a small fraction of that on tokens for the same work, and the platform fee that replaces the per-seat contract does not scale with headcount. The budget line becomes predictable in the way a utility bill is predictable: it follows use, it has a cap, and the institution can see where every dollar went, by agent and by model, in the spend view.

## **The exit you never use**

Chapter 4 argued that the only real exit clause is an architecture that makes the exit a fact. The operating model keeps that fact true.

Once a quarter, verify the four questions from Chapter 3. Can your engineers build and deploy the platform from source? Is every connector and skill in your own repositories? Can you switch the model on a production agent today? Does the air-gapped build still start? None of these takes more than a day. An institution that can answer yes to all four every quarter is never negotiating from weakness, and in my experience never has to use the exit it has kept open. The leverage is the point.

## **The ten-year view**

A CIO buying a sovereign AI platform is making a decision whose consequences will outlast every model named in this book, most of the vendors in the market, and probably their own tenure. The question to ask is not which product is best this year. It is which architecture the institution will be glad to be standing on in ten, when the models are commodities nobody remembers the names of, the applications have been regenerated a dozen times, and the data has never left.

The architecture this book describes is built to be that: code the institution holds, data that stayed in place, models that are settings, one identity across every surface, and agents in a harness the institution can read. Everything else will change. The people I have worked with at ibl.ai are a family-owned company in New York, and we have built the platform this way because we intend to be here for the long run too, as a partner the institution could leave and chooses not to.

Bring us a hard problem. Keep the code. That is the whole operating model, and it fits on a page.

# Framework: the five properties of a sovereign AI stack

A checklist a CIO can verify, not a slogan. Each chapter of this book maps back to one of these five properties; Appendix A turns them into a maturity model and Appendix B into a scored assessment.

1. **You own the code.** The complete source of the platform, backend and infrastructure included, under a perpetual license. You can read it, audit it, modify it, and run it after the vendor is gone.
2. **You own the data, in place.** Agents reach systems of record where they live, through connectors and a broker you control. Nothing is copied into a vendor's index, and nothing leaves your perimeter that you did not route out deliberately.
3. **You choose the model, and can change it.** Every model is a setting, not a dependency. Commercial APIs, open-weight models on your own GPUs, or both, routed by cost, latency, capability and data classification, switchable without touching an integration.
4. **Your clients share one identity.** Web, browser extension, desktop, mobile and terminal are one backend and one sign-in. User-scoped tokens live with the user; the organization's credential lives only on the server. New applications are built without anyone handling a secret.
5. **Your agents run inside a harness you can inspect.** The loop that reads, decides and acts is yours to read, with memory scopes, approval modes, step and token budgets, a sandbox with a network boundary, a stop button, and an audit line for every action.

# Sovereignty Maturity Model

A five-level model for placing an institution's AI estate on the path from rented to owned. Each level is defined by what the institution can verify, not by what it has bought. Use it with Appendix B, which scores an institution against the questions here.

## The five levels

| Level | Name              | Defining property                                         | Typical state                                                                                                                                                                                                |
|-------|-------------------|-----------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1     | <b>Rented</b>     | The vendor runs everything; the institution has accounts  | Per-seat SaaS assistants; data in the vendor's cloud; one model; no source; no connectors the institution owns                                                                                               |
| 2     | <b>Configured</b> | The institution controls settings but not the system      | Hosted platform with the institution's SSO and some connectors; model fixed; data copied to the vendor; export on request                                                                                    |
| 3     | <b>Deployed</b>   | The institution runs the platform; the vendor supports it | Platform in the institution's cloud account from source under license; models routable; connectors through an institution-run broker; data read in place                                                     |
| 4     | <b>Owned</b>      | The institution could run it without the vendor           | Source built and deployed by the institution's own engineers; open-weight models on institution hardware for restricted data; connectors and skills in institution repositories; quarterly exit verification |
| 5     | <b>Sovereign</b>  | Every property holds with the network cable unplugged     | Air-gapped build proven; signed update bundles; all five properties of the framework verified, including local models, local embeddings and local identity                                                   |

## What distinguishes each level

**Level 1 to 2.** The institution stops using personal accounts and consumer tools and adopts a platform with institutional identity and administration. Nothing structural has changed: the vendor still runs the system and holds the data, and the price is still per seat. Most institutions in 2026 are here, and many believe they are further along because the contract has a data-residency clause.

**Level 2 to 3.** The structural step. The platform is deployed into infrastructure the institution controls, from source the institution holds under license. The model becomes a setting. Connectors read systems of record in place through a broker the institution runs. This is the level at which the two problems from Chapter 1 are solved, and the level most of this book describes.

**Level 3 to 4.** The institution's own engineers can build and deploy the platform without the vendor's help, and have done so at least once. Restricted workloads run on open-weight models on the institution's own GPUs. Every connector, skill and agent configuration lives in repositories the institution owns. The four exit questions from Chapter 3 are verified on a schedule. The vendor relationship is now a choice renewed on merit.

**Level 4 to 5.** The institution has proven the air-gapped build: the platform starts and serves with no outbound network, with local models, local embeddings and local identity, and updates arrive as signed bundles. Few institutions need to operate at Level 5. Every institution at Level 4 benefits from being able to demonstrate it, because the demonstration proves there are no hidden dependencies.

## Common misplacements

Three patterns recur when institutions self-assess.

**A sovereign region is not Level 3.** Running a vendor's hosted product in a hyperscaler's in-country region keeps the disks local. It does not give the institution the source, the model choice, or custody of the data path. That is Level 2 with a map.

**Open source is not Level 4.** A platform whose client is open source but whose backend is the vendor's hosted service cannot be run by the institution. Level 4 requires that the whole system, backend and infrastructure definitions included, builds and runs on the institution's infrastructure.

**A pilot is not a level.** An institution running a Level 3 pilot for one department while the rest of the estate is Level 1 is at Level 1. The model is applied to the estate the institution's people actually use.

## Moving up one level in a quarter

Each transition has one decisive act.

- **To Level 2:** consolidate onto one platform with institutional sign-in and administration; retire personal accounts.
- **To Level 3:** deploy the platform from source into the institution's cloud account; connect the first system of record through the broker; make the model a setting and switch it once to prove it.
- **To Level 4:** have the institution's engineers perform a deployment end to end without vendor hands on keyboards; stand up the first local open-weight model for a restricted workload; move every connector and skill into institution repositories.
- **To Level 5:** build and start the platform on a disconnected network; load one model and one connector; verify the signed-bundle update path.

Appendix F sequences these into a ninety-day plan for an institution starting at Level 1 or 2 and aiming at Level 3.

# The CIO's Readiness Assessment

A scored assessment in five sections, one per property of the framework. Each statement is scored 0 (false), 1 (partly true or true for a pilot only), or 2 (true across the estate and verifiable today). Thirty statements, sixty points. The bands at the end map a total to a level in Appendix A. The assessment is designed to be completed by the CIO with the enterprise architect, the CISO and the head of data in one sitting; disagreements about a score are usually the most valuable output.

## Section 1: You own the code

| #   | Statement                                                                                                                               | Score |
|-----|-----------------------------------------------------------------------------------------------------------------------------------------|-------|
| 1.1 | Our engineers can read the complete source of the AI platform we use, including backend services and infrastructure definitions, today. |       |
| 1.2 | The license permits us to modify the source and run the modified platform in production.                                                |       |
| 1.3 | Our engineers have built the platform from source at least once in the last year.                                                       |       |
| 1.4 | Our engineers have deployed the platform to an environment we control without vendor hands on keyboards.                                |       |
| 1.5 | The platform that runs in the vendor's cloud is the same code, byte for byte, as the one we can deploy ourselves.                       |       |
| 1.6 | If the vendor ceased to exist tomorrow, the platform would keep running and we could patch it.                                          |       |

## Section 2: You own the data, in place

| #   | Statement                                                                                                             | Score |
|-----|-----------------------------------------------------------------------------------------------------------------------|-------|
| 2.1 | AI agents read our systems of record at query time rather than from a copy ingested in advance.                       |       |
| 2.2 | Every connector between an agent and a system of record runs on infrastructure we control.                            |       |
| 2.3 | The credentials connectors use to reach our systems live in a secret store we control, never in an agent or a client. |       |
| 2.4 | Tool calls are denied by default and allowed by an explicit, versioned policy table we administer.                    |       |
| 2.5 | Every tool call is written to an audit log we own, with user, agent, tool and decision.                               |       |
| 2.6 | No copy of our institutional data exists in the vendor's infrastructure that we could not delete today.               |       |

## Section 3: You choose the model, and can change it

| #   | Statement                                                                                                                | Score |
|-----|--------------------------------------------------------------------------------------------------------------------------|-------|
| 3.1 | The model behind any agent can be changed by an administrator in a settings screen without engineering work.             |       |
| 3.2 | We have switched a production agent's model at least once and measured the result against an evaluation set.             |       |
| 3.3 | At least two commercial providers and at least one open-weight model are configured and routable.                        |       |
| 3.4 | Routing rules can confine data of a given classification to approved models, and the rules are enforced by the platform. |       |
| 3.5 | A fallback chain exists so that a provider outage does not stop our services.                                            |       |
| 3.6 | We see model spend by agent, by model and by day, against a cap we set.                                                  |       |

## Section 4: Your clients share one identity

| #   | Statement                                                                                                                                                    | Score |
|-----|--------------------------------------------------------------------------------------------------------------------------------------------------------------|-------|
| 4.1 | Every client of the platform, web, browser extension, desktop, mobile and terminal, signs in through one hand-off using our institutional identity provider. |       |
| 4.2 | No client application, ours or the vendor's, holds an organization-level credential.                                                                         |       |
| 4.3 | A team can build a new application on the platform with institutional single sign-on, roles and organization scoping without writing authentication code.    |       |
| 4.4 | The sign-in flow exists in one shared component, so a security fix lands in every client at once.                                                            |       |
| 4.5 | Multi-organization membership, switching and visiting are handled by the shared identity layer, not by individual applications.                              |       |
| 4.6 | An agent's memory, guardrails and approval behaviour are the same on every surface because the components are shared.                                        |       |

## Section 5: Your agents run inside a harness you can inspect

| #   | Statement                                                                                                       | Score |
|-----|-----------------------------------------------------------------------------------------------------------------|-------|
| 5.1 | We can read the harness that runs our agents: the loop, its tools, its limits.                                  |       |
| 5.2 | Every run has a step limit and a token budget, and a human can stop it at any time.                             |       |
| 5.3 | Approval modes exist and are set per agent, so consequential actions can require a human.                       |       |
| 5.4 | Agent memory is held in three scopes in our own database, and each person can see and switch off their own.     |       |
| 5.5 | Code an agent writes runs in a sandbox on our infrastructure with an explicit network boundary.                 |       |
| 5.6 | We have tried to violate a governance rule from the chat window and the infrastructure, not the model, refused. |       |

## Scoring

| Total    | Level         | Reading                                                                                                                                                                  |
|----------|---------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 0 to 12  | 1, Rented     | The vendor runs the estate. Begin with Section 1: without the source, the rest cannot be fixed.                                                                          |
| 13 to 24 | 2, Configured | Institutional identity and administration exist; the structure is still the vendor's. The deciding move is a source-licensed deployment into infrastructure you control. |
| 25 to 40 | 3, Deployed   | The two problems from Chapter 1 are solved for the deployed estate. Close the gaps in Sections 2 and 5, then rehearse the exit.                                          |
| 41 to 52 | 4, Owned      | The institution could run without the vendor. Prove it quarterly and add local models for restricted data if not yet done.                                               |
| 53 to 60 | 5, Sovereign  | Verify the air-gapped build annually. The remaining work is operational discipline, not architecture.                                                                    |

### Using the result

Score the estate, not the best pilot. Record who scored each statement and why; a 1 that one person thinks is a 2 is a conversation worth having. Re-score every quarter alongside the exit verification from Chapter 20, and keep the history: the trend is the real measure of a sovereignty programme, and it is the figure a board understands.

# Reference Architecture

The five layers from Chapter 18, drawn as one system, with the data flows and the trust boundaries that the body of the book argued for. The diagram is intentionally product-neutral: it is the shape a CIO should look for in any platform.

## The layers and what crosses between them

| Layer         | Components                                                                                                           | What it holds                                                                                              | What crosses its boundary                                                                                                       |
|---------------|----------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------|
| Interface     | Web app, browser side panel, desktop and mobile shells, embed widget, terminal skills                                | Nothing durable. Session tokens only, scoped to the person                                                 | User requests carrying the person's session token; rendered answers and action narration                                        |
| Orchestration | The harness: loop, tool dispatch, budgets, approval modes, stop, run narration                                       | Run state for the duration of a run                                                                        | Model calls out through the model router; tool calls out through the broker; approval prompts up to the interface               |
| Knowledge     | MCP connectors, the broker, the ontology, the sandbox                                                                | Connector code; the broker's secret store, policy table and audit log; the ontology's mappings. No records | Calls into systems of record as the person or a scoped service identity; results back to the harness                            |
| Memory        | Global, agent and shared memory stores; extraction jobs; the retrieval index                                         | Memory rows in the institution's database; embeddings for retrieval                                        | Relevant memories into the model context at each turn; extracted memories written back after a turn, subject to the three gates |
| Governance    | Model router with classification policy; broker policy; sandbox network policy; approval settings; spend caps; audit | Policies and logs                                                                                          | Nothing. Governance is enforced where the other layers' calls pass, not as a separate flow                                      |

## The trust boundaries

Four boundaries matter, and each is a line a request crosses in only one direction.

**Client to platform.** The only thing a client sends is the person's session token and their request. The only thing it receives is answers, narration and approval prompts. No organization credential, no connector secret, no raw system-of-record access ever crosses to the client side.

**Platform to model.** Every model call leaves through the router. The router applies the classification policy, selects the provider or the local model, attaches the provider credential from the platform's own secret store, and records the decision. Restricted data reaches only models on the restricted list, which in most deployments means local inference on the institution's hardware.

**Platform to systems of record.** Every tool call leaves through the broker. The broker checks the policy table, deny by default, for the person's role, the agent and the tool; attaches the connector credential from its secret store; scopes the call to the person; forwards it; and writes the audit line. Connectors run inside the institution's perimeter.

**Harness to sandbox.** Code the agent writes runs in an isolated environment whose network policy is empty by default. Allowed routes are explicit and go through the broker under the same identity as any other tool call.

## Deployment mapping

The same system maps onto the four deployment tiers from Chapter 3 by moving the boundary of "infrastructure the institution controls" inward, with no change to the components.

| Tier                        | Where the five layers run                                                      | Where the models run                                                      | Where the data path goes                                                                               |
|-----------------------------|--------------------------------------------------------------------------------|---------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------|
| Vendor's cloud              | Vendor-operated, institution-owned code                                        | Commercial providers via router; optional institution-hosted local models | Connectors reach institution systems from the vendor's environment through institution-approved routes |
| Institution's cloud account | Institution's AWS, Google Cloud or Azure tenancy, including government regions | Commercial providers via router plus local models on cloud GPUs           | Entirely inside the institution's account                                                              |
| On-premise                  | Institution's data centre                                                      | Local models on institution GPUs; commercial providers optional           | Entirely inside the institution's network                                                              |
| Air-gapped                  | Sealed network                                                                 | Local models only; local embeddings                                       | No outbound route exists; updates by signed bundle                                                     |

## What the diagram proves if it is true

If a platform matches this shape, the four questions from Chapter 3 answer themselves. The code is readable because the institution deploys it. The data is in place because the knowledge layer stores no records. The model is a setting because every call goes through the router. The air-gapped tier is possible because no component depends on

anything outside the layers shown. A CIO who can draw this diagram for their own estate, with real component names in each box, is at Level 3 or above in Appendix A.

# Hardware Sizing Tables

Sizing guidance for local inference at the three tiers from Chapter 7. The arithmetic is shown so that it can be redone when the next generation of hardware arrives. The figures for memory and bandwidth are the manufacturers' published specifications at the time of writing; the throughput figures are estimates derived from them by the method below, not measurements, and should be validated on the institution's own workload before purchase.

## The arithmetic

Two quantities determine whether a machine can serve a model and how fast.

**Fit.** A model must fit in memory with room for the working context. Parameters times bytes per parameter, plus a margin of roughly 20 percent for the context and runtime.

| Precision | Bytes per parameter | 70B model | 200B model | 400B model |
|-----------|---------------------|-----------|------------|------------|
| 16-bit    | 2                   | 140 GB    | 400 GB     | 800 GB     |
| 8-bit     | 1                   | 70 GB     | 200 GB     | 400 GB     |
| 4-bit     | 0.5                 | 35 GB     | 100 GB     | 200 GB     |

**Speed.** For one conversation, each generated token requires reading the whole model from memory once, so tokens per second for a single user is bounded by memory bandwidth divided by model size in bytes. Serving many users at once batches that read, so aggregate throughput rises with concurrency until compute becomes the limit.

Single-user ceiling (tokens/second)  $\approx$  bandwidth (GB/s)  $\div$  model size (GB).

Conversational use feels fluent above roughly 15 tokens per second; batch and background work tolerates far less.

## The department tier: Apple silicon

| Machine          | Unified memory (max) | Bandwidth (approx.) | Largest practical model (4-bit) | Single-user ceiling on a 35 GB model | Suited to                                                                                             |
|------------------|----------------------|---------------------|---------------------------------|--------------------------------------|-------------------------------------------------------------------------------------------------------|
| Mac mini, M4 Pro | 64 GB                | ~270 GB/s           | ~70B class, tight               | ~7 tok/s                             | A small team; classification and summarization at volume; a private assistant for a handful of people |

| Machine              | Unified memory (max) | Bandwidth (approx.) | Largest practical model (4-bit) | Single-user ceiling on a 35 GB model | Suited to                                                                          |
|----------------------|----------------------|---------------------|---------------------------------|--------------------------------------|------------------------------------------------------------------------------------|
| Mac Studio, M4 Max   | 128 GB               | ~550 GB/s           | ~200B class                     | ~15 tok/s                            | A department; a capable private model for a few dozen people                       |
| Mac Studio, M3 Ultra | 512 GB               | ~800 GB/s           | Several-hundred-billion class   | ~22 tok/s                            | A department needing the largest open-weight models locally, at modest concurrency |

Concurrency on this tier is limited; plan for a handful of simultaneous conversations at fluent speed, more for background work.

### The team tier: NVIDIA desk-side systems

| Machine             | Memory           | Notes from the manufacturer                                                                                   | Suited to                                                                                                                             |
|---------------------|------------------|---------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------|
| DGX Spark (GB10)    | 128 GB unified   | Positioned for models up to ~200B parameters at reduced precision; two units can be linked for roughly double | A team developing, fine-tuning and serving mid-sized models on the data-centre software stack; restricted workloads for a group       |
| DGX Station (GB300) | ~784 GB coherent | Desk-side form factor with data-centre class GPU memory                                                       | The largest open-weight models at useful precision for a team, with real concurrency; a restricted-data inference node for a division |

Both run the same CUDA software stack as the rack systems, so a model served here moves to the data centre without change.

### The institution tier: data-centre GPUs

Rack-scale DGX systems and equivalent multi-GPU servers. Sizing at this tier is a function of three inputs the institution supplies: peak concurrent conversations, the model class to serve, and the latency target. The method:

1. Choose the model and precision; compute its size from the fit table.
2. Choose the GPU memory per node; the model must fit on one node or be sharded across a node's GPUs with the serving software's support.
3. Measure, with the free serving software and the real model, aggregate tokens per second at the target concurrency on one node.
4. Divide peak demand by that figure for the node count; add one for failover.

Do not accept a vendor's single-user token rate as a proxy for step 3.

## Routing across machines

Where an institution has machines at several tiers, a local inference router (NVIDIA's Personal AI Router, open-sourced in September 2026, is one example) treats them as one pool for independent requests: each request goes to one eligible node; models are never split across nodes. Eligible nodes include RTX-class GPUs from the 20 series onward, DGX Spark and Apple M4-class systems. Use it to put idle machines to work on volume and background tasks, not to replace a data-centre node for interactive load.

## Budget lines beyond the box

- Inference serving software (free, open source) and the person who runs it.
- Monitoring and a model catalogue, supplied by the platform.
- Power and cooling, especially at the institution tier.
- A refresh cycle: assume the hardware is superseded in two to three years and the model it was bought for is superseded in one. Both are fine, because the platform's router makes the hardware a setting too.

# Connector and Broker Policy Catalogue

A starting catalogue of the systems a sovereign platform should reach through Model Context Protocol connectors, by sector, with the scopes a broker policy should distinguish and the audit fields every call produces. It is a template to be edited into the institution's own policy table, not a complete list.

## Audit fields every call produces

Whatever the connector, the broker writes one line per call with these fields. They are the minimum a regulator, an auditor or an incident review will ask for.

| Field          | Content                                                     |
|----------------|-------------------------------------------------------------|
| timestamp      | When the call was made                                      |
| user           | The person on whose behalf the call was made                |
| role           | The role the policy was evaluated against                   |
| agent          | The agent that made the call                                |
| surface        | Web, side panel, desktop, mobile, terminal, scheduled       |
| tool           | Connector and tool name                                     |
| scope          | The scope granted: self, caseload, department, organization |
| parameters     | The call's inputs, with secret fields redacted              |
| decision       | Allowed or denied, and the policy rule that decided         |
| result_size    | Rows or bytes returned                                      |
| classification | The data classification of what was returned                |
| model_target   | Which model later received the result, from the router      |

## Higher education

| System                     | Example products                               | Read scopes                                                                | Write scopes                        | Classification     |
|----------------------------|------------------------------------------------|----------------------------------------------------------------------------|-------------------------------------|--------------------|
| Student information system | Banner, PeopleSoft Campus, Jenzabar, Colleague | Own record (student); caseload (advisor); program (chair); all (registrar) | Holds, notes (registrar roles only) | Restricted (FERPA) |
| Learning management system | Canvas, Blackboard, D2L, Moodle                | Own courses (student); taught courses (instructor); all (admin)            | Announcements, grades (instructor)  | Restricted         |

| System                          | Example products                                 | Read scopes                               | Write scopes             | Classification |
|---------------------------------|--------------------------------------------------|-------------------------------------------|--------------------------|----------------|
| Recruitment and advancement CRM | Slate, Salesforce                                | Own territory (recruiter); all (director) | Notes, tasks             | Confidential   |
| Advising platform               | EAB Navigate and equivalents                     | Caseload                                  | Notes, alerts            | Restricted     |
| Ticketing and service desk      | ServiceNow, TeamDynamix, Jira Service Management | Own tickets (requester); queue (agent)    | Create, comment, resolve | Internal       |
| Identity directory              | Active Directory, Okta, institutional IdP        | Own profile; lookup by role               | None                     | Internal       |
| Document stores                 | Google Drive, SharePoint, Box                    | Own and shared-with-me                    | Create in own space      | Varies by item |
| Library and research databases  | Discovery layers, publisher APIs                 | Catalogue search                          | None                     | Public         |

## Enterprise

| System                    | Example products                          | Read scopes                          | Write scopes                                     | Classification |
|---------------------------|-------------------------------------------|--------------------------------------|--------------------------------------------------|----------------|
| CRM                       | Salesforce, HubSpot, Dynamics             | Own accounts; team; all              | Notes, tasks, opportunity fields                 | Confidential   |
| ERP and finance           | SAP, Oracle, NetSuite, Workday Financials | Own cost centre; department; all     | Draft journal entries, approvals (finance roles) | Restricted     |
| HR system                 | Workday, SuccessFactors, BambooHR         | Own record; direct reports; all (HR) | Time-off requests                                | Restricted     |
| Support desk              | Zendesk, Freshdesk, Intercom              | Own tickets; queue                   | Create, reply, resolve                           | Internal       |
| Project and work tracking | Jira, Asana, Monday                       | Own projects; team                   | Create, update                                   | Internal       |
| Collaboration             | Slack, Teams, email                       | Own channels and mail                | Post, send (approval recommended)                | Internal       |
| Data warehouse            | Snowflake, BigQuery, Redshift             | Governed views by role               | None                                             | Varies by view |
| Contracts and documents   | DocuSign, SharePoint, Box                 | Own and shared                       | Create drafts                                    | Confidential   |

## Healthcare

| System                   | Example interfaces                      | Read scopes                                                      | Write scopes                                | Classification                        |
|--------------------------|-----------------------------------------|------------------------------------------------------------------|---------------------------------------------|---------------------------------------|
| Electronic health record | FHIR APIs; vendor-sanctioned interfaces | Own patients (clinician); panel (care team); none (non-clinical) | Draft notes only, clinician review required | Restricted (HIPAA); local models only |
| Scheduling               | Practice management systems             | Own schedule; department                                         | Requests                                    | Restricted                            |
| Clinical documents       | Document management                     | Own patients                                                     | None                                        | Restricted                            |
| Billing                  | Revenue cycle systems                   | Department                                                       | None                                        | Restricted                            |

Healthcare connectors should route exclusively to models on the restricted list, which in practice means local inference on institution hardware.

### Legal and financial services

| System                     | Example products                       | Read scopes                 | Write scopes                                | Classification                       |
|----------------------------|----------------------------------------|-----------------------------|---------------------------------------------|--------------------------------------|
| Matter and case management | Clio, iManage, NetDocuments            | Own matters; practice group | Notes, drafts                               | Privileged; local models recommended |
| Document management        | iManage, NetDocuments, SharePoint      | Own and matter-shared       | Create drafts                               | Privileged                           |
| Billing and time           | Clio, Elite, Aderant                   | Own entries; partner view   | Time entries                                | Confidential                         |
| Market and reference data  | Vendor feeds                           | Licensed scope              | None                                        | Licensed                             |
| Research databases         | Legal and financial research platforms | Licensed scope              | None                                        | Licensed                             |
| KYC, AML and compliance    | Screening and case tools               | Own queue; compliance team  | Case notes, escalations (approval required) | Restricted                           |

### Writing the policy table from the catalogue

Each row above becomes several rows in the broker's policy table: one per role and tool, with the scope the role gets and whether the tool may write. Three rules keep the table honest.

- **Start empty.** Deny by default; add the first agent's needs; add the next agent's. The table should read as a history of deliberate grants.

- **Scope before allow.** Never grant a role "all" when "own" or "caseload" is what the agent needs. The source system's own permissions are the backstop, not the control.
- **Writes require an approval mode.** Any tool that changes a system of record should be paired with the ask-first approval mode in the harness, at least until the agent has an evaluation history.

# A 90-Day Sovereign Pilot Plan

A week-by-week plan for an institution at Level 1 or 2 in Appendix A to reach Level 3 with one working, governed, multi-system agent in production. It assumes a sovereign platform deployed from source into the institution's own cloud account, a vendor engineering team embedded with the institution's, and a hard problem chosen in advance. The plan is deliberately front-loaded with decisions, because the decisions are what take time; the engineering is faster than most institutions expect.

## Before day one

- **Choose the problem.** One process that spans at least two systems of record and that real people do every week. Write down what "better" would mean in a sentence a department head would sign.
- **Name the owners.** A CIO sponsor, a process owner from the department, an engineering lead, a security lead, and the vendor's embedded engineers.
- **Settle the contract.** Source under perpetual license; everything built in the pilot owned by the institution; model providers paid directly or through capped credits; no per-seat line.
- **Pick the deployment tier.** For the pilot, the institution's own cloud account.

## Weeks 1 to 2: Deploy and sign in

- Deploy the platform from source into the institution's cloud account, by the institution's engineers with the vendor beside them. Record the steps; they are the beginning of the runbook.
- Connect institutional single sign-on. Verify roles and organization scoping with three real users of different roles.
- Configure two commercial model providers and set a spend cap. Confirm the spend view shows tokens by agent.
- Run the four exit questions from Chapter 3 as a baseline. Two will be yes already; write down the other two as the pilot's structural goals.

**Milestone:** a signed-in user can chat with a default agent on the institution's own deployment, with a visible cost.

## Weeks 3 to 4: The first connector and the broker

- Stand up the broker inside the institution's account, empty policy, deny by default.

- Connect the first system of record through a Model Context Protocol connector, running in the institution's account, with its credential in the broker's secret store.
- Write the first policy rows: one role, read scope "own" or "caseload," no writes.
- Build the pilot agent with that connector and a first skill written by the process owner in plain language.
- Verify the audit log shows every call with user, role, tool and decision.

**Milestone:** the agent answers a real question from live data, as the asking user, and the call is in the log.

## Weeks 5 to 6: The second system and the ontology

- Connect the second system of record. Define the three or four concepts the pilot needs in the ontology and map them to the two connectors.
- Extend the skill to use both systems. Add the identity-resolution rule that links the person across them.
- Write the evaluation set: thirty to fifty real prompts with the properties of a good answer, graded by the process owner.
- Run the evaluation on both configured models. Record the results in the platform.

**Milestone:** the agent completes the multi-system task; the evaluation set exists and has been run.

## Weeks 7 to 8: Surfaces and memory

- Put the agent where the work happens: the browser side panel for the process owner's team, the desktop application for anyone who wants it, an embed on the relevant internal page.
- Turn on memory with the three scopes; have two users verify they can see and switch off their own.
- Write the agent's shared memory: the standing facts and rules the process depends on, by the process owner.
- Set the approval mode to ask-first for any write tool, and add the first write tool if the process needs one.

**Milestone:** the team uses the agent from the side panel in their daily work; memory and approvals are visible and governed.

## Weeks 9 to 10: Governance and the local model

- Add a classification tag to the restricted connector's results and a routing rule confining them to an approved model.

- Stand up one open-weight model on a team-tier machine or a cloud GPU; add it to the router as the restricted target and as the fallback.
- Re-run the evaluation set on the local model; record the difference.
- Attempt to violate a policy from the chat window, twelve ways; confirm the broker, not the model, refused, and that the attempts are in the log.

**Milestone:** restricted data never leaves the institution's infrastructure, and the attempt to make it leave is on record.

## Weeks 11 to 12: Operate and verify

- Hand the runbook to the institution's team. Perform one deployment of a platform update end to end without vendor hands on keyboards.
- Re-run the four exit questions. Three or four should now be yes.
- Score Appendix B for the pilot estate. Set the target for the next quarter.
- Present to the sponsor: the sentence from before day one, with the evidence: usage, evaluation results, cost by model, the audit log, the exit verification.

**Milestone:** Level 3 in Appendix A for the pilot, with a documented path to extending it across the estate.

## What the plan does not include

It does not include a data migration, because there is none. It does not include a security review of a new authentication system, because the pilot reused the institution's. It does not include a procurement cycle for hardware, because the local model runs on a team-tier machine or a rented GPU for the pilot. Each of those is a quarter in a conventional plan, and their absence is why ninety days is realistic.

# Glossary of Sovereign AI Engineering Terms

Terms as this book uses them. Where the industry uses a word loosely, the entry says what it means here.

**Agent.** A model running inside a harness: it reads its situation, decides one step, acts through a tool or answers, and repeats. The capability is mostly the harness.

**Agent memory.** What one agent remembers about one person, filed by category, extracted from conversations when permitted and editable by the person and an administrator. One of the three memory scopes.

**Air-gapped deployment.** The platform running on a network with no route to the public internet: local models, local embeddings, local identity, updates by signed bundle. The strictest deployment tier and the proof that no hidden dependencies exist.

**Approval mode.** A harness setting that decides whether consequential actions, a submit, a send, a payment, a delete, proceed automatically or pause for a human's approve-or-deny.

**Audit line.** The record the broker writes for every tool call: who, which agent, which tool, what scope, allowed or denied, what came back, which model later saw it.

**Broker.** The institution-run service every tool call passes through. It holds connector credentials, evaluates the deny-by-default policy table, scopes calls to the person, and writes the audit log. The component most AI products lack.

**Budget cap.** A ceiling on model spend the institution sets, after which the platform stops spending rather than invoicing. The financial shape that replaces per-seat pricing.

**Classification routing.** A router rule that confines data of a given classification, restricted, privileged, regulated, to an approved list of models, typically local ones.

**Connector.** A small service that exposes a system of record to agents as a set of tools, over the Model Context Protocol, translating between the model's world and the system's API. Runs inside the institution's perimeter.

**Deny by default.** The policy posture in which a tool call no rule explicitly allows is refused. The opposite of allow-with-exceptions.

**Deployment tier.** One of four ways to run the same codebase: the vendor's cloud, the institution's cloud account, on-premise, or air-gapped. A business decision, provided the

architecture supports all four.

**Embedding.** A numeric representation of text used for retrieval by similarity. In a sovereign deployment, produced by a model the institution runs.

**Evaluation set.** A collection of representative prompts with the properties of a good answer, run against a model before an agent reaches users and whenever the model or a skill changes.

**Exit verification.** The quarterly check of the four sovereignty questions: can we build and deploy from source; are connectors and skills in our repositories; can we switch the model today; does the air-gapped build start.

**Fallback chain.** An ordered list of models the router tries when the preferred one fails, ending in a local model so services never stop.

**Forward-deployed engineering.** The vendor's engineers working inside the institution's team on a real problem, with everything built belonging to the institution.

**Framework (the five properties).** You own the code; you own the data, in place; you choose the model, and can change it; your clients share one identity; your agents run inside a harness you can inspect.

**Global memory.** What the platform remembers about a person across every agent in their organization. One of the three memory scopes.

**Harness.** The loop an agent runs in, and everything around the model: tools, memory, budgets, approvals, sandbox, stop, audit. Where the institution's control lives.

**Headless application.** An agent, a skill set and an API, with a user interface only where a human needs one. The post-SaaSocalypse shape of departmental software.

**Identity hand-off.** The sign-in pattern in which an application redirects to the platform's hosted sign-in service and receives user-scoped session tokens back, never seeing a password or holding an organization credential.

**Lock-in (four kinds).** Model, identity, connector, and format-and-memory. Each defeated by a specific decision at a specific layer.

**Model Context Protocol (MCP).** The open standard for how an agent discovers and calls tools, introduced by Anthropic in 2024, adopted across the industry in 2025, and stewarded by the Linux Foundation. The reason connectors are portable.

**Model-agnostic.** The property that the model behind any agent is a setting, reachable through one standard interface, switchable without touching an integration.

**Ontology.** A description of the institution's concepts, their relationships, and the system and tool that answers for each. A pass-through: it stores mappings, never

records.

**Open-weight model.** A model whose weights are published and can be downloaded and run on the institution's own hardware. The basis for local inference and the restricted-data model list.

**Organization credential.** A key with administrative authority over an organization. Lives only on a server the institution controls; never in a client.

**Per-seat pricing.** A fixed monthly fee per person regardless of use. The pricing shape this book argues is structurally wrong above a hundred users.

**Reading in place.** Reaching a system of record at the moment of the question, through a connector, as the asking person, instead of ingesting a copy in advance.

**Router.** The component every model call passes through, which selects the provider or local model by cost, latency, capability and classification, attaches the provider credential, applies the fallback chain, and records the decision.

**SaaSocalypse.** The trade-press name for the question of what happens to software subscriptions when AI lets enterprises build applications themselves. This book's reading: the application becomes headless; data access, identity and agents remain scarce.

**Sandbox.** An isolated environment, on institution infrastructure, in which code an agent writes runs, with an explicit network boundary that is empty by default.

**Session token.** A short-lived credential representing one signed-in person with exactly their permissions in exactly their organization. Lives with the person, in the client.

**Shared component.** A library every client and every institution-built application mounts: sign-in, chat, memory, approval card, embed. The mechanism by which governance becomes behaviour.

**Shared memory.** Standing facts and rules an agent carries into every conversation, written by hand, never extracted, visible to everyone who talks to the agent. One of the three memory scopes.

**Signed bundle.** A release package of images, model weights, dependencies and a manifest, cryptographically signed, by which an air-gapped deployment is updated.

**Skill.** A plain-language document that tells an agent how to accomplish a task with the tools it has. The unit of institutional knowledge in a headless estate, and the way a coding agent learns to build on the platform.

**Sovereign.** Four yeses: readable source, movable data, switchable model, and a system that runs with the cable unplugged.

**Three gates.** The organization's memory switch, the agent's, and the person's own; a level that is off stops everything beneath it.

**Tier (hardware).** Department (Apple silicon desktops), team (NVIDIA desk-side systems), institution (data-centre GPUs). Sized by memory and bandwidth first.

**Usage pricing.** Paying for tokens consumed, or for electricity on owned hardware, under a cap the institution sets. The shape that captures the falling price of models for the institution rather than the vendor.

# Bibliography

Sources cited or relied on in the text, grouped by subject. Where a source is a living web page, the address given is the one current at the time of writing; where no stable address exists, the publication is named so the reader can find it.

## Standards and protocols

- Anthropic. "Introducing the Model Context Protocol." November 2024. <https://www.anthropic.com/news/model-context-protocol>
- Model Context Protocol. Specification and reference implementations. <https://modelcontextprotocol.io>
- The Linux Foundation. Announcement of the Model Context Protocol's move to neutral, foundation stewardship, 2025.

## Hardware

- NVIDIA. "DGX Spark." Product specifications. <https://www.nvidia.com/en-us/products/workstations/dgx-spark/>
- NVIDIA. "DGX Station." Product specifications. <https://www.nvidia.com/en-us/products/workstations/dgx-station/>
- NVIDIA. "Personal AI Router." Project page and system requirements, September 2026. <https://www.nvidia.com/en-us/ai-on-rtx/personal-ai-router/>
- Apple. "Mac Studio." Technical specifications. <https://www.apple.com/mac-studio/>
- Apple. "Mac mini." Technical specifications. <https://www.apple.com/mac-mini/>

## The SaaSpocalypse debate

- Horton, Christine. Coverage of the "SaaSpocalypse" in CRN UK, 2026: whether enterprises will keep paying for software subscriptions when AI lets them build applications themselves, and what happens to per-seat pricing.

## Cost and pricing

- ibl.ai. "Pricing." Self-serve credits, the free tier and enterprise terms. <https://ibl.ai/pricing>
- ibl.ai. "The True Math of Per-Seat University AI." <https://ibl.ai/blog/university-ai-per-seat-cost-true-math>
- ibl.ai. "Enterprise AI: Ownership vs. Rental Cost." <https://ibl.ai/blog/enterprise-ai-ownership-vs-rental-cost>

- Amigot, Miguel. Remarks on per-seat list pricing versus developer token pricing, panel on AI and enrollment management, Harvard Graduate School of Education, Leadership in Strategic Enrollment Management, 2026.  
<https://www.gse.harvard.edu/professional-education/program/leadership-strategic-enrollment-management>

## Architecture and agents

- ibl.ai. "The Five-Layer Agent Architecture: What Ships Is Layer One."  
<https://ibl.ai/blog/five-layer-agent-architecture-what-ships-is-layer-one>
- ibl.ai. "NVIDIA PAIR: the Personal AI Router and the Local Inference Cluster."  
<https://ibl.ai/blog/nvidia-pair-personal-ai-router-local-inference-cluster>
- ibl.ai. "Your ibl.ai Agents in a Chrome Side Panel, on Any Page." Platform update, October 2026. <https://ibl.ai/updates/chrome-extension-agents-side-panel>
- ibl.ai. "Shared Memory: Facts Every Agent Chat Starts Knowing." Platform update, October 2026. <https://ibl.ai/updates/agent-shared-memory>
- ibl.ai. "Security model of a vibe app." Developer documentation.  
<https://ibl.ai/developer/vibe/security-model>
- ibl.ai. "How sign-in and organizations work." Developer documentation.  
<https://ibl.ai/developer/vibe/auth-model>
- ibl.ai. "Memory System." Platform documentation.  
<https://ibl.ai/developer/platform/agents/memory>
- ibl.ai. Agent skills catalogue. <https://ibl.ai/developer/skills>
- ibl.ai. Platform documentation. <https://ibl.ai/docs/os/overview>

## Source code

- iblai/os. The agent operating system: web, desktop, mobile and the Chrome side-panel extension (extensions/chrome). <https://github.com/iblai/os>
- iblai/vibe. Skills and the application starter. <https://github.com/iblai/vibe>
- iblai/infra-cli. Infrastructure provisioning for self-hosted deployments.  
<https://github.com/iblai/infra-cli>
- ibl.ai Chrome extension. Chrome Web Store listing.  
<https://chromewebstore.google.com/detail/iblai/gppjhejgkdljdhnmlfghfbalfhgaepfd>

## Deployments cited

- ibl.ai. "AI Sovereignty at Syracuse University." Case study. <https://ibl.ai/case-study/syracuse-university>

- Amigot, Miguel. Remarks on a SUNY IITG deployment and on Syracuse's API-based deployment, Harvard Graduate School of Education panel, 2026 (above).

## Companion volume

- Amigot, Mikel. *The AI-Native University: Blueprint for the Institution of the Future*. Memorare Press, 2026. <https://ibl.ai/book/ai-native-university>

# About the Author

Miguel Amigot is Chief Technology Officer of ibl.ai, where he leads engineering, architecture and product. His background is in computer science and electrical engineering, and it shapes how he thinks about AI infrastructure: as systems that keep an organization free to run any model and switch at any time, and in full ownership of its own code and data.

He works with universities, companies and government agencies to deploy agentic AI inside their own infrastructure, so that they avoid the vendor lock-in and per-seat pricing that make AI prohibitively expensive at scale. The platform described in this book, its model router, its connectors and broker, its identity hand-off and its harness, is the one he and his team build and run.

He has made the case for sovereignty and ownership to institutional leaders in several settings: on a panel on AI and enrollment management at Harvard Graduate School of Education's Leadership in Strategic Enrollment Management program, alongside the founders of EdSights and Kai and moderated by Paul LeBlanc; and at a Banco Santander gathering of roughly one hundred university presidents in Mexico City, on what agentic AI makes possible in higher education today. His throughline in both was the one this book expands: an institution should control its own code and data, and the technology now makes that practical rather than merely principled.

Miguel's writing and talks are collected at [ibl.ai/miguel](https://ibl.ai/miguel). He lives and works in New York.

*Sovereign AI Engineering* is his first book, and the second title from Memorare Press on building institutions around artificial intelligence, following *The AI-Native University* by Mikel Amigot.

# About Memorare Press

Memorare Press publishes books on how institutions change when artificial intelligence becomes a foundation rather than a feature. Its titles are written by practitioners who build the systems they describe, and each is published free to read online, one page per section, with a downloadable edition.

## Published:

- **The AI-Native University** (Higher Education), by Mikel Amigot.  
<https://ibl.ai/book/ai-native-university>
- **Sovereign AI Engineering** (For the CIO), by Miguel Amigot.  
<https://ibl.ai/book/sovereign-ai-engineering>

## Upcoming titles in the AI-Native series, by Mikel Amigot:

- **The AI-Native School** (K–12)
- **The AI-Native Enterprise** (Business)
- **The AI-Native Finance** (Financial Industry)
- **The AI-Native Government** (Public Sector)
- **The AI-Native Hospital** (Healthcare)
- **The AI-Native Society** (Broader societal implications)

# About ibl.ai

ibl.ai is an Agentic AI Operating System that organizations deploy on their own infrastructure with full source code and data ownership. It supports any LLM, any cloud, and 160+ AI agent templates — at 85% lower cost than per-seat alternatives. Used by 1.6M + users at 400+ organizations including NVIDIA, Google, MIT, Syracuse University, and George Washington University across higher education, enterprise, government, K-12, legal, financial services, and healthcare.

ibl.ai eliminates vendor lock-in. Organizations choose any LLM provider, deploy on any infrastructure, and pay only for what they use — not per seat. Self-serve plans run on rechargeable credits — a budget cap you set, pooled across every user, model, and agent, with no per-seat fees and auto-refill opt-in. Enterprise engagements are one-time, not subscriptions: a pilot from \$15K, full integration and deployment at \$25K–\$80K, or a six-figure codebase transfer that hands you the full source under a perpetual license with complete data sovereignty. Unlike ChatGPT, Copilot, or Gemini licenses at \$20-60/user/month locked to one vendor, ibl.ai costs a fraction at any scale — and you own everything you run.

At the center of the platform is Agentic OS — the flagship and single most important product, the core AI operating system that every other ibl.ai product is built on. It unifies institutional data via MCP-based interoperability, powers the AI agents, and provides the shared per-user memory, model-agnostic LLM routing, identity/SSO, and governance layer that the rest of the products run on top of as modules: app development (Agentic Vibe), learning management (Agentic LMS), course design (Agentic Course), video production (Agentic Video), and verifiable credentials (Agentic Wallet). Professional services include AI transformation, forward-deployed engineering, Claude agent development, AI NYC studio production, and faculty enablement.