RJL SOLUTIONS

A PRACTITIONER'S GUIDE

The Day After the Demo

Getting AI Out of the Pilot and Into the Work

Richard White
RJL Solutions · rjlsol.com

First edition · 31 October 2026

The Day After the Demo
Getting AI Out of the Pilot and Into the Work
First edition, 2026

Published by RJL Solutions, LLC
New Jersey, United States
rjlsol.com

Copyright © 2026 Richard White. All rights reserved.

You may download and share this PDF free of charge, provided it is distributed in its entirety, without alteration, and with this copyright notice intact.

Except as permitted by applicable copyright law, including fair use, or by the permission granted above, no part of this publication may be reproduced, adapted, sold, or republished without prior written permission.

For permissions, contact rwhite@rjlsol.com.

For the latest edition and companion materials, visit rjlsol.com.

A note on the examples

Every scene in this book is a composite, built from patterns rather than from any one organization, and no employer or client is identified. The numbers in worked examples, sample records, targets and implementation sequences are illustrative; none is a client result. The recurring claims assistant prepares case context, and examiners keep the case decision. Any later authority, such as creating internal notes, is a separate scope with its own approvals.

Vendor disclosure

I have worked closely with several vendors named in this book, including Cyera, ServiceNow, Snowflake, Databricks, DataHub and Microsoft, as a customer and practitioner. No vendor reviewed or paid for this book. Products, models and tools, including Cursor and the assistants and models of OpenAI, Anthropic, Google, Microsoft and AWS, are named as examples of patterns or candidates to test, not endorsements. Any substitute must keep the same contracts and controls.

Views and currency

The views in this book are my own and do not represent those of any current or former employer or client. Technical patterns need validating in their actual environment. Legal, product and supplier references are dated, and internal acceptance is not external certification. Product observations were checked between 19 and 29 October 2026 and will change; the numbered notes give each source.

Contents

PREFACE	Why now	5
INTRODUCTION	The day after the demonstration	10
Part 1 Own the outcome		**16**
CHAPTER 1	Start from a decision and a baseline	17
CHAPTER 2	Owners with real authority	24
CHAPTER 3	Redesign the work and the review	30
Part 2 Build the team that runs it		**43**
CHAPTER 4	Choose teams that can keep the service running	44
CHAPTER 5	A platform that removes repeated work	50
CHAPTER 6	Fund the running and the people	55
Part 3 One system across AI, data and security		**63**
CHAPTER 7	Trace the four-part system	64
CHAPTER 8	Build on the systems you already run	75
CHAPTER 9	A lean implementation and its limits	80
Part 4 Data the business can trust		**86**
CHAPTER 10	A data product is a maintained promise	87
CHAPTER 11	Certification that expires when the evidence changes	96
CHAPTER 12	Meaning before the query	101
Part 5 Agents you can answer for		**113**
CHAPTER 13	Give an agent a bounded operating contract	114
CHAPTER 14	Enforce permissions beyond the prompt	124
CHAPTER 15	Make data AI-ready with DSPM and governed access	132
CHAPTER 16	Operate governance with people and agents	146
CHAPTER 17	Evidence that supports an inspection	154
Part 6 Delivery and adoption are part of the product		**161**
CHAPTER 18	Adoption through changed daily work	162
CHAPTER 19	Change the software lifecycle around governed intent	167
CHAPTER 20	Release, operate, transfer and retire	176
CONCLUSION	Your first governed capability	184
APPENDIX A	Production readiness diagnostic	189
APPENDIX B	Product contract and certification records	191

APPENDIX C Capability ownership inventory 195
APPENDIX D Product fit and replacement tests 201
APPENDIX E Rules by jurisdiction and use 204
APPENDIX F Economics and operating templates 207
Glossary 212
Notes and sources 216
About the author 225

Illustrations

Figure 1. The evidence between tool use and realized value 6
Figure 2. Where production detail can run out 11
Figure 3. A claims request traced through identity, data, evidence and action 12
Figure 4. Workflow redesign and data foundations as routes to production 13
Figure 5. Total cost of ownership beyond the demonstration 20
Figure 6. Decision rights that contribute to a release 25
Figure 7. Skills evidence maintained across the work lifecycle 32
Figure 8. Choosing task authority from consequences and judgment 37
Figure 9. Capacity and cash value in the Chapter 1 annual example 57
Figure 10. Illustrative cash economics of three service designs 59
Figure 11. Choose material value and prove a bounded workflow 60
Figure 12. AI, data and security designed together, inside one operating model 65
Figure 13. Four architectural responsibilities connected by authoritative records 66
Figure 14. How the brain receives validated context 68
Figure 15. Governance as four rings: dependencies, not a sequence 81
Figure 16. From source to published product: preparation stages and the point of promise 88
Figure 17. Foundational invoice and payment products composed into a derived receivables product 91
Figure 18. Delivery stage, assurance scope and operating condition 98
Figure 19. Procedure validity and the history of a corrected policy date 104
Figure 20. Shared concepts and governed domain ontology versions 108
Figure 21. Action states, approval and uncertain outcomes 115
Figure 22. Three kinds of evidence for one release decision 122
Figure 23. Access enforcement and changes across derived copies 128
Figure 24. DSPM in the governed AI data path 136
Figure 25. Governed development from metadata to runtime 142
Figure 26. Governance operations with people and bounded agents 147
Figure 27. Linked records for one request 158
Figure 28. Architecture as a versioned delivery contract 169
Figure 29. Agentic software delivery from governed intent to observed outcome 171
Figure 30. Planning periods and the proof for the next commitment 185

PREFACE

Why now

I started this book because I kept seeing the same pattern. I'm finishing it because the pattern has become expensive. For twenty-five years, across insurers, banks, a central bank and a global professional-services firm, I watched one wave of technology after another arrive with a promise and then settle into a pilot. That used to be a tolerable waste. With AI it isn't. The technology can now change how work gets delivered and paid for, and a useful capability stuck in a pilot is money left on the table.

Broad adoption, uneven returns

McKinsey's 2026 State of AI survey shows how wide the gap is. Forty-four percent of respondents said their organizations were scaling AI across the enterprise, up from 38 percent a year earlier. Thirty-seven percent reported some enterprise EBIT impact. Only 6 percent qualified as high performers. The survey defined those as organizations with at least 5 percent of EBIT attributable to AI plus significant self-reported value. Remember that respondents reported these numbers themselves; nobody audited them. [42]

Don't subtract one of those percentages from another and call it a conversion rate. They measure different things. A team can show heavy adoption without a better process. It can show hours saved without a dollar of cash benefit. So throughout this book I follow a longer chain: approved use, then changed work, then a measured outcome, then the action that turns the outcome into value.

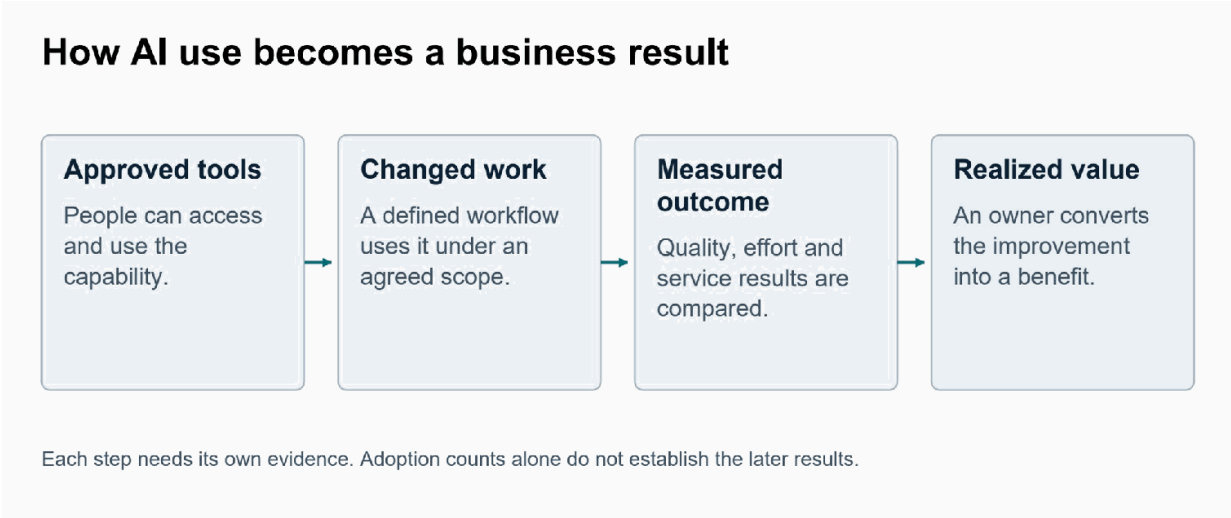


Figure 1. The evidence between tool use and realized value

Stanford's June 2026 research note looked at twelve economic indicators and found no decisive sign of an AI-driven takeoff. Seven were flat, total factor productivity among them. [43] That note tests an unusually aggressive growth scenario. It does not say AI has produced nothing, and it can't explain why your own program stalled. For a manager, the lesson is simpler. Measure the workflow in front of you.

Where it hits hardest

I sort industries with two questions. How much of the work can AI do? And if it does that work, does it break the way the firm makes money? Where both answers come back high, the business model has to change along with the workflow.

Consulting, law and IT outsourcing mostly sell expert time. They price by the hour or by headcount. So when AI shortens the tasks those contracts charge for, the firm has a pricing problem as well as a delivery opportunity. In the Thomson Reuters Institute's 2026 survey, 71 percent of in-house legal professionals expected their outside firms to change their commercial models as AI use grows; 28 percent of law firms said they had already changed pricing. [45] The researchers surveyed the two groups separately, so read them as two views of the same pressure. Insurers face a different version of the question. What does automation do to expense, service, underwriting and claims results inside a model that already works?

The same two questions explain why so many programs report savings and so little growth. Full autonomy is oversold. In many core businesses it runs straight into how the firm earns its revenue. Take a mutual life insurer whose holistic advice reaches customers through career financial professionals. There, the relationship is the product. An AI that advised customers on its own wouldn't be a better workflow. It would be a different com-

pany, with a different channel, different compensation and a different regulatory footing. Today's value lies in making the professional better: preparing the conversation, servicing the policy, following up on what was promised. Growth from autonomy comes only if leaders decide to change the model. That's a strategy call.

Elsewhere I would look at both the workflow and the business model. Clinical documentation is a strong candidate for giving clinicians time back. Retail and media raise harder questions about distribution and about who owns the customer. Treat those as places to look; nobody has measured results there yet. The management job stays the same everywhere. Find where a capability changes an outcome, put the whole service into production, and keep it dependable.

Banks and insurers

I had banks and insurers most in mind while writing. Intake, processing and review all offer real openings for AI. The pace and the consequences, though, differ from one workflow to the next. PwC's January 2026 workforce analysis lists the risks leaders should plan for: expertise concentrating in a few experienced people, fewer entry-level learning opportunities, changed work for managers and blurred decision ownership. [44] That's analysis of what could happen. It isn't evidence that insurers have already stripped out those roles.

The learning risk worries me most. PwC warns that while a few experts get stronger, everyone else gets less practice at challenging an AI result. [44] Stanford's payroll sample points the same way. Since November 2022, employment for 22- to 25-year-olds in the most exposed occupations fell at an annualized 3.8 percent, while the least exposed grew 2.0 percent. [43] It's a selected sample, and correlation isn't causation. Still, the question it raises is practical. Where will the next generation of examiners and analysts get the judgment they'll need to supervise the models?

That's why so much of this book is about people and ownership. Named owners with real authority (Chapter 2), review designed and funded as a job (Chapter 3) and training for the changed work (Chapter 18) all belong to the control system. They sit alongside technical boundaries that are actually enforced, and independent challenge where the risk calls for it.

What I would start and stop

Boil the industry picture down to a few moves, and these are the ones I would make.

- Start measuring AI by completed work and business outcomes. Pilots launched and prompts written tell you very little.

- Start naming a human owner for every AI-assisted decision in a regulated workflow: someone with the authority to change the process and to stop it.
- Start treating well-defined, well-owned data as an asset that compounds. Its quality and permitted use limit what an agent can do as much as the model does.
- Start redesigning how people learn the business, so junior staff still build the judgment the firm will need.
- Stop tolerating shadow AI. In the same Thomson Reuters survey, 34 percent of law-firm professionals said they used AI tools their firm hadn't authorized. [45] Give people capable, sanctioned tools and a supported way to use them.
- Stop buying point tools by reflex. Thirty-two percent of McKinsey's respondents said their organization had passed on at least one software product because it could build the capability internally with agentic coding tools. [42] Build versus buy deserves a fresh look, on full lifecycle cost.

The person behind the request

An assistant needs more than the words in a question. It has to know who's asking, what they're trying to get done, and which person, case or piece of business they mean. Picture three people: a claims examiner preparing a coverage review, a claimant asking about her own case, and a finance partner reviewing reserves. All three can type the same word. Each needs a different answer, and each has a different right to see it.

Inside a firm, it works the same way. An employee checking her own benefits, a staffing coordinator matching people to work and an engagement leader looking at work in progress are in different situations. Identity, department, current assignments and the reporting period can all matter. A persona can set the vocabulary and tone of a reply. Tone can even tell you when to ask a clarifying question. Neither one can establish who someone is, what they're allowed to do, or whether a business fact is true.

I use two terms for this. A context specification describes what a task needs to know. A context record holds the resolved values for one request. The application builds the record from verified identity, permitted source systems, approved definitions and whatever the user has to clarify; the model sees only the slice it needs. Chapters 7 and 12 follow that path, including what happens when a required fact can't be resolved. Get this right and context becomes something you build and test, instead of something the demo quietly assumed.

What this book is for

This isn't a forecast. The figures above will be stale within a year, and the products I name later will go stale sooner. What follows is a method for the part that lasts: deciding who owns the result, what proof counts, and who runs the service after the builders move on. A firm with those disciplines can adopt the next model faster. It will still need to prove each new combination of task and model for its intended use.

The Introduction begins where most programs stall. The day after the demo worked.

INTRODUCTION

The day after the demonstration

The meeting goes well. A small team shows an assistant answering questions about claims. It finds the right procedure, summarizes the file and drafts a note to the customer. People are impressed. Then someone asks the obvious question: how soon can we roll this out?

Six months later the assistant is still a pilot. Nobody decided it had failed; it just never became something people could rely on. The data team is waiting for someone to choose a customer definition. Security hasn't approved the service account. The claims leader likes the idea but hasn't changed how her examiners work, so they use it now and then and check everything twice. Meanwhile, the vendor is proposing a bigger license.

This book is about the distance between that meeting and a tool people use every day. I've spent my career connecting data and technology to the operations of insurers, banks, a central bank and a global professional-services firm. I've seen this pattern often enough to stop calling it bad luck. Most of the causes are decisions about ownership, data and operation. The demo never had to face any of them.

The demo proves less than you are about to fund

A demonstration shows a model producing useful output on chosen cases, under conditions someone set up. That's a real achievement. The room, though, tends to hear something far bigger: that the business can rely on that output at scale, with ordinary users, changing data and real constraints. You earn that larger claim only through representative testing and by watching the thing run.

The gap sits in the conditions the demo quietly supplied. Someone picked the documents. The presenter knew what each term meant. A service account could reach everything. A plausible answer ended the exercise, and when something broke, the builders fixed it on the spot. Daily operation supplies none of that.

Table 1. What the demonstration leaves to establish

In the demonstration	What everyday operation requires
Someone selects the right documents	A maintained route to sources that apply and that the user may see
The presenter knows what a term means	Approved definitions, and a way to choose the right one for the purpose

In the demonstration	What everyday operation requires
A service account can reach everything	Authority limited to the person asking and the action allowed
A plausible answer ends the exercise	A useful result, a review path and a measured outcome
The builders handle problems informally	Named owners for support, recovery, change and maintenance

Look at what the right-hand column covers: information, authority, ownership and value. A fluent answer settles none of them. Model quality still matters. But a better model won't resolve a disputed definition, and it won't find someone to take the support calls. Each discipline has its own job. Data supplies the case facts, the relationships and the dates that apply. Security decides which of those facts this caller and this workload may use, and what they may do with them. The model interprets and drafts within that view. Operations keeps the result running and responds when something it depends on fails. Excellence in one of those can't make up for a serious failure in another.

As a program moves past its demo, the detail tends to thin out. The user sees a polished answer. Behind it, definitions may be unsettled, a shared account may have far too much access, and support may depend on the two people who built it. That's the moment to give the plumbing the same attention as the screen.

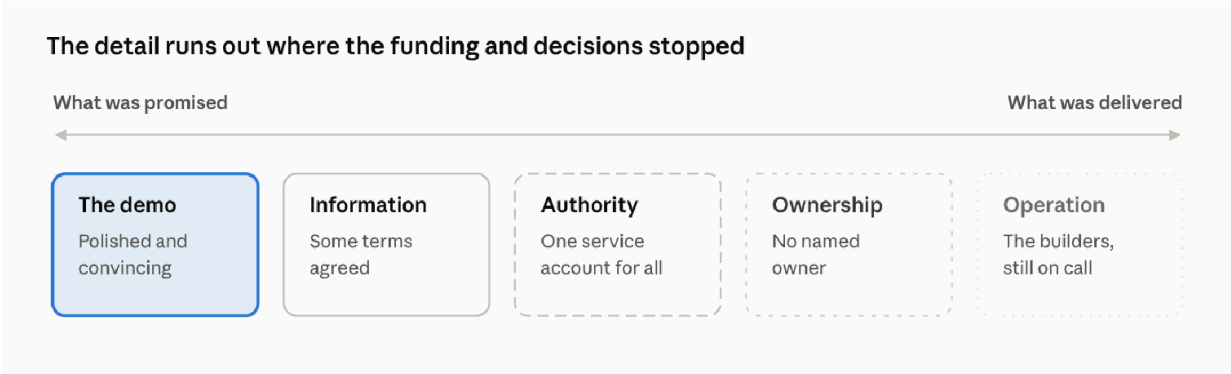


Figure 2. Where production detail can run out

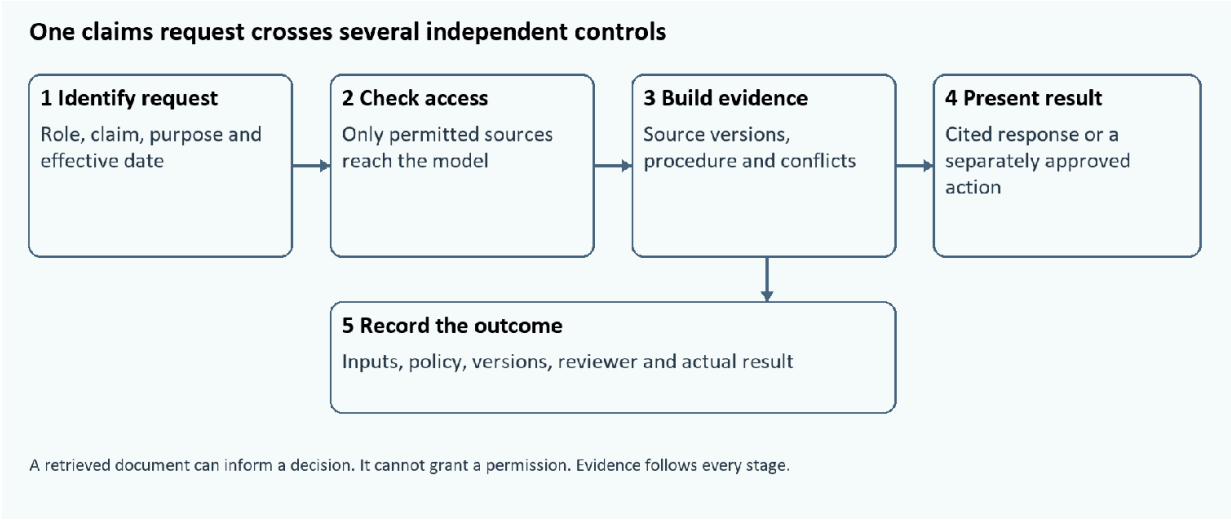


Figure 3. A claims request traced through identity, data, evidence and action

Follow one request past the screen

Want to see the gap fast? Follow one request all the way through. Take the claims assistant again. It's a composite, built from patterns I've seen in several organizations, and it comes back throughout the book. An examiner asks for the status of a claim and the procedure that applies. In the demo, the assistant returned both and everyone nodded. In daily use she needs the status of the right claim, the procedure that applied to that policy on that date, and nothing she isn't entitled to see. A confident answer can miss on any of those and still look exactly like a correct one.

That request carries two identities: the examiner asking, and the person or case she's asking about. Selecting a claim doesn't prove she may see it. Before it hands anything to the model, the application has to verify who she is, establish which cases she can work on and pin down what she's trying to do. A claimant asking about her own claim goes down a different path, with different limits.

Now list what has to be true for the answer to be right.

- The information is right. The person linked to the claim could be the insured, the policyholder, a beneficiary, an employer or a representative. Link the wrong role and you get a fluent, wrong summary. Choosing that link is a business decision, and somebody has to own it.
- The authority is right. The examiner may read the case. An agent acting for her may not automatically update it, pay it or contact the customer. Each of those is its own permission, enforced where the data is read and where the action happens.
- Someone owns the result. When a definition is disputed or the procedure changes, a named person has to be able to decide, and a team has to be able to change the service.

- The value is real. Whether people liked the answer is beside the point. Did completed, useful work go up once you count the checking, the corrections and the exceptions? That's what matters.
- It can be run. Someone takes the incident at two in the morning, shuts off a tool, rolls back a release and pays for the upkeep.

I ask teams to walk through this with one real request before they ask for more money. Wherever their explanation turns into an assumption, the release still has work to do. It's the simplest diagnostic I know. It works as well in a bank as in a portfolio company.

Why it keeps happening

If the causes are this predictable, why do so many organizations keep repeating them? I see three reasons.

First, the funding model rewards the demo. Innovation budgets pay for proofs of concept; operating budgets pay for running things. The new capability falls between them, and nobody gets funded for the dull work of definitions, permissions, support and measurement.

Second, the work is split across functions that rarely design together. The AI team builds the assistant. The data team owns the warehouse, security owns access, and the business owns the process. Each does its part well, and the whole still has no owner. In Part 3 I argue that AI, data and security form one system and need designing as one.

Third, AI gets bolted onto the existing job. Add an assistant to an unchanged process and you get a small gain at best, because the examiner still does every old step and now checks the machine too. The bigger gains come from deciding which steps go away, where people keep their judgment and who owns the outcome, and then measuring the new process. That work is uncomfortable. It's also where the money is.

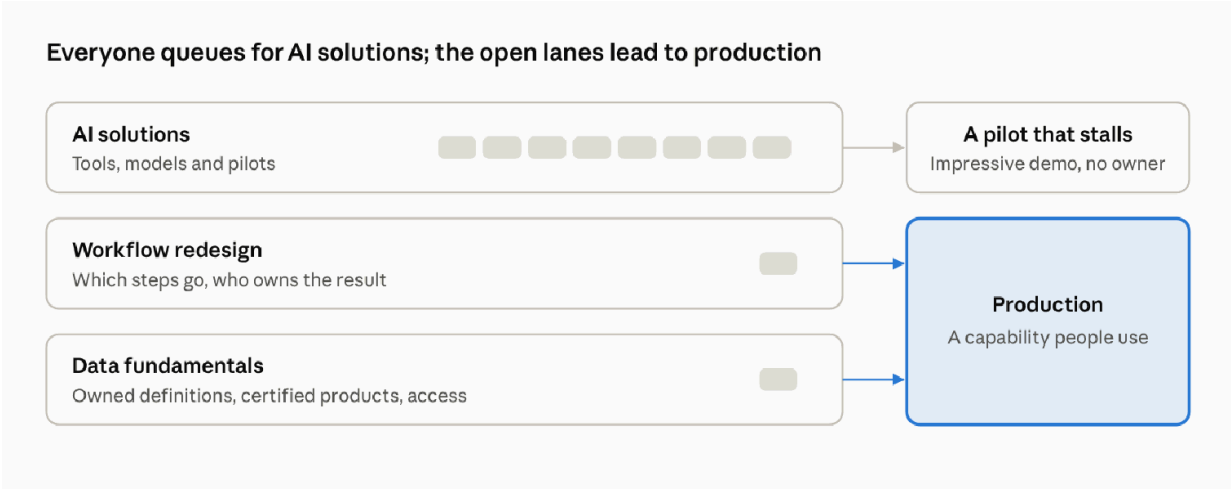


Figure 4. Workflow redesign and data foundations as routes to production

Figure 4 is a teaching picture, not a survey result. It shows a familiar imbalance: money flows to tools while the process decisions and data responsibilities stay unresolved. Parts 1 and 4 deal with the work and data changes that have to happen alongside the technology.

What I recommend

The recommendation fits in a sentence. Fund one complete workflow end to end, with real users, approved information, enforced authority, a measured outcome and an owner who stays, and let its results decide whether you expand it, reshape it or stop.

The rest of the book explains how to do that well. It also shows how to do it again without starting over each time. The six parts follow the order in which the questions come up.

- Own the outcome. Start from a business decision, name owners who can actually decide, and redesign the work instead of decorating it.
- Build the team that runs it. Stable domain teams on a shared platform, with roles and funding for running the service as well as building it. In a smaller firm that means a few part-time roles, not a department.
- One system across AI, data and security. Design the three together, on the systems you already run, with a lean version for companies that need results without a big program.
- Data the business can trust. Treat data as products with owners and promises, certify what people rely on, and settle meaning before anyone runs a query.
- Agents you can answer for. Give agents limited, written authority, make protected information usable, run governance with people and automation together, and keep records an inspector can follow.
- Delivery and adoption are part of the product. Change the daily work, change how software gets built around approved intent, release only what passes its tests and hand over something your own team can run.

The Conclusion turns all of this into a 90-day plan with a decision at the end of each phase.

Who this book is for

I wrote for two kinds of leaders. They face the same problem under different constraints.

The first group runs regulated enterprises: banks, insurers and other supervised firms. You need named owners, tested controls and release records that a risk committee or an examiner can inspect. And you need to move faster than your governance process was built for. The second group runs fast-moving companies: private-equity portfolio compa-

nies, venture-backed firms and mid-size businesses. You need AI working without a large program, with a small senior team and just enough governance to pass diligence. Where the two need different answers, I say so. Chapter 9 walks through a lean version from start to finish.

Both need proof in proportion to the consequences. The legal duties, review arrangements and scale will differ.

How to read it

I've tried to write the way I was trained to solve problems. Each chapter opens with its answer, gives the reasoning, then ends with what to do. Read the first page and the closing checklist of every chapter and you'll have the argument.

Executives should start with Chapters 1, 2 and 6, then read Chapters 16 and 20 and the Conclusion to see how investment, governance and release decisions connect. Technical readers can follow the build through Chapters 7 to 17, Chapter 19 and the appendices. Everyone should read Chapters 3, 4 and 18 before settling on an operating model. Those chapters describe the work, and the human capacity the technology depends on.

Two books shaped that approach, and I recommend both. Charles Conn and Robert McLean's Bulletproof Problem Solving gives you a discipline for defining and breaking down a problem. Barbara Minto's pyramid principle gives you a way to organize an answer so the reasoning holds up. [36; 39] The request trace simply applies those two ideas to an AI workflow. Work out what has to be true. Then lay the argument out so anyone can check it.

The examples are composites. Product names appear mostly in Part 3 and the appendices, dated and labeled, because products change faster than principles. Nothing here depends on a particular vendor, model or regulator. It depends on three decisions: who owns the result, what counts as proof, and who will run the service after the builders leave.

That's where we begin.

Further reading

- Charles Conn and Robert McLean, Bulletproof Problem Solving (Wiley, 2019): defining and disaggregating the problem before analysis.
- Barbara Minto, The Minto Pyramid Principle (1996 edition): answer first, supporting ideas grouped beneath it.
- Richard White, 25 Years of Lessons Learned as a Practitioner (RJL Solutions, 2026): forty lessons in eight themes, the field notes behind several chapters. [87]

PART 1

Own the outcome

CHAPTER 1

Start from a decision and a baseline

Pick a first capability that improves a decision or task people repeat every day. Give it an owner who can change how that work gets done. Measure where you start before anyone builds a thing. Choose the capability for its value and for a realistic path through data, access and review; how well it demos is beside the point. Then close the first phase with a real choice: proceed, reshape or stop.

That sounds obvious. Most programs start somewhere else.

How programs usually begin

They start with a list. A steering group collects ideas from every function, plots them on a two-by-two of value and effort, and funds the top ten as proofs of concept. Six months on, the company has ten demos and ten enthusiastic sponsors. Nobody uses any of them.

Why does the list mislead? Its scores cover two things only: how valuable a good answer would be, and how hard the model is to build. The questions that actually decide the outcome never make it onto the chart. Can anyone change the surrounding work? Do the sources have owners? Can you measure the result? Ask those first, before anybody ranks anything.

A quieter bias sits underneath. Ideas that demo well get funded. A chatbot that answers questions about policy documents demos beautifully, while a change to how finance reconciles intercompany balances doesn't, even when the reconciliation work is worth ten times as much. Executive visibility and demo polish make poor guides to value.

So start at the other end. Start with the work.

What makes a good first workflow

I look for five properties, and I use them as a filter before any scoring begins.

Table 2. What makes a good first workflow

Property	What it means in practice
A repeated decision or information task	Something done many times a week, by many people, in a similar way
Identifiable users	A defined group who do the work today and will use the result

Property	What it means in practice
An owner who can change the work	A business leader with the authority to change the process and accept results
Accessible source material	The documents and data exist, and someone can approve their use
A boundary narrow enough to test	A slice you can prove end to end in weeks, not a whole function

A claims support assistant can pass that filter. So can a finance reconciliation workflow, or a controlled internal knowledge service. A general-purpose assistant can have an owner too. The trouble is breadth: it does so many different things that scoping it, testing it and proving its benefit all get harder. Know exactly which work the first dollar is supposed to improve.

Take the composite insurer that runs through this book. Its candidate is assembling case context for claims examiners: status, history, parties and the procedure that applies. Examiners do this dozens of times a day. The head of claims operations can change how they work, the sources exist (scattered, but they exist), and the first slice can stay inside one product line in one region.

The composite distributor, a private-equity-backed company, picks something different. Its candidate answers customer questions about order status and credit holds. Service agents spend much of the day looking things up across three systems. The chief operating officer owns the process, and the sources already sit in the company's data platform.

Neither is glamorous. Both can be measured, owned and proven.

Map the work as it happens today

Before anyone designs a solution, draw the work as it happens now. Who gathers the information? Who reconciles it, and who reviews it? Where does it wait? And what happens when the answer is wrong?

The map pays for itself twice. First, it shows which parts of the job AI could genuinely improve. Second, it exposes the parts that need a data or process fix first, because no model can settle them. In the claims example, the map usually reveals examiners losing a lot of time working out who is who on the file. Three systems record parties differently. That's a definition problem, and an assistant will only help people consume the confusion faster.

The map hints at the redesigned job, too. Suppose examiners spend most of their time gathering context and very little exercising judgment. Then the opportunity is to remove the gathering, and bolting a chatbot onto it misses the point. Chapter 3 picks this up.

Measure the baseline before you build

Measure first. Set targets after.

Choose a baseline period that covers normal workload plus the exceptions that matter. Agree up front how you will treat shifts in case mix, seasonality and adoption. Then compare against the whole future job, including the hours people will spend reviewing and fixing the machine's output.

Table 3. Measure the baseline before you build

Measure	What to establish before the build
Business outcome	Cycle time, rework, reconciliation effort or another result of the workflow
Volume and adoption	Eligible transactions, actual use and completed work
Quality and risk	Material errors, escalations, prohibited disclosures and policy breaches
Full operating cost	Model, compute, retrieval, support, review and ongoing data upkeep
Reuse	Which certified data products or controls the next workflow can use

Write down the target, and the outcomes you won't accept, before the team sees a single result. Finding a flattering measure after the fact is remarkably easy. Your comparison has to stand up. That means a measured baseline, a comparison group working the old way, or another design you can defend, with its limits stated plainly. Reuse counts and projected cost curves? Those are design arguments. They become results only once you observe them.

Do the economics honestly

For planning, I use a simple estimate of annual net capacity-equivalent value:

Annual net capacity-equivalent value = $V \times A \times (M \div 60) \times H - C$

V is eligible annual volume. A is expected adoption, as a fraction. M is net minutes saved per adopted case after the extra review and correction. H is the loaded hourly cost of the people doing the work. C is the incremental annual cost of running the service. Keep time in hours and money in one currency. If some adopted cases never end in an accepted, useful result, adjust for acceptance and charge the failed attempts. And don't count review twice, once in M and again in C.

Here's a worked planning example with assumed numbers. Take 120,000 eligible cases, 60 percent adoption, six net minutes saved per adopted case, a $65 loaded hourly cost and $220,000 of incremental annual operating cost. You release 7,200 hours of capacity, worth $468,000 at labor cost. Subtract the operating cost and the capacity-equivalent value comes to $248,000. Treat that as a scenario to test.

Two cautions keep it honest.

Time saved is capacity, not cash. It becomes a saving only when staffing, overtime or bought-in capacity actually changes. Keep that distinction visible in the investment case, or finance will draw it for you later, and less kindly.

The second caution is about the denominator. Measure cost per completed useful task, not cost per answer. Divide the relevant operating cost by the tasks completed in the same period, and define "completed" tightly enough that retries and abandoned requests can't flatter the number. An assistant can cut the time to draft an answer while adding more time to check it. A cheaper model can throw more exceptions. Only the complete task tells you which option wins.

The cost below the waterline

Business cases tend to show the build: implementation, licenses, the first integration. The costs that keep arriving afterwards are another matter. Support, data upkeep, evaluation, review and supplier commitments can outweigh the build over the life of the service, so estimate when each one starts and how big it gets. Leaving an obligation out of the case doesn't make the capability affordable. It just hands the bill to someone else.

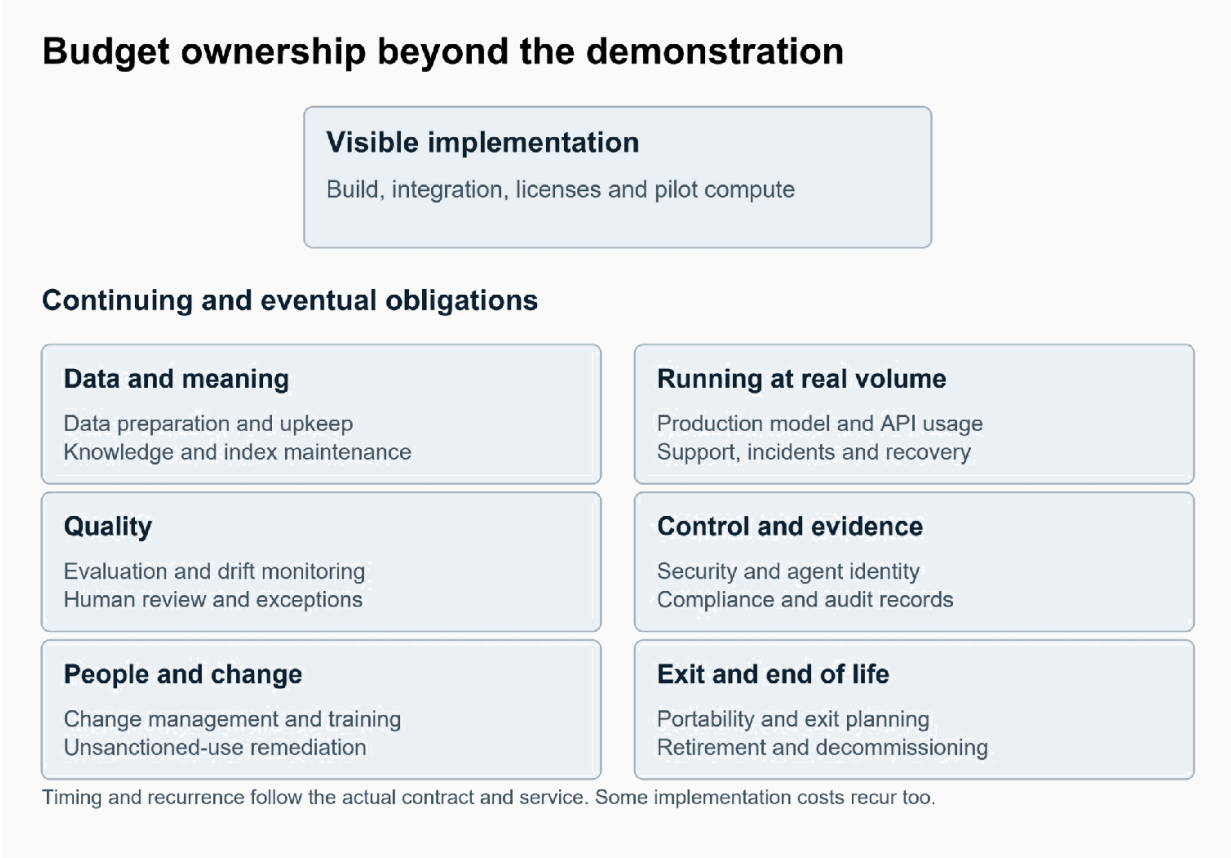


Figure 5. Total cost of ownership beyond the demonstration

Figure 5 groups the costs that continue or show up later. Licenses, vendor fees and some integration costs recur as well. Classify every line by its real contract and service obligation.

AI adds new variable drivers to familiar technology costs. More adoption means more calls, more context, more retrieval and more review; caching, batching, routing, price tiers and commitments pull the other way. Any material change to a model, prompt, source or tool needs an impact assessment and fresh testing. Meanwhile the information has to stay current, permitted and explainable. Estimate all of it from the actual design. Usage cost rarely moves in a straight line with adoption.

Table 4. The cost below the waterline

Cost below the waterline	What it covers	When it starts	Where the book covers it
Data preparation and upkeep	Integrating sources, keeping data products certified and definitions current	Frame and Prove, then ongoing	Chapters 10 to 12
Knowledge maintenance	Re-indexing changed documents; carrying deletions and permission changes to indexes and caches	Before ingestion, then continuous	Chapters 12 and 14
Production usage	Model and API calls, retrieval and compute at real volume, not pilot volume	Prove and release; volume-dependent	Chapter 9
Evaluation and drift monitoring	Test sets, change impact assessment, re-evaluation and monitoring	Prove, then material changes	Chapter 20
Human review and exceptions	Checking and correcting output, handling escalations and the cases the system cannot	Prove and release, then ongoing	Chapters 1 and 3
Security and agent identity	Identities and permissions for agents, enforcement at the data, testing hostile content	Frame and Prove, then ongoing	Chapters 13 and 14
Compliance and audit trail	Request records, retention, reviewer access and model risk documentation	Frame and Prove, then ongoing	Chapter 17
Support and operation	On-call cover, incidents, recovery drills and release management	Before release, then ongoing	Chapter 20
Change management and training	Redesigned roles, hubs, training and measuring adoption	Before release, then refresh and turnover	Chapter 18
Shadow AI cleanup	Finding, replacing or retiring unsanctioned tools, and the data they hold	Discovery, then periodic	Chapter 5
Vendor lock-in and exit	Contract terms, portability and the cost of switching a model or platform	Design, then at each renewal	Chapter 8
Retirement	Decommissioning, archiving records, removing indexes and access	End of life	Chapter 20

Three habits keep the full cost in view.

- Put every category in the business case with an owner. Mark each line as one-time, recurring, transition or exit, and give it a timing. A line marked “unknown” beats a missing line.
- Use the full cost in your unit economics. Cost per completed useful task means something only when the top of the fraction includes everything below the waterline, not just model calls.
- Fund the running when you fund the building. A service that depends on an annual budget nobody owns will decay once the project ends, however good the launch was.

Watch the quiet items too: source loads, transfer, storage, serving compute, synchronization, and the hours people spend checking and correcting output. Zero-copy is not zero-cost. And a benefit claim without an observation window, a baseline and the full operating cost? That’s a hope.

Chapter 6 turns this table into a funding model. Appendix F gives it to you as a worksheet.

Choose the first scope deliberately

You want a candidate with enough value and a workable route through its data, identity and review requirements. Sometimes the most valuable workflow needs unresolved identity matching, documents you can’t reach and autonomous actions, all at once. Don’t take on all of it. Start with a narrower information task inside it.

At the insurer, the first release gathers and cites case context for examiners to read. It doesn’t update the claim. It doesn’t pay it or write to the customer either, because each of those would need its own authority, approvals, failure handling and recovery plan. Starting with reading keeps the consequences small while the team learns. Even a read-only path still needs complete access and disclosure controls.

The distributor’s first release answers order-status questions for service agents, not for customers. The agent stays in the loop, sees the sources and fixes the answer when it’s wrong. That way the company discovers where its data is bad before a customer does.

End the phase with a decision

I call the first phase Frame. It should end in one of three ways: proceed, reshape or stop.

Proceed when the sponsor accepts the baseline, the business owner can change the workflow, and the team can test the constraints that will decide success. Reshape when the workflow or the assumptions need to change. Stop when the value isn’t there, or the path to it isn’t.

Naming all three outcomes up front changes behavior. Nobody has to be the person who killed the pilot, because stopping was always on the menu. A pilot that stopped and taught you something costs far less than one that drifts for months.

What the source papers add

The Connected Enterprise Data & AI Blueprint links the workflow baseline to information, authority and ownership. AI Token Economics goes further on two distinctions: capacity versus planned cash benefit, and allocated versus incremental versus avoidable cost. Read them together before you choose an investment. [2; 16]

Chapter 3 then makes the workflow redesign explicit. You compare the whole process before and after, exceptions and checking included.

Before you fund the first capability

- The workflow is a repeated decision or information task with identifiable users.
- A named business owner can change the workflow and accept measured results.
- Today's work is mapped, including review, waiting, exceptions and errors.
- The baseline covers a representative period, with case mix and seasonality agreed.
- Targets and unacceptable outcomes are written down before any results are seen.
- The economics count review, correction and full operating cost, and keep capacity separate from cash.
- The first scope is the narrowest slice that can prove value, read-only unless an action is essential.
- The phase ends with an explicit proceed, reshape or stop decision.

CHAPTER 2

Owners with real authority

An AI capability needs an owner who can change the work. It also needs other decision holders, whose authority that owner can't borrow. Separate four kinds of decision: business outcome, information meaning, the technical service and risk. Give each a name, a deadline and a route for escalation. A committee can look at the facts together. But nobody on the team should have to guess who can say yes, who can say no, and who acts when the service stops meeting its commitments.

The decision everyone could discuss

A composite scene. For six weeks the claims assistant has waited on one word: open. Claims operations says a claim is open while it still needs an examiner's work. Finance says a claim stays open while a payment or recovery is unsettled, even after the examiner has closed the file. Each is right for its own purpose. Neither will give up its measure.

Every Thursday the product owner brings the disagreement to the working group, and the group asks for more analysis. Engineering produces another reconciliation, this time showing 312 claims that are open under one definition and closed under the other. Next Thursday, the same people restate the same positions. Meanwhile the assistant runs on a temporary rule nobody approved. Ask the room who can actually decide, and every head turns toward someone else.

Another count won't fix this. The missing piece is a decision: which purpose does the first release serve, and who has the authority to approve a definition for that purpose? If the assistant prepares an examiner's worklist, claims operations can own the operational definition. Finance keeps its financial definition, with a documented reconciliation wherever the two meet. Engineering can encode both. A dispute over an enterprise-wide identifier might need a different authority altogether.

Ownership earns its place when it settles that kind of question without pretending one function's legitimate purpose doesn't exist. The Data and AI Operating Model draws the lines between domain authority, product coordination and shared enterprise responsibilities. [5]

A sponsor is not an operating model

The sponsor makes the initiative matter and helps find the money. The process owner decides how the work changes and accepts the business outcome. The product owner sets priorities and manages the capability over its life. And the service owner answers for keeping it running. One person can hold several of these roles. They remain different commitments.

Consider the gaps. A sponsor can fund an assistant with no authority to change the claims procedure. A product owner can prioritize a feature with no authority to grant access to medical information. A technology executive can run a reliable service and still be the wrong person to judge a coverage interpretation. Call one person “accountable for everything” and you blur exactly the lines you need.

Bring the acceptances together

A release that crosses those lines needs every acceptance your institution actually requires. Record what each holder accepted, for which population and which version, plus any condition that has to stay true. One holder may reject a new use while another approves the technical deployment; until they agree, the service runs at the narrower scope. A joint committee minute helps only if it keeps those separate decisions visible.

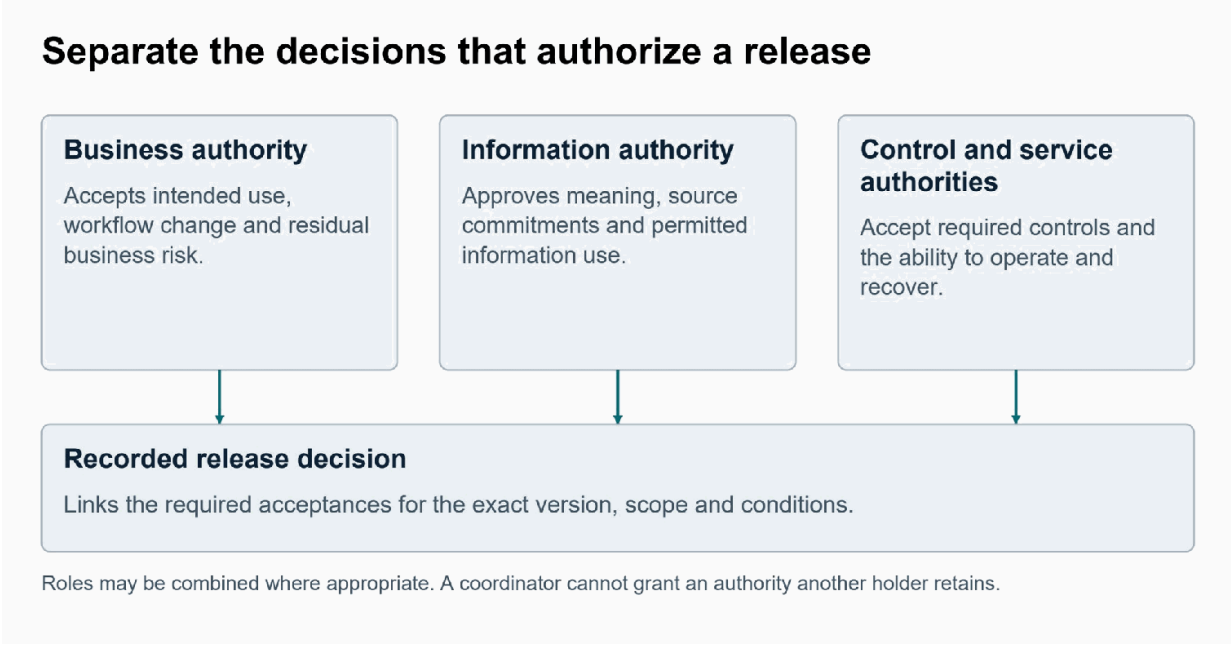


Figure 6. Decision rights that contribute to a release

Start with the decisions the first workflow will really need. Who approves the intended use? Who settles a disputed term? Who owns a wrong source record? Who can approve a

new action? Who accepts a residual risk, and up to what limit? Who can shut off a route at night? Who pays for the work after an incident?

Name a holder and a delegate for each. Write down the scope of their authority and what they need in front of them to decide. A title like CDO or CTO doesn't answer the question, because every institution divides those jobs differently. Escalation should reach someone who can decide this particular dispute. Someone merely higher up the delivery chart won't do.

Risk acceptance has limits too. A business sponsor can't approve an unlawful use. Nor can they waive someone else's legal obligation or stretch a control owner's delegated authority. Where your rules or the law require independent review, the product team has to get it. Clear decision rights simply surface those conditions early.

Build the register around a real request

The claims assistant's first release retrieves permitted case information and the procedure text that applies. It doesn't decide coverage, issue payments or contact customers. That keeps its decision register short enough to actually use.

Table 5. Build the register around a real request

Decision	Accountable decision holder	What they need and where the line sits
Change the examiner's preparation workflow	Claims process owner	Baseline, revised work steps, review capacity and outcome measure
Approve a term for examiner preparation	Claims meaning authority	Purpose-specific definition, effective date and known-answer tests
Maintain procedure applicability	Procedure owner	Approved versions and written applicability rules
Permit access to a claim population	Delegated access authority	Role, purpose, restrictions and demonstrated enforcement
Release the named capability	Authorized release authority	Business, control and operational acceptance for the exact scope
Restrict a failing route	Named incident or service authority	Pre-agreed containment conditions and an escalation record
Fund continuing support	Budget holder	Service workload, recurring cost and capacity commitment

Treat this as one way to allocate the decisions. It isn't an organization chart to copy. Your institution may split release across several authorities; if so, record each approval and name who pulls them together into the final call. Never hide a necessary veto behind a single signature that can't grant it.

Name contributors in the register as well. Finance may challenge the economics. Privacy may review the use of personal data, security sets enforcement requirements, and the procedure owner supplies the history. Their input matters. It doesn't make every contributor jointly accountable for every decision.

Deadlines change what authority means

A name without time to decide just creates a queue. Give routine decisions an expected response time, and say what happens when someone misses it. Match the clock to the consequence. A production incident and an ordinary definition proposal don't belong on the same meeting cycle.

Say the claims team agrees that a minor definition clarification gets a first response within two business days. The clock starts when the owner receives a complete proposal, not when someone first grumbles about the issue in a chat. The proposal names the term, gives examples, lists the affected consumers and states the decision requested. If the owner needs more, the request is on the record. If the deadline passes, the product owner escalates to the named delegate.

Silence still isn't approval. Escalation doesn't license engineering to pick whichever definition is easiest to build. The affected feature stays restricted until someone with authority records a decision. Sometimes a read-only capability can go ahead without the disputed measure. An action that depends on it may have to wait.

Track repeated delays by type of decision. Twenty similar requests to one owner? You need a standing rule with delegated limits. Requests that keep arriving half-prepared point to a broken proposal template. And when the same cross-domain dispute comes back every month, the enterprise boundary itself is unclear. Another weekly meeting fixes none of these.

Chapter 16 turns standing authority into an operating pattern. Automation can assemble the facts, or carry out a routine action someone already authorized. It can't settle a new business dispute by producing an approval record.

Exceptions need an end date

An exception is a decision to operate under stated conditions despite a known gap. It doesn't turn an "unknown" into a "yes". Write down five things: what's missing, why the remaining scope is acceptable, who may accept that exposure, which compensating measures apply, and when the decision ends or comes back for review.

Suppose an approved limited trial shows information from a daily snapshot while the team is still building the live interface. That can be fine for historical preparation, pro-

vided users see the snapshot time and the workflow rules out decisions that need current status. It wouldn't justify showing the same snapshot as live payment information. An exception belongs to a purpose and a population. It doesn't belong to a piece of software alone.

Some gaps can't be carried by an exception at all. Unenforced access, an unknown tenant boundary or an unapproved financial action should stop that path. Your register should distinguish who can accept a limited residual risk from who, if anyone, can override a particular policy. Signing an exception gives the signatory no power they didn't already have.

Revisit exceptions when their assumptions change. A fixed expiry date helps. A source change, a new user population or a bigger transaction limit can invalidate the decision sooner, though, so the service needs a way to notice those changes and restrict the function involved.

Stop authority has to work at 2 a.m.

Most teams can tell you who approves launch. Fewer can tell you who can switch off a tool after launch. Settle that before release.

The first responder should be able to contain a known kind of failure without negotiating a new mandate mid-incident. Restricted documents turning up in answers? They may need to suspend retrieval from that collection. Claim-status source gone stale? They may restrict status answers and leave everything else running. Make containment as wide as the facts justify. No wider.

Pre-agreed containment doesn't remove accountability. Log the trigger, the action, the time, the affected users and who was told. Bringing the service back needs its own proof and its own authorized decision. Note that the person who can stop a route may not be the one who can approve its return.

This makes support realistic. Picture a service owner who answers for an incident but can't see the logs or restrict the tools. That person has the responsibility without the means. Test those rights during the recovery exercise, using the receiving team's own identities.

The lean version

The distributor doesn't need a governance department. Its chief operating officer can own the order-service workflow, and its senior engineer can own the assistant's operation. The credit controller keeps authority over holds. Where the builder would otherwise sign off

on their own work, a finance leader or a suitable outside adviser can challenge a consequential release.

Use a one-page register, and name cover for absences. If the only engineer is out, someone still has to know how to turn the assistant off and send service agents back to the old order system. If no qualified person can independently review a proposed write capability, stay read-only until someone can.

Company size doesn't change what a wrong action costs. It changes how much machinery you can afford. Combine roles where you must. Narrow scope where capacity is thin. And keep the important authorities separate.

A register the team can use

Before the next build increment, take the last three decisions that got stuck and put them in the register. Ask the proposed holders to decide a real example. Then check three things: the delegate can act, the supporting facts can be produced, and escalation reaches someone with the mandate to decide.

Keep the register next to the product contract and the release records. Review it after incidents, after material changes in scope and when people leave. Nobody settles ownership once on a slide. It works only while the named people keep the authority, the time and the information the decisions require.

The test of the register is simple. When the next disagreement comes, the team knows who decides, what that person needs, and what stays restricted until the decision is made.

CHAPTER 3

Redesign the work and the review

The business case rests on the work that changes after the assistant arrives. Map the old process and the new one at the level of a completed case, with review, waiting and correction included. Decide which steps go away. Decide which stay as fixed rules, where the model helps and where a person keeps judgment. Then treat review as a real, funded job with the information, time and authority it needs. Bolting an assistant onto every old step and asking people to check its work twice? That's an expensive way to prove adoption.

A new front end on an old process

A composite scene. A large firm set out to redesign how it staffed its work, and the ambition was right. It wanted to match people to work faster, see capacity and financial performance across the firm, and give senior leaders a simpler way to find the people they needed and manage the work they owned. The centerpiece was an AI front end. Leaders could ask, in plain language, who was available and who had the right experience. The firm's data and AI teams built the data products and AI capabilities behind it.

Behind the front end, no process changed. Many teams still ran staffing on spreadsheets and local habits. A natural-language layer can only be as current and complete as the sources underneath it. Here the most important source, the record of what each person could actually do, was the weakest.

Think about what a staffing decision needs to know. Many firms assess skills when people join, and those results ought to flow into the record used for staffing. This record leaned on assignment history instead: which engagements a person had worked on. For a generalist, that's a poor proxy. Working on a bank engagement doesn't mean you can build a credit model. Sitting next to a data team doesn't make you a data engineer. Skills change with every assignment and every course, and unless someone updates and validates them, the system matches people on where they've been rather than on what they can do.

The life of a consulting engagement shows where that data has to be captured and kept current. At each stage below, the first sentence says what the staffing decision needs; the second names the failure that tends to sit there.

- Recruit and onboard. The decision needs proficiency results, prior experience, languages and certifications. The failure: test results stay in recruiting and never reach the staffing record.
- Build the profile. It needs a skills record in one shared taxonomy, with a level and the proof behind it. The failure: self-reported skills in free text, described differently by every group.
- Request staff. It needs the role, skills, level, dates and location the engagement requires. The failure: requests sent by email and spreadsheet in local shorthand no system can match against.
- Match and assign. It needs current availability and validated skills, alongside development goals. The failure: matching on assignment history, personal networks and availability kept in private spreadsheets.
- Deliver the work. It needs a record of the skills actually used, at what level and on what kind of problem. The failure: only the engagement code is captured, so the work leaves no trace in the skills record.
- Review and develop. It needs feedback from people who saw the work, plus courses and certifications mapped to skills. The failure: reviews and training records live in separate processes that never touch the skills record.
- Roll off and redeploy. It needs an accurate end date and a skills record refreshed by the assignment just finished. The failure: end dates slip, and the person rolls into the next request with the profile they joined with.

Put those failures together. They describe a missing data product with an owner: a skills record captured at hiring, updated after assignments and courses, validated by people who saw the work, and kept current by someone who answers for it. Your firm's failures may differ. Look at your own definitions, input quality and incentives before assuming these apply. The general lesson holds, though. A better interface depends on the information and responsibilities behind it.

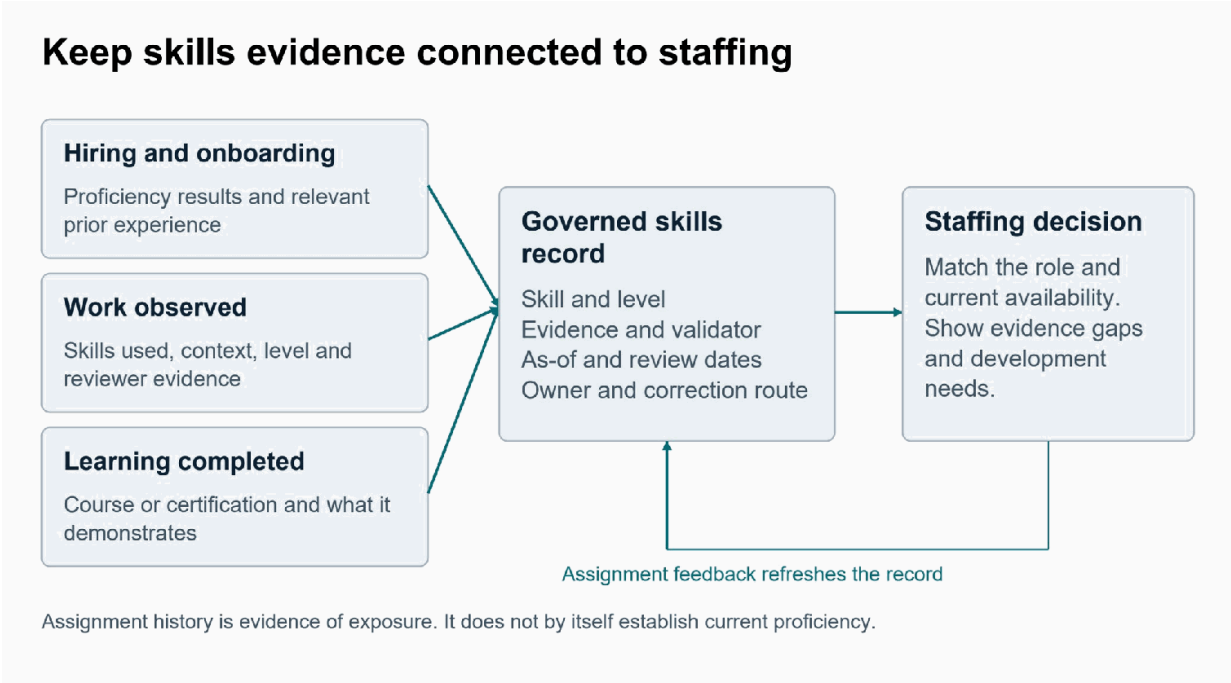


Figure 7. Skills evidence maintained across the work lifecycle

If you build this, give the skills product a correction route as well as an update feed. A course completed, an assignment attended and a proficiency demonstrated are different observations. Keep each one with its date. Say who can validate a level, and let people challenge a wrong record. Limit sensitive staffing information to those who need it. A matching system should show its uncertainty; it shouldn't turn missing information into a permanent verdict on someone's ability.

A skills profile should support a staffing judgment that someone can review. Decide which observations are relevant, give employees a way to correct the record, and test whether matching quietly excludes people through missing or proxy data. Don't turn conversational tone, inferred personality or unrelated personal details into a hidden suitability score. Employment-law and privacy review depend on the actual use and jurisdiction. A "confirm" button for a human doesn't settle them.

The redesign was meant to cover the money as well as the people. Senior leaders needed to see how their teams were doing: time charged, work in progress, billing and collections against each engagement's budget. Here work in progress, or WIP, means work performed and recorded but not yet billed, valued on finance's approved basis. An unusually high balance can mean delayed billing, or work that won't be recovered. A low balance needs explaining against the work plan and billing cycle before anyone concludes the team is underused.

WIP, an invoice and a receivable are three different states. Billing unbilled work turns WIP into an invoice without adding a dollar of cash. Cash arrives later, on collection. So an

early billing alert needs a record of what happened next. Did it fix a data error, trigger a bill, bring in a collection or avoid a loss? Count every alert as financial value and you overstate the benefit.

The redesign replaced the reports and dashboards leaders had relied on. The replacements didn't work either, mostly because of the data underneath. A report showing last month's WIP tells a leader what went wrong after the chance to fix it has passed. A report built on poor data tells them the wrong thing at any speed. Many leaders went back to the spreadsheets they trusted. The financial side has its own lifecycle, and it depends on the same records as staffing.

- Set up the engagement. It needs fees, budget, billing terms and the team that owns the work. The failure: finance and staffing set up engagements and teams differently, so the same team appears twice or not at all.
- Record the work. It needs time and expenses charged to the right engagement every week. The failure: late or miscoded time, which understates WIP until month end.
- Track WIP. It needs unbilled work measured against budget and billing schedule, with its age. The failure: WIP visible only in a month-end report.
- Bill and collect. It needs invoices raised when milestones or dates arrive, and collections followed up. The failure: billing that depends on someone remembering.
- Report performance. It needs team results that agree with staffing and with each other. The failure: dashboards that disagree, so leaders keep their own numbers.

The design went further than dashboards. The firm built agents that acted on the numbers, within limits the senior leaders set. They watched WIP on every engagement against thresholds agreed by the engagement leader and finance. When a threshold was crossed, they did the routine work.

- When WIP was too high, say unbilled work above an agreed share of budget or older than an agreed number of days, the agent alerted the engagement leader and the finance contact. It showed which time and milestones were driving the balance, drafted a billing request for approval and flagged work that had run past the agreed fee so the leader could raise it with the client.
- When WIP was too low, say time charged well below plan, it first looked for missing or miscoded time. A low balance on its own can't tell incomplete records from recent billing or a real shortfall in delivered work. If a shortfall was confirmed, it alerted the leader and the staffing team and proposed options: redeploy people, pull planned work forward or revisit the plan.
- In every case the agent read, alerted and drafted. People decided. It couldn't issue an invoice, change a budget or move a person on its own. Every alert recorded the data it

used and when that data was current, and every threshold had an owner and a version. Chapter 13 sets out the contract an agent like this needs before it's allowed to do more.

The agents worked as designed. The processes around them didn't change, so the agents never became the way the work was run. Their alerts went to people whose jobs nobody had redesigned to act on them. And the agents read the same late time entries and mismatched team structures as the reports they replaced. Send an alert built on bad data into an unchanged process and you get a faster wrong answer. An agent is only as good as the records it reads and the process that acts on what it says.

Meaning was the other half of the problem. Finance owned the business meaning behind the financial numbers, as it should; it was the authority on WIP, on a billable hour and on a team's financial results. But finance never published those definitions into the shared semantic layer. Publishing them would have changed who controlled how the numbers were defined and changed, and nothing in the program made that worth it to the people who held them. So staffing and finance kept separate versions of terms that the front end and the agents needed to reconcile.

The incentives were wrong here too. A function rewarded for controlling its definitions, and not for making them reusable, has every reason to hold them back. Chapters 2 and 12 describe the arrangement that would have worked. Finance stays the authority for financial meaning. It publishes each definition into the shared layer with an owner, a version and an effective date, and approves every change through that contract. Owning a meaning doesn't require owning every copy of it. In fact, publishing through a managed layer gives the owner more control over how the meaning gets used. In this scene, expectations and rewards didn't move with the new interface. People could keep using the old route, and nothing in how they were measured asked them to work differently. That operating problem compounded the data, integration and definition problems, and treating the interface as the whole solution left all of them in place.

A better interface helps when the information and workflow behind it support the decision. That combination is what I would change first.

Context for the staffing and finance questions

If you build this, start by recognizing an incomplete question. "Why is my team's WIP below plan?" leaves a lot unsaid. The application has to work out who "my" refers to, which team and engagements are meant, which reporting period applies and which finance-approved definition of WIP to use. A person's home department may not tell you which engagements they're responsible for right now. And a finance partner and an engagement leader may be entitled to different views of the same work.

The record below is design guidance; it doesn't describe the earlier implementation. Trusted identity maps the caller to an employee reference, and authoritative organization and assignment records establish the relationships. A workflow selection or a clarifying question fixes the task and period. The policy service and the source interfaces decide which financial or staffing facts this caller may receive.

Table 6. Context for an illustrative WIP question

Context element	How this request establishes and uses it
Caller and service	Verified sign-in maps to an employee reference; the application also identifies its own workload. The caller is distinct from employees discussed in the answer.
Purpose	Explain below-plan WIP for the selected team and period; propose checks. No authority to issue an invoice, change a budget or reassign a person.
Team and engagements	Resolve the team and current engagement assignments through approved records. Ask which team if several authorized scopes fit.
Period and meaning	Confirm the reporting period, finance-approved WIP basis, plan version, currency of time entries and billing information.
Permitted information	Apply current rights to each engagement and employee record. Team totals don't authorize individual pay, performance or personal details.
Sources and gaps	Link time-entry completeness, billing events and plan comparisons to their sources and dates. Missing information stays visible as a gap.
Communication	Use the requested detail and finance terminology. An impatient tone can change presentation, never the scope or the financial conclusion.
Result and limits	Explain supported causes, separate them from hypotheses and suggest review. If team, period or required records can't be resolved, clarify or hold back that conclusion.

A staffing question changes the record. "Who can join this engagement next month?" needs the role requirements, the candidates the caller may see, validated proficiency, current availability and the proposed dates. It doesn't inherit the WIP caller's financial scope. Nor does it treat past project attendance as proven skill. A person's development needs can matter where the approved staffing purpose allows; unrelated personal information doesn't. Now suppose an employee asks "What is my WIP?" The application first asks whether they mean recorded time, unbilled work on their engagements or something else. It shouldn't silently adopt a manager's team-wide reading. Keep requester, subject and purpose separate, and a familiar phrase can't turn into an accidental cross-department query.

Follow the case through both processes

Back to the claims composite. Today the examiner opens several systems, checks who is who on the file, finds the procedure and writes a preparation note. The proposed assistant pulls those sources together. Its first design, however, leaves the old process untouched.

The examiner opens the assistant and reads its summary. Then she visits every original source, repeats the preparation and records the same note as before.

Some checking is necessary, especially while you're still evaluating. But name what each check is for. Repeating the whole task forever eats the time the assistant was supposed to save. Removing a needed control because the model sounds fluent creates a different failure.

Before choosing either path, watch a sample of real work: ordinary cases, hard cases and interruptions. Separate the examiner's active effort from time spent waiting. A lookup may take two minutes of work and three hours of waiting for access. An assistant that removes typing won't touch the wait. Record errors and the corrections that follow, including work other teams do.

The Adoption by Design paper puts process, roles, incentives and capacity into one design. That's the practical starting point here. You're redesigning the whole workflow, not the assistant's reply. [3]

Separate different kinds of work

Some steps need exact execution of an approved rule. A claim identifier has to be valid. A calculation has to use its approved definition, and an entitlement has to be enforced. Use ordinary deterministic code wherever you can write those rules down and test them. A language model shouldn't invent a different eligibility test because a retrieved document sounds persuasive.

Other steps benefit from a model's help. It can draft a summary, suggest relevant passages or write an internal note within a narrow task. Those outputs need testing suited to how people will use them. Retrieval should narrow the material to permitted, applicable sources before the model sees anything.

Some decisions stay with people. They involve delegated judgment, uncertainty with consequences or a policy line. In this book's first claims release, the examiner keeps the case decision; the assistant supplies context and never determines coverage. That's the scope chosen for the example. It isn't a rule for every claims process.

One task can hold all three. The service checks the claim identifier by rule. It retrieves and summarizes the procedure. Then it leaves an ambiguous case to the examiner. Label the whole workflow "AI" or "human" and you hide the boundaries you most need to design.

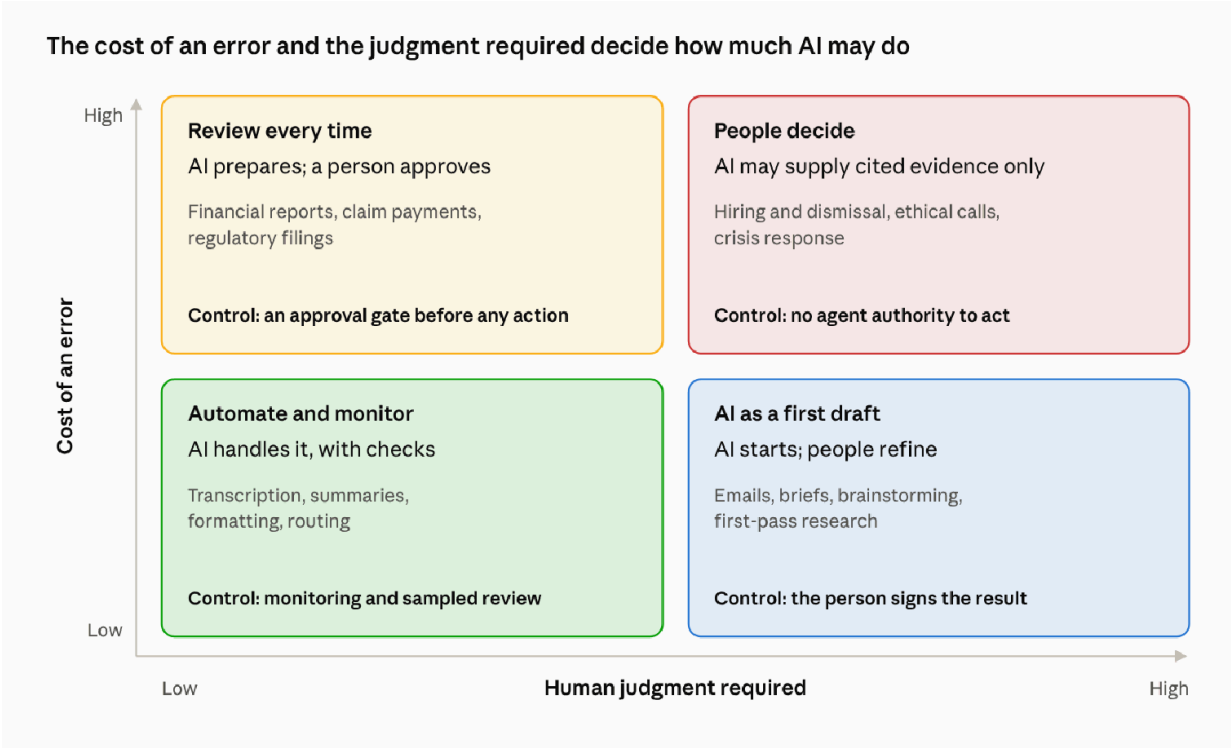


Figure 8. Choosing task authority from consequences and judgment

Use Figure 8 to start the conversation about authority, not as a classification of jobs. A summary, an email or a transcript can carry real consequences in the wrong setting. Look at the specific use and the people affected before choosing the level of oversight.

Make the new process explicit

Here's the comparison the claims team draws up. It's a design to test. It doesn't show savings anyone has achieved.

Table 7. Make the new process explicit

Activity	Before	Proposed first release	What must still be shown
Identify the claim and examiner's authority	Several screens and separate access checks	Approved claim tool enforces identity and scope	Wrong identifiers and unauthorized access are rejected
Assemble case facts	Manual copying and reconciliation	Cited facts from named interfaces, with as-of times	Facts are complete enough for the task and conflicts stay visible
Select applicable procedure	Search and interpret document metadata	Retrieval applies the approved applicability rule	Wrong-date, wrong-product and superseded material is handled correctly
Prepare an internal working summary	Examiner writes from sources	Assistant proposes a summary	Review plus correction takes less effort at acceptable quality
Decide the case	Examiner judgment	Examiner judgment	The interface implies no authority it hasn't been given

Activity	Before	Proposed first release	What must still be shown
Handle incomplete information	Informal investigation	Defined incomplete state and review route	Users can carry on safely without guessing

The change that matters is specific. The examiner no longer copies an approved source field by hand just to prove the copying happened. Instead, the service supplies the provenance, and the team tests the mapping. The examiner still inspects real ambiguity and anything that bears on the decision. Which checks can change depends on the control objective, the obligations that apply and what testing shows.

Before release, take this map to the leader of the affected function and ask two questions. Which old requirement goes away or changes? Who can authorize that? If every old step has to stay, rewrite the business case around the real operating model. There may still be value in consistency, accessibility or fewer errors. Just don't claim time savings the process won't allow.

Human review needs capacity

"A person reviews it" means little until you know what arrives, how long review takes and what the reviewer can do about it. Define what the reviewer sees, the expertise required, the options available, the escalation route and the expected turnaround. Show the actual source passage, the version that applies and any material uncertainty. A confident summary alone isn't enough.

Run the numbers on an assumed example. Say 1,000 items arrive each business day and each takes three minutes to review. The queue needs 3,000 minutes, or 50 hours, of review a day. If each reviewer has five hours a day for this after their other duties, you need ten reviewers' worth of time, and that's before cover for absence, swings in demand and harder exceptions. It doesn't become eight because the launch plan lists eight names.

An average isn't a service guarantee either. A spike in arrivals, a run of hard cases or a source outage can grow the queue even when the monthly average looks affordable. Look at the spread of review times and the arrival pattern. Load-test the human process alongside the technical one.

When capacity falls short, you have four honest options. Narrow the scope. Improve how the material is presented. Remove avoidable review through approved controls. Or fund the work. A backlog growing quietly is not a control. Neither is lowering the review standard in a busy week without its owner's say-so.

Give the reviewer a real decision

A reviewer needs to be able to reject, correct, ask for more information and stop an unsafe path. If the only button says Accept, you've built confirmation, not judgment. And if fixing the output takes longer than starting over, reviewers will either skip the tool or wave weak material through to clear the queue.

Capture the reason for an override in a form you can use: a short structured reason, with optional detail. Distinguish a wrong source fact from a poor summary, a wrong applicability rule, a missing permission and a legitimate professional disagreement. Each has a different fix and a different owner.

Test review with cases whose right answers someone has settled independently, in an approved exercise environment. Include plausible wrong answers, missing information and answers that are genuinely right. Do reviewers catch and resolve the defects that matter? How long does it take? Measure both. You're after a better review design, not a culture that rewards suspicion of everything.

Automation bias means leaning too heavily on an automated recommendation. Fluency, workload, time pressure and screen design all push on it, and no number of required clicks proves sound judgment. Research on cognitive forcing functions found benefits and usability costs in a specific experiment. [67] That supports testing your review design. It doesn't prove one intervention works in every job.

Engineering against automation bias and fatigue

A review step earns its place when it lowers the chance of a consequential error reaching the work. Build it around the fact or decision in dispute. For a procedure recommendation, show the case's relevant date, the period the procedure applies to and the passage behind the recommendation. Have the reviewer resolve any mismatch before approving. Opening a source is a click. Understanding and reconciling it is the work.

Use friction where it pays. Before a consequential action, you can require the reviewer to check the subject, amount, date or destination against the authoritative source and record how they resolved a conflict.

Requiring two citation clicks on every answer, by contrast, tends to become ritual. Two passages may repeat the same source, and some tasks have only one authoritative record. Test whether the interaction you choose improves detection without swamping the queue.

Measure review quality with controlled exercises. Use synthetic people, cases and defects in an isolated evaluation queue, or a separately authorized test partition with no customer effects, no real cross-tenant disclosure and no downstream writes. Keep exercise results out of live business records and routine performance reporting. Never seed wrong claim

facts or fake identities into ordinary customer work just to test vigilance. And be wary of borrowed rates. A figure like 0.5 percent has no universal basis.

Size the sample from the decision you need to make: how severe the defects are, how often they occur, the variety of cases and how precisely you need to judge the change. Measure five things: detection of planted defects, false alarms on correct cases, time to resolve, quality of escalations and reviewer workload. State the sample size and its limits. Then use the results to improve the screen, the staffing and the training. Approval speed and click counts tell you nothing about the quality of supervision.

Make uncertainty something a reviewer can act on. Model weights are learned parameters, not a confidence meter for each answer. Token probabilities, retrieval similarity and classifier scores answer different questions; they aren't interchangeable measures of correctness. Validate any decision threshold on representative held-out cases. Check calibration wherever you claim a probability, and test again after material changes. Missing authority, incompatible dates or an unresolved identity trigger the contract's response, however high the model's score.

When sources conflict, show the competing permitted facts, their sources and dates, and the decision required. A side-by-side reconciliation view often helps; test it with real reviewers. Offer an "incomplete" or escalation path, so nobody has to choose between accepting a polished narrative and abandoning the task. Make every source link open the version that was reviewed, within the reviewer's own access.

Finally, fund attention. Watch queue age, case difficulty, interruptions and the effort of correction. Give reviewers the authority to stop an action and someone qualified to escalate to. If review capacity can't carry the proposed scope, shrink the scope. You want reliable judgment at a pace people can sustain. The highest possible approval rate is the wrong target.

Design for the wrong procedure

Go back to a failure in the claims example. The assistant finds a valid procedure, but the version it picked doesn't apply to this case. The summary describes the document accurately. It's still the wrong answer for the task.

In the book's teaching scenario, an approved rule uses the policy date to choose the procedure. Real claims processes may use other dates and conditions. Either way, the review screen has to show the rule and the date clearly enough for the examiner to spot a mismatch. A citation alone won't do it.

Test the whole response: what the service retrieves, what the model says, what the reviewer sees and how the case gets recorded. The failure isn't closed just because the re-

trieval engineer fixed a filter. Add a regression case. Check earlier uses that may have been affected. Have the process owner accept the corrected behavior.

This is also why exceptions belong in the baseline. If one case in ten needs a difficult applicability review, an average preparation time hides it, and the business case misleads. The assistant may handle routine cases well and need a separate route for the rest.

Keep a fallback that works

The manual process has to stay usable for whoever is meant to fall back on it. That doesn't mean everyone repeats it on every case. It means the access, instructions and capacity exist, and the team can tell which work needs them when the assistant is down or restricted.

Test the switch. Can an examiner get back into the source systems? Does the queue keep enough state that an unfinished case isn't lost? Who takes the extra workload, and how do customers or dependent teams hear about delays? If the assistant proposed an action and the outcome is uncertain, a manual operator mustn't create a second effect while trying to help. Chapter 13 develops that point.

Keep the skills that supervision and recovery depend on. You can cut repetitive preparation and still practice source investigation, exception handling and incident response. Where you let a skill fade, write down what depends on it and what happens without it. A runbook doesn't replace experience overnight.

Redesign incentives with the job

An examiner measured only on throughput may rush a hard review. An engineer rewarded only for deployment frequency may let support slide. A manager praised for adoption may discourage honest reports of workarounds. The measures around a tool shape how people use it.

The staffing front end at the start of this chapter shows why making a tool available isn't enough. Make the reason to use the new workflow visible in the work itself: better preparation, fewer repeat lookups, clearer decisions. Training can't forever make up for a process whose burdens outweigh its benefits.

Review outcome, quality, burden and use together. Ask people what they stopped doing. Ask what new effort appeared, and which cases still push them back to the old route. Put training and evaluation time in the work plan; don't treat participation as free labor squeezed in around a full existing workload.

For the distributor, all of this fits on one page. Service agents keep customer communication. The assistant pulls together approved order and shipment facts. The company drops duplicate copying where its control owners agree, leaves credit decisions with the con-

troller and records open promises explicitly. The process is smaller. The before-and-after comparison still matters.

By the end of this chapter you should have three things: a revised process map, a calculation of review capacity and a fallback you've tested. Together they show whether the assistant changes useful work or just adds a screen. Chapter 18 then tests whether people use the new process under ordinary workload, and whether the improvement survives the trial.

PART 2

Build the team that runs it

CHAPTER 4

Choose teams that can keep the service running

Pick a team arrangement that can carry the capability through ordinary change and ordinary failure. Keep domain knowledge close to the business. Make shared services dependable, and put scarce experts where they're truly needed. Stable capacity matters more than a fashionable org chart. A team that builds the assistant and then disperses hasn't settled who will maintain its definitions, watch its behavior or bring it back when it breaks.

When the project closes

A composite scene. The claims assistant goes live on a Friday, and the project team celebrates. By the end of the month, the specialists borrowed from five functions are back in their old jobs. The claims platform group has inherited the application. The procedure owner assumes technology will keep the index current, while technology assumes claims will notice when a procedure goes out of date. Nobody has funded either job. Nobody has noticed that nobody has.

Three weeks later, claims publishes a revised procedure. The assistant keeps serving the old one. An examiner spots it and raises a ticket, and application support restarts the service and finds the infrastructure healthy. The service is still broken, though: the path that was supposed to carry content changes didn't. Support can't decide which procedure applies, or whether cases prepared during the gap need another look. The people who can decide sit outside the support arrangement. So who owns the assistant now?

You don't necessarily solve this by putting everyone in one department. You solve it with a lasting team that both delivers and operates the service, with its dependencies written down. The operating-model paper separates domain delivery, platform services, temporary enablement and specialist capability. [5] Those categories tell you what the team needs without dictating identical staffing everywhere.

Allocate the work before choosing labels

A central group makes sense when skills are scarce, common needs dominate or the domains can't yet run their own services. It builds deep expertise and keeps engineering

consistent. Its weakness is distance. A central backlog turns into a queue, and people who don't know the work start guessing what terms mean.

Embedded teams put engineers next to decisions and users. When the shared foundation is weak, they can also duplicate tools and controls. Federation keeps responsibility in the domains under common services and standards. It works only if the domains have the authority, skills and lasting capacity to carry it. Hand a business area ownership without those resources, and you've created a dependency dressed up as autonomy.

Most institutions run several arrangements at once. A central group may own enterprise identifiers. Claims may own a preparation workflow, and finance a reconciliation product, with a specialist integration service supporting both. What matters is that each product and service has a usable boundary, an owner and a funded home once it's running.

So size the smallest workable team from the first workflow. List what it takes to change that workflow safely: process knowledge, information modeling, engineering, integration, evaluation, security, review and operation. Decide which skills have to be on hand all the time, and which you can bring in for specific decisions. Eight responsibilities don't mean eight full-time jobs.

The jobs themselves shift as automation grows. Data engineers spend more time maintaining contracts and keeping consumers' data reliable. Integration specialists protect state and identity across boundaries. Stewards spend more time deciding meaning and less copying metadata. Chapter 16 sets these shifts beside the changes in software engineering and data science. Treat them as an allocation to design; I'm not predicting that whole professions vanish.

Keep one backlog

The claims capability needs one ranked list of its work. That list covers user improvements, data defects, permissions, reliability, evaluation upkeep and eventual retirement. Screen changes are only part of it.

The product owner sets priorities within agreed limits. A mandatory fix, a source being retired or a failed control can jump the queue. When that happens, record why. Don't treat unplanned support as proof the team missed its feature plan. You can't maintain a service and schedule as though maintenance doesn't exist.

Professional disciplines supply hiring standards, career development and engineering practice. They should make people better at their work. They shouldn't create a second backlog. When the engineering manager hands out one set of tasks and the product owner another, the team spends its time negotiating.

Business participation works the same way. An examiner who drops into the occasional demo can't supply everything design, testing and adoption need. Give named experts protected time and a direct line to the process owner. Rotation broadens knowledge. Keep enough continuity, though, that someone can explain why the team made its earlier decisions.

Make shared services easy to consume

The platform group should offer what domains need through supported interfaces: identity, deployment, monitoring, an approved runtime and other repeated needs. A domain team should know how to use each service, what it promises and where responsibility changes hands.

Say the platform owns a retrieval runtime, and claims owns the approved procedure collection plus the rules for which procedure applies. The contract between them has to answer five questions. Who notices a source change? Who updates the metadata? Who checks the index? Who restricts use when the content goes stale, and who tells the examiners? "Platform handles search" answers none of those.

Test the boundary with an ordinary change. Have the claims team publish a new approved procedure version through the supported path. Can they do it without the person who designed the platform? Does the platform show which version is live and whether it's healthy? If every content update needs an undocumented conversation, the service isn't consumable yet.

Team Topologies distinguishes stream-aligned, platform, enabling and complicated-subsystem teams, each with defined ways of interacting. [25] I'm applying those ideas to data and AI responsibilities here. I'm not reproducing the framework, and nobody needs to create four teams.

A group that never became a team

A composite scene. A public institution brought together an integrated analysis group. The idea was sound. Draw people from business areas across the institution, and AI, analytics and data could be shared instead of rebuilt in every area. A central data team built the platforms that sourced the data the group was meant to share.

The group formed. The tribes never disbanded. People kept the priorities and habits of the areas they came from, nothing of substance crossed those lines, and the effort failed. The platforms could make the data available. They couldn't make the group share it.

Looking back, the missing pieces are the ones this chapter describes. A shared name isn't shared work. The group needed one backlog, data and analysis that another area could

use under a contract, and a real change in who set priorities and how contributions were recognized. Everyone expected collaboration to happen because people sat together. Real collaboration leaves behind something the next team can use without asking.

Collaboration should leave an interface behind

Working closely together is valuable while two groups discover a new boundary. Claims and platform may need to work side by side on carrying identity through the system, or on revoking a source. Give that collaboration an end product: a tested contract, examples, support arrangements and a way for each side to change independently.

Once the interface settles, using the service should replace negotiating it. Where a team lacks a skill, temporary coaching should leave it able to do the task. A team that can't function without its coach hasn't been enabled.

A specialist team makes sense when concentrating expertise improves reliability, or lifts a burden no single domain could reasonably carry. Complex entity resolution might qualify. So might a critical shared authorization service. The specialist service still needs a contract and an operating owner. Scarce expertise is a reason to write the boundary down more carefully. It's no excuse to skip it.

Look at the requests that keep crossing between teams. They may point to a missing service, a vague contract or a team asked to understand too many unrelated systems. Measure the waiting and rework they cause before assuming more coordination meetings will help.

Protect the ability to operate

Building and running compete for the same people. Hold back capacity for incidents, dependency changes, documentation, evaluation and onboarding users. The right share depends on how mature the service is and how much load it carries; no single percentage makes a team sustainable.

Use recent numbers. Count support demand, repeat defects, maintenance forced by suppliers and changes to the content. If the team can't keep up with ordinary upkeep, cut feature commitments or add capacity. A growing list of deferred maintenance is a future obligation. It isn't free productivity.

Be clear about support hours and business impact. An internal assistant used during one region's working day may not need the same cover as a customer-facing decision service. Match the commitment to how people actually rely on it. If users keep working overnight, an overnight failure is their problem, whatever the project plan calls the tool.

Absence cover matters at every size. At minimum, containment and fallback have to work without the one person who understands the deployment. If the service can't support that, limit its use and make the dependency visible to its owner.

Test a transfer before declaring it

The receiving team should demonstrate its part of the operating model early. Ask it to add and remove a user, investigate a request, assess a procedure change, restrict a route and recover from a simulated failure. It should use its own access and tools. The builders can watch. They shouldn't quietly do the hard steps.

The exercise usually turns up useful gaps. A support engineer can restart the application but can't find the policy decision. A data owner knows the right definition but not how to publish a new version. An incident lead carries the responsibility and has no authority to switch off a tool. Assign each gap to someone before the next release; don't carry it as an informal favor.

Chapter 20 makes transfer a formal release decision. The implication for team design comes earlier. Put the people who will run the service into the work while there's still time to learn and to change it. A training session at the end can't substitute for being there throughout.

A small team with believable boundaries

The distributor may have one senior engineer, one analyst and a part-time business owner. Managed services supply most of the platform, and a specialist reviews permissions and deployment. For a limited internal order-status assistant, that can be enough, provided the team can genuinely meet the support and record-keeping commitments.

It stops being enough when the company adds several client tenants, round-the-clock use or authority to release credit holds. Each adds a different kind of burden. The next investment might be independent review, stronger isolation or operational cover. A better model may not be on the list at all.

A private-equity owner should be able to see these dependencies in diligence. Which services would keep running if the lead engineer left? Which business definitions are written down? Which suppliers can change the operating cost? Which approvals still rely on one person's access? A small company is easier to run, and easier to sell, when those answers don't live only in its founders' heads.

This chapter's output is a map of services and teams: the domain backlog, the shared services, temporary support, specialist dependencies and the people who will keep running

it. Judge the map with a real change and a recovery exercise. The chart helps only if it helps the people on it carry those responsibilities.

CHAPTER 5

A platform that removes repeated work

A platform earns its place when teams can build and run useful capabilities with less repeated effort. Start from needs that genuinely recur. Give teams a supported path with the common controls already built in, and measure the work it takes away. Count the work it adds, too: learning, integration, waiting, migration and support. A pile of centrally purchased tools is not yet a platform.

Start with the second customer

A composite scene. Underwriting has seen the claims assistant and wants one. The claims team agrees and hands over its repository. Two weeks later, the underwriting engineers are still reading claims-specific scripts and asking which settings matter. Then they discover that the retrieval filters assume claims entitlements underwriting doesn't use. Identity, deployment and record-keeping carried over. The procedure rules, source permissions and business authority didn't. They couldn't.

The second customer shows you what was actually reusable. A template full of claims-specific scripts and tribal knowledge may be a fine first implementation. It isn't a general service.

The opposite mistake costs just as much. Turn every claims rule into a configurable enterprise framework, and you build complexity the second customer never asked for.

Pull out the stable part. A shared retrieval service can accept a defined content contract, apply supported access filters, keep source references and report its own health. Each domain supplies the approved content, the metadata that says what applies when, and a person who maintains it. With that split, the platform standardizes the engineering while business meaning stays with the people who own it.

The Federated Data Platform paper recommends exactly this incremental approach. Each bounded delivery should leave behind products and services others can use. The paper is also clear that reuse has ongoing costs, and that real consumption has to justify them. [7]

A service needs a consumer contract

Pick the first platform services from demand you've actually seen repeat. Common candidates include carrying identity through the system, publishing contracts, deployment checks, request records and operating telemetry. None of this means building a complete platform before the first workflow ships.

Agentic delivery adds a new recurring need. Teams want a supported way to assemble current system context, give an agent a limited set of tools and keep a record of what it proposed or changed. Treat those as platform services only once a second team can use them through a documented contract. A shared agent runtime without managed context, scoped identities and independent checks just centralizes a new kind of repeated risk.

For each service, write down who uses it, the task it supports, the interface, prerequisites, limits, support, how it's charged and how it changes. Include a working example. Does a team need three specialists to interpret the documentation before its first successful call? Then ask what the service has quietly pushed onto its users.

Table 8. A service needs a consumer contract

Service	What the consumer gets	What the platform keeps doing
Product discovery	Find a relevant supported interface and its owner	Current metadata, search behavior and visible lifecycle status
Scoped access	Get and revoke authorized access	Identity integration, decision records and tested enforcement
Delivery template	Deploy a reviewed change reproducibly	Supported versions, tests, migration and troubleshooting
Evaluation service	Run the relevant test pack against a named release	Reproducible runs, protected datasets and retained results
Request records	Investigate an observed result	Correlation, access, retention and retrieval after failure
Service telemetry	Detect a breach and reach its owner	Defined measures, reliable routing and useful alerts

Consumers have obligations too. A template can't supply a missing data owner. It can't validate a business definition without approved examples, either. Spell out those prerequisites and help teams meet them. And never let a deployment succeed in a way that implies someone checked the prerequisites when nobody did.

Put the controls in the path people use

A supported delivery path can carry typed contracts, identity, policy checks, evaluation, monitoring and rollback. The payoff is repeatability. Domains stop reinventing the same mechanisms. Once deployed, though, the controls still have to work.

Take a product manifest. It can declare the owner, freshness commitment, allowed consumers, dependencies and version. Delivery tooling can check the required fields, publish metadata, set up tests and wire alerts. That's useful automation. But a completed file proves nothing about whether a source is fresh or a permission is enforced.

Connect each declaration to behavior. A freshness breach should trigger the serving path's agreed response. A revoked entitlement should stop authorizing access within its stated window. A source schema change should stop an untested mapping from quietly going live. The catalog reports the condition; the runtime applies the restriction.

This is where platform engineering makes governance practical. The standard path produces the record as the work happens. Teams don't have to reconstruct it for a final presentation. Reviewers can inspect a specific request and its policy result while the design is still cheap to change.

Hold generated code and handwritten code to the same secure-development lifecycle. Chapter 15 maps the NIST SSDF v1.1 practices to requirements, toolchains, review, testing and release integrity. [52] The platform should make those controls repeatable, while keeping business acceptance and AI evaluation as separate, visible decisions.

Self-service has prerequisites and limits

Self-service means a qualified team can do an approved task without a fresh negotiation every time. It doesn't mean any employee can provision any data, identity or tool. The service needs an entry rule, and a way to tell supported requests from exceptions.

An engineer might spin up a sandbox from an approved template with synthetic data. Touching real claims information takes additional conditions on purpose, identity and controls. Promoting to a service people rely on adds operating commitments, evaluation and a release record. The interface should show those differences. Labeling every environment "Ready" hides them.

Automate decisions whose conditions are clear and already authorized. Leave business judgment and unusual risk decisions with the right holder. A workflow can gather the facts and route an approval; it can't approve itself. Where a person still has to review, show the queue and how fast it moves.

Measure the whole journey

Track the elapsed time from an accepted need to working, authorized use. Split it into finding the service, meeting prerequisites, getting access, integrating, testing and reaching supported operation. A fast deployment step can sit alongside weeks of delay everywhere else.

Pair speed with burden and reliability. How many manual interventions did the first deployment need? How much specialist time did it eat? How many defects followed? Compare like with like, and state the period you observed. A new team on an unfamiliar source isn't comparable with an experienced team repeating a standard deployment.

Count supported reuse, not repository copies. A product is reused when another consumer relies on its approved interface and the owner keeps that commitment. A forked template creates an unsupported version whose upkeep stays hidden. Track versions, exceptions and migrations so the platform team can see that load.

Put cost in the same review. A common service can cut duplication and still raise shared operating cost. That trade can be worth making. Show both sides. Don't let a conceptual reuse curve stand in for savings; finance needs to see work avoided, commitments removed or capacity redirected to a named use.

Treat exceptions as product feedback

The supported path won't fit every need. Give teams a way to propose an exception: the unmet requirement, the alternatives considered, the extra controls, an owner and an expiry or review trigger. Separate a genuine capability gap from a preference for a familiar tool.

When several domains want the same exception, revisit the platform backlog. A service that can't handle a common source may be shutting out valuable work. And if exceptions keep sidestepping identity or record-keeping, you have a control problem, not healthy variety.

Exceptions come with running costs. Who supports the alternative? How does it get security updates? Can you investigate incidents in it the usual way? How will its users eventually move off it? Whoever approves the exception should see that cost before the alternative becomes permanent.

Keep the supported path adaptable through versioned interfaces. An ever-growing pile of undocumented switches does the opposite. Retire old patterns with dates and migration help. A platform that hoards every past exception eventually hands its complexity back to every consumer.

Prioritize from repeated effort

Build a backlog that ties each service change to a consumer problem. For each candidate, note the teams affected, how often the problem occurs, the effort and risk involved, and the conditions under which teams will actually use the improvement. Include the cost of running the service and keeping it compatible.

Suppose three domains have each built the same request-correlation mechanism. Support it once and you remove repeated engineering and make investigations consistent. That's a far stronger case than building a general agent framework because every vendor has one. Start with the smallest supported component that fixes the real need.

Security and resilience can justify work before many consumers show up. When they do, say so and name the control objective. Don't pretend the investment pays back through reuse. The platform has several legitimate purposes, and honesty about which one you're funding helps leadership choose between them.

Review the first consumer after delivery and the second after reuse. What got easier? What still needed personal help? Which responsibility stayed fuzzy? Let the answers shape the next version.

Keep the small version small

The distributor may need a repository, a deployment pipeline, identity integration, an approved route to its models and a simple operating dashboard. Managed services cover much of that. Its platform contract might be a few documented conventions and a tested template, maintained by the same engineers who build the assistant.

Don't import an enterprise organization into a company that can't sustain it. And don't confuse managed infrastructure with handing off accountability. Someone still owns provider access, invoices, limits, recovery and the data permissions the service exposes.

This chapter's output is a platform backlog ranked by measured friction for real consumers, plus a contract for the first shared service. The next team should be able to show the improvement using its own people and access. That demonstration, and the service's operating record, will tell you more than any count of tools bought or templates published.

CHAPTER 6

Fund the running and the people

Fund the ongoing service before anyone relies on it. Keep the cost of building separate from the cost of running. Keep the cost assigned to a workflow separate from the cost that would actually disappear if you stopped it. And make someone responsible for the actions that turn freed-up time into business benefit. A cheap model call can't make up for an unfunded review queue, an unsupported data product or a benefit nobody has been asked to deliver.

The budget that ends before the obligation

A composite scene. The claims project's budget looks tidy: engineering, licenses and pilot usage, all approved. Now look at what's missing. The examiners will absorb review. Technology will absorb support. Because the procedure library already has an owner, everyone assumes keeping the index current is free. Evaluation appears once, as a launch task.

Six months after launch, the project code closes. The examiners are still reviewing, now with no time set aside for it. The model provider announces a version change, and nobody has budget to re-test. When the procedure owner asks who pays for re-indexing, the budget's answer is nobody. The project only looked affordable because it had pushed its costs onto other teams. Its obligation started the day people began relying on the assistant. Source corrections, staff turnover, model changes and incidents don't wait for the next project.

A sustainable funding request names the service owner, the workload, the support arrangement and the recurring costs. It separates work absorbed by existing staff from genuinely new spending. And it asks whoever owns that existing capacity to sign up for the commitment. Staff time has an opportunity cost even when no new invoice arrives.

AI Token Economics organizes this conversation around accepted work, benefit realization and three distinct views of cost. Its companion workbook is an illustrative operating forecast. It doesn't prove delivered cash returns, and it isn't a full investment cash-flow model. [16]

Define accepted work before you price it

For the claims assistant, one useful completion is an accepted preparation task for an eligible case, meeting the agreed quality criteria. A model response, a session and a finished case are three different units. One accepted task may take several model calls. Failed attempts burn resources without adding a completion.

Record attempts and useful completions separately. A workflow with cheap responses can still be expensive if people keep retrying or giving up. Shifts in case mix change acceptance too, so keep the denominator stable enough for comparisons to mean something.

Build the forecast from its drivers: eligible volume, adoption, attempts, model calls, input and output volume, retrieval, review, exceptions and recurring services. Swap in measured pilot values as soon as you have them. Use provider rates from your actual contract, model, processing method and effective date. A token isn't a fixed number of words. A caching discount doesn't apply to every request.

Reconcile the forecast to real invoices and internal cost records. Where you can't trace a cost directly, state the allocation rule. A precise-looking total that leaves half the operating estate out of scope makes a weak basis for an investment decision.

Three views of cost answer three questions

Fully allocated cost tells you how the portfolio's costs are spread. Incremental cost tells you what a change adds. Avoidable cost tells you what you could release by stopping, and when. You need all three. None substitutes for another.

Here's an assumed annual example. A workflow carries $80,000 of shared platform allocation and $20,000 of running cost that would disappear if it stopped. Its fully allocated cost is $100,000. Say it produces $70,000 of validated annual benefit on a consistent valuation basis. Its benefit-cost ratio on that basis is 0.70.

Stopping it doesn't save $100,000. If the $80,000 platform commitment stays and stopping wipes out the $70,000 benefit, the recurring position gets $50,000 worse: $20,000 saved, $70,000 lost. Exit costs and service obligations could make it worse still. All of this assumes the benefit and the avoidability are real. That's exactly what finance has to test.

The reverse happens too. A workflow can look great on a light allocation while expansion triggers new staff, licenses or review capacity. The incremental decision has to include that step. Never scale on the assumption that the next ten thousand tasks will cost what the last ten thousand did.

These are management views; they don't replace accounting policy. Finance should set how costs and benefits get reported, keep that policy consistent and explain the sensitivi-

ties that could swing the decision.

Name the action that realizes the benefit

The same discipline applies in an investment review or in private-equity diligence. A forecast of hours released should name the action that turns those hours into more work done, spending avoided or another measured outcome. Does the benefit depend on one particular employee? On a supplier change nobody has signed? On work another team will have to absorb? Check, and record that dependency against the benefit's owner. A sound case can include service or risk improvements without giving them an invented cash value.

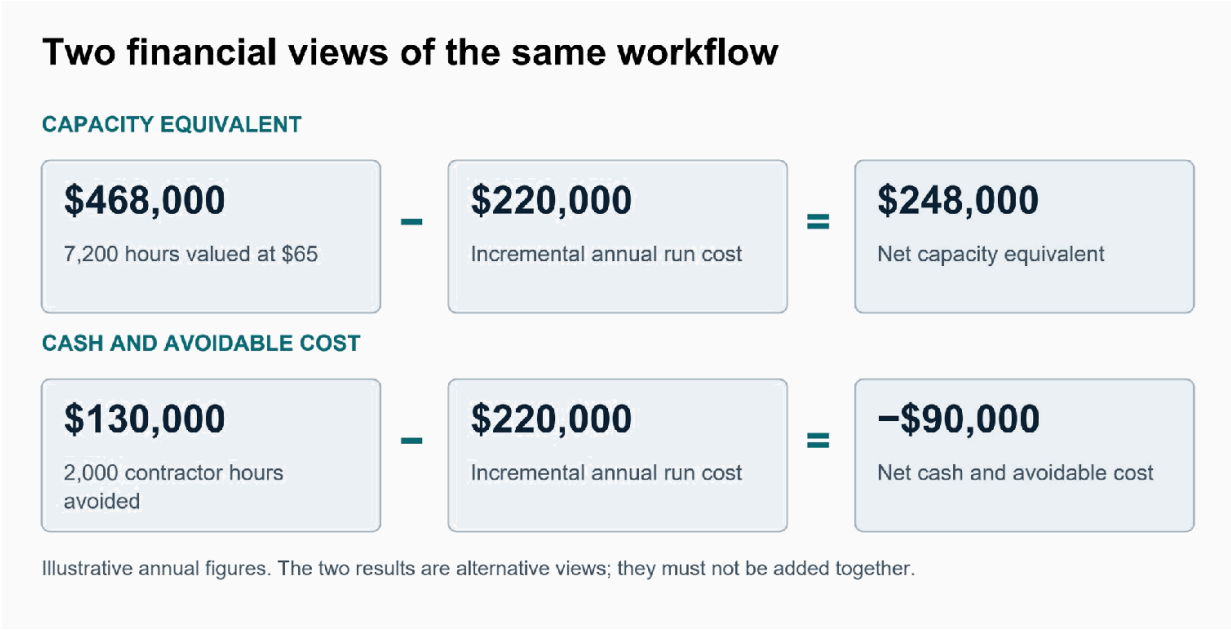


Figure 9. Capacity and cash value in the Chapter 1 annual example

Capacity needs a destination

Chapter 1 estimates net time released after review and rework. That capacity can be valuable without cutting spending right away. It might absorb more demand, improve service or clear work someone had deferred. Call it what it is.

A cash or avoided-cost claim needs a specific action and a date. The business owner might cut overtime, cancel planned contractor capacity or change hiring. Finance checks the basis and whether the action really happens. A general pledge to "redeploy the time" is not a plan for realizing cash.

For each benefit, record the baseline, the work affected, the acceptance criteria, the measurement window, the realization action, the owner and the start date. Keep the forecast, the pilot measurement and the realized result separate. When the same people support

several workflows, make sure nobody claims their saved hours more than once, or beyond the hours they actually work.

Count review effort once, and say how. If net time saved already subtracts existing reviewers' effort, don't subtract it again from capacity value. If you need new paid reviewers, show that spending and explain how it relates to the time calculation. The worksheet in Appendix F makes the treatment explicit.

Measure total active effort on comparable eligible cases. That includes failed attempts, non-use where it matters, rework and the manual route. A task abandoned after five minutes of checking still used five minutes. If you apply an acceptance rate, keep the effort and cost of the rejected work in the numbers. Multiply only the successful savings by an acceptance rate and you hide what failure costs. Then separate active labor from elapsed waiting before you put a value on either.

Lower risk and better service can justify investment in their own right. Measure them directly where you can. Don't force a speculative dollar figure onto every benefit just to improve a ratio.

Spend on premium service where it pays

Sooner or later a leadership team asks for an "Apple call center" everywhere. They mean instant answers, expert attention, round-the-clock availability and a polished screen for every request. Treat the phrase as an aspiration. It isn't a claim about how Apple actually runs its operations. Translate it into measurable service requirements and a costed operating design. Then ask which customer moments, business decisions and failure consequences really justify the extra spend.

Premium service may be essential where it sets the firm apart, protects an outcome that matters or meets an obligation. Elsewhere, a dependable standard service, a scheduled response or a simpler interface will do. Adequate security, privacy, accessibility and legal compliance aren't luxury tiers; differentiate the experience above them. Google's SRE guidance makes the same point about reliability. It costs resources and opportunity, so set it from the service's business needs. [64]

The comparison below is entirely illustrative. It treats benefits as incremental cash contribution or avoidable spending, not as the labor-equivalent capacity value from Chapter 1. The targeted option spends more where the assumed benefit rises. Premium-everywhere gains a little more benefit at a much larger cost. The numbers show a method. They don't estimate any company, industry or service level.

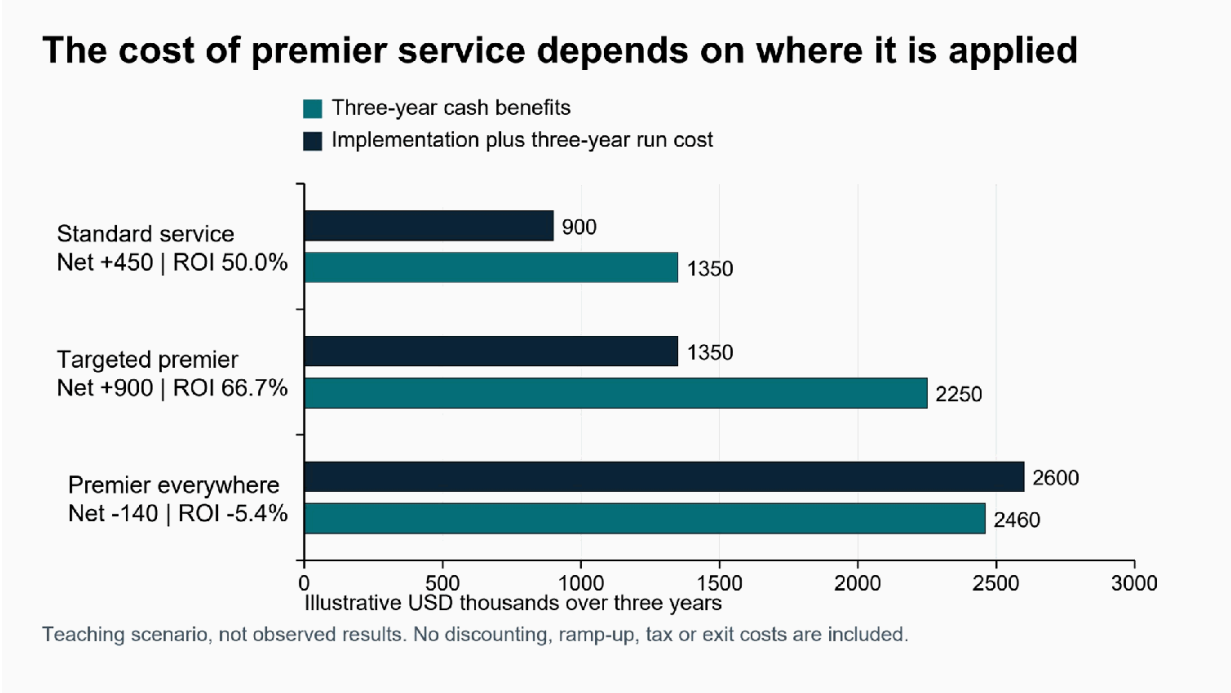


Figure 10. Illustrative cash economics of three service designs

Table 9. Illustrative investment comparison in USD thousands

Service design	Implementation	Annual run	Annual benefit	Three-year ROI
Standard service	300	200	450	50.0%
Targeted premier	450	300	750	66.7%
Premier everywhere	800	600	820	−5.4%

Three-year net benefit is three years of assumed benefit minus implementation and three years of running cost. The simple three-year ROI divides that net benefit by the same total cost. The table leaves out tax, financing, discounting, ramp-up, residual value and exit costs. A real decision has to add those where they matter. It also has to test adoption, realization, demand, how long the benefit lasts and how sensitive the result is to service cost.

Look at the upgrade on its own. Going from targeted premier to premier everywhere adds $70,000 a year of benefit, $300,000 a year of running cost and $350,000 of implementation. It loses money, even though its gross benefit is the highest on the table. A mandatory service obligation might still require it. If so, record that reason. Don’t manufacture a positive ROI.

Transform material work without chasing complexity

Modernization and transformation answer different questions. Replacing an unsupported component may be necessary for continuity, security or a future capability, even if it

saves little now. Replacing a small, cheap process with a big platform can raise its allocated and incremental cost. Neither outcome is automatic. Say which problem the work solves, and show new costs, displaced commitments and downstream benefits on the same basis.

Low cost and high complexity both make poor guides to value. Complexity raises delivery cost and risk. An inexpensive component can still unlock a valuable workflow. BCG's cost-transformation work emphasizes high-impact workflows, mature use cases and staged reinvestment, and its operating analysis emphasizes process redesign and the trade-offs of implementation. [65; 66] Both argue for deliberate selection and sequencing.

Go after material, addressable cost, revenue, delay or risk, with an owner who can change the work and realize the benefit. An expensive process can turn out to be mostly unavoidable obligations with little to automate. A simple, high-volume process can be worth a lot. A small enabling change can open up a valuable workflow downstream. Look at those relationships before ranking anything by project size, technical difficulty or the price of today's component.

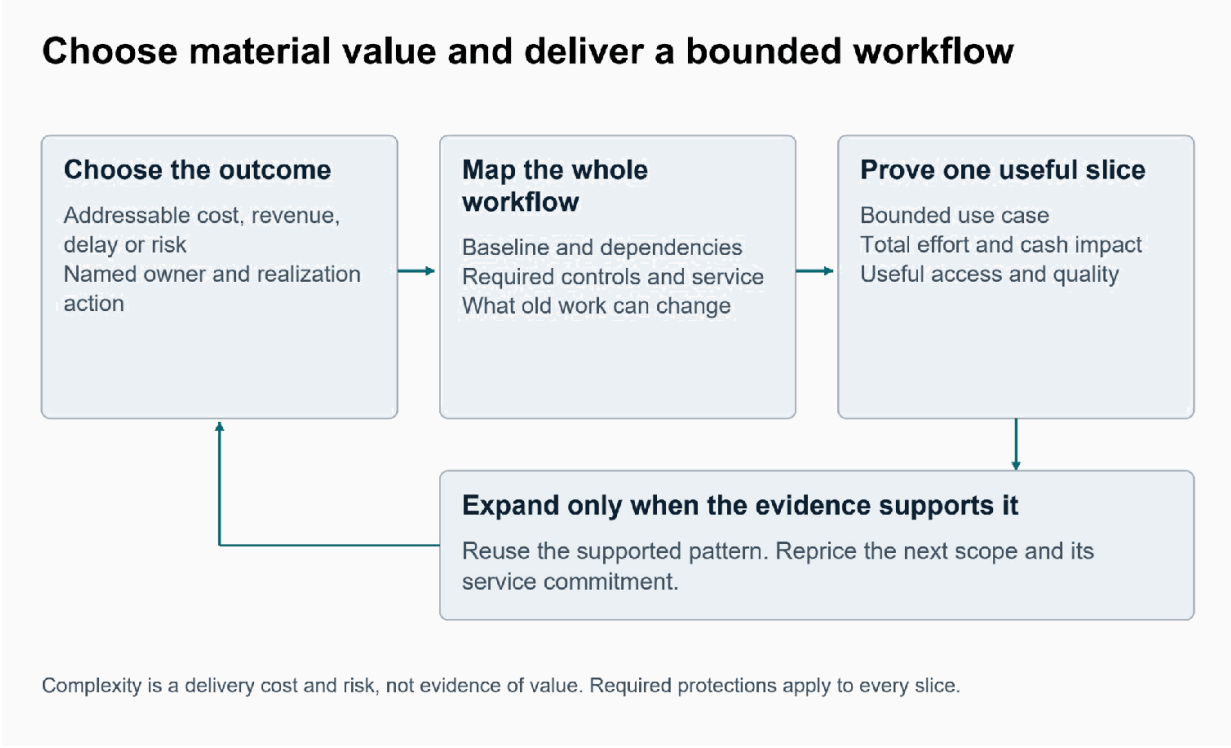


Figure 11. Choose material value and prove a bounded workflow

Design the end-to-end destination. Then deliver one bounded use case inside it. For claims, that might mean preparation before internal notes, and notes before any decision workflow. Baseline the slice you're actually changing. Remove or change old steps only with authority, and measure total effort and outcomes. Expand when the results support

the next commitment. Done well, use case by use case builds one coherent workflow and reusable services. Done badly, it leaves a scatter of disconnected pilots whose costs pile up.

Budget the obligations below the model call

The twelve-line ledger from Chapter 1 groups the continuing responsibilities that budgets tend to miss: data preparation, knowledge maintenance, production usage, support, evaluation, review, security, record-keeping, change, cleaning up unsanctioned tools, exit and retirement. Appendix F keeps those categories and records who supplies each piece of work.

Costs behave differently. Some are fixed, some move with use, and some jump in steps. A gateway subscription stays flat within a tier. Review effort tracks eligible cases. A new region may need a second support arrangement. And a retained copy of a source brings storage, rights and deletion costs that have nothing to do with model usage.

Show the timing. Implementation, transition, running old and new in parallel, and eventual exit can all land outside the year your operating forecast covers. A funding decision needs those cash flows and, where it applies, your organization's approved discounting or investment method. A benefit-cost ratio isn't net present value, internal rate of return or demonstrated cash ROI.

Ask procurement which commitments can be reversed. Minimum volumes, annual licenses and supplier exit terms decide whether the avoidable view is believable. Don't assume prices will fall forever, or that an introductory discount will last. Test flat and rising rates against plausible adoption and review loads.

Fund people as a continuing service

Stable capacity can serve a business-owned backlog without writing a blank check. Agree what the team is funded to run, which changes fit inside that envelope and which need a new investment decision. Revisit the envelope when demand, criticality or supplier commitments change.

The service needs time for updating evaluations, absorbing source changes, following up incidents and training users. That doesn't mean innovation has slowed. It's the cost of keeping a capability the business chose to depend on. When recurring demand outruns capacity, leadership has three choices: cut scope, improve the design or fund the gap.

Support and independent challenge need resources too. A risk reviewer who shows up only at launch may not be available to assess a later material change. Agree a review route

and expected turnaround. A small business can buy specialist review for defined needs instead of hiring a function. The budget should still include it.

Include the cost of the stewardship automation itself: model and tool usage, metadata connectors, evaluation, reviewer time and incident recovery. Track total effort per accepted change. A shorter ticket queue isn't a saving if the same work has moved into unmeasured review or failed automated repairs. Chapter 16 gives the operating measures.

At the distributor, one engineer may carry several of these obligations. Say how many hours are really available, and who covers absences. If the company can't afford to support a write service with real consequences, keep the assistant in its narrower information role. An unfunded responsibility is a reason to change scope. It's no reason to call the feature self-managing.

Put cost into the release decision

Judge a model or retrieval change on the same useful work as the current release. Measure accepted completions, review, latency, attempts and total cost. A more capable model can cost more per call and less per accepted task. A cheaper one can create enough correction work to cost more overall.

Set spending limits with defined behavior. When the service hits the budget or call limit, it might defer eligible work, return a narrower result or send the user to the manual path. Record the unmet demand and the effort it caused. Otherwise the apparent saving may just push spending or work outside what you measure.

Freeze the approved budget for comparison. Update the forecast as you learn, and reconcile actuals. Explain variance through volume, price, usage per task, acceptance, review and recurring commitments. A rising bill isn't waste if accepted work and value rose with it. A flat bill isn't success if the work stopped being useful.

Make the next funding decision concrete

At an agreed review, look at outcome, quality, support and economics together. Expansion needs a believable incremental benefit and the controls to carry the bigger scope. Redesign may be right when the value is real but review or running cost is too high. Retirement needs an avoidable-cost and transition analysis. A disappointing allocation ratio alone won't do.

You should leave with an operating budget and a benefits contract, backed by a downside case. The budget names each obligation, its owner and its funding source. The benefits contract says what will change in the business and who will make it happen. Together they turn a project estimate into a commitment the organization can keep.

PART 3

One system across AI, data and security

CHAPTER 7

Trace the four-part system

An architecture should explain how a permitted business request turns into a supported result. I use the Four-Part System to assign the jobs. The brain assembles and checks the request's context, settles meaning and picks the route to the information. The nervous system carries approved operations. The body supplies delivery and operation, and the immune system keeps it secure. A control plane ties together their versions, decisions and records. Then you follow a real request and test the points where those jobs meet.

A responsibility model, not a shopping list

The four parts are a way of explaining the system. They don't require four products, four teams or a fixed order of network calls. One platform can cover several responsibilities. One responsibility can span several services. Provisioning usually happens before a request arrives, while enforcement can happen at several points as it runs. [9]

Keep the biological names only while they clarify the engineering. The brain isn't a business authority. And calling something an immune system doesn't make it adapt safely on its own. People approve definitions and policies. Software carries out limited behavior. Operators investigate failures and change the system through controlled processes.

Figure 12 shows AI, data and security as disciplines you design together. The four-part view shows the architectural jobs inside that same capability. The two views should agree, but you can't swap their categories. The Reference Implementation paper shows one particular combination of products; the Four-Part System keeps the jobs independent of any combination. [10]

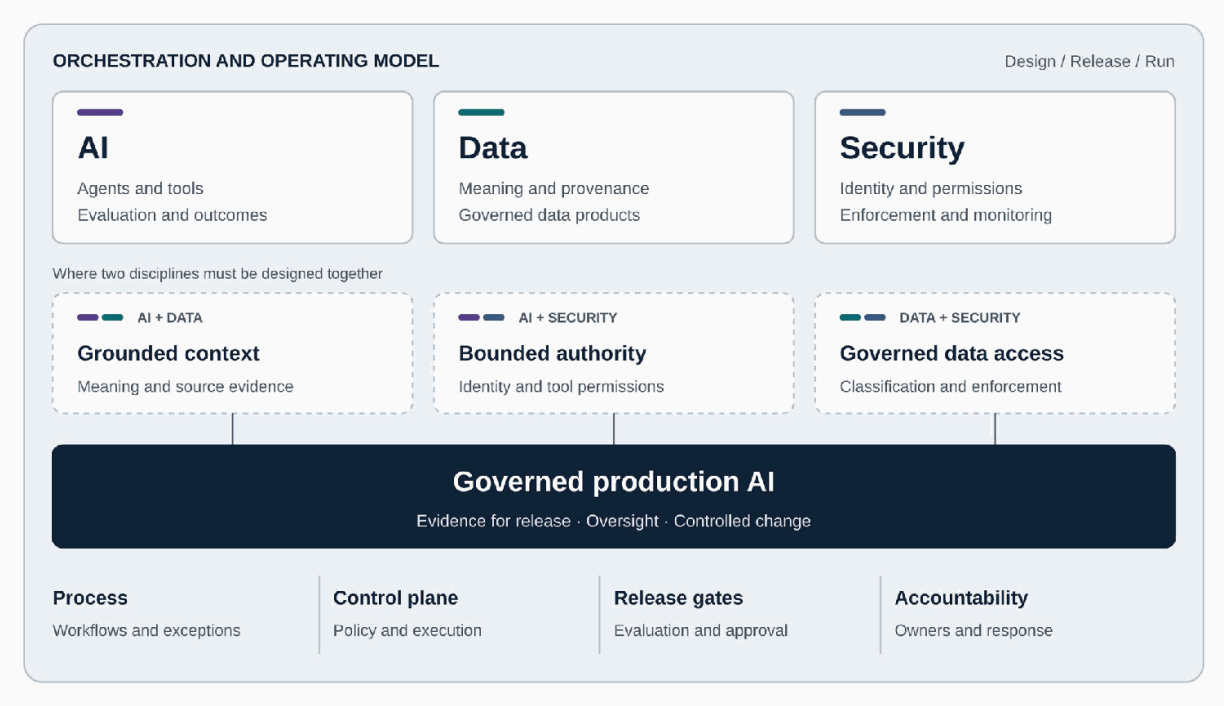


Figure 12. AI, data and security designed together, inside one operating model

Table 10. The four responsibilities and what they supply

Responsibility	What it supplies	What you can check in the claims workflow
Brain	Context assembly, approved entities and relationships, metrics, applicable knowledge and routing	Verified requester and subject; purpose, scope, dates, source provenance and known-answer tests
Nervous system	Versioned read tools, commands and events	Typed interfaces, limited scope, identities and observable outcomes
Body	Environments, deployment, workflow, support and lifecycle	Reviewed release, operating limits, runbook and receiving-team exercise
Immune system	Classification, policy, enforcement, detection and response	Allowed and denied paths, revocation, tenant tests and incident handling
Connecting control plane	Authoritative records of versions, decisions and condition	Linked approvals, release identity, dependency state and request records

The control plane connects the four. You don't buy it as a fifth capability. It can be spread across services, provided everyone knows which record is authoritative and how they stay in sync. It coordinates metadata, policy and operating records. The serving services enforce the decisions.

The same control plane can feed a system-understanding service for delivery work. That service pulls together the relevant code and dependency graph, data and semantic contracts, policy and security bindings, runtime observations and accepted decisions. Each comes with its source, version, freshness and access boundary. The result is task-specific

context that someone can reproduce and challenge. It isn't a permanent model memory, and it isn't an unrestricted copy of the enterprise.

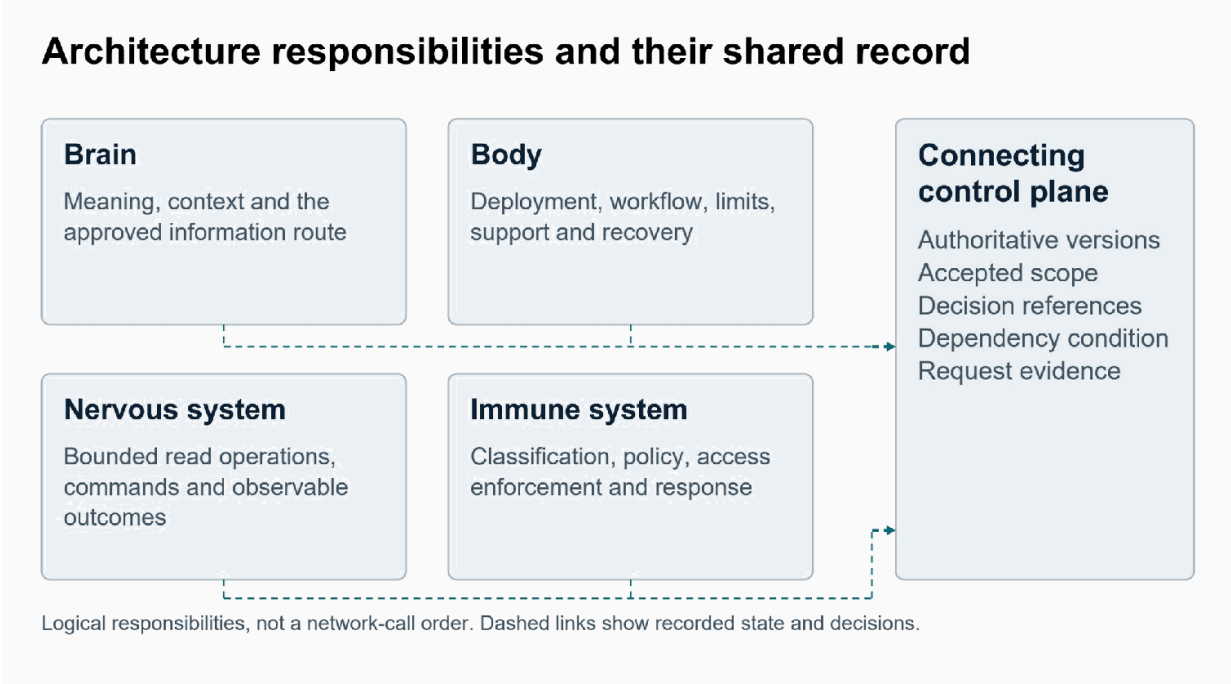


Figure 13. Four architectural responsibilities connected by authoritative records

Use the view to test a change. Labeling a diagram is the wrong use. Say someone withdraws a procedure. Who records that at the source? Which retrieval route stops serving it? What's the operating response? And what shows when each consumer received the change? A shared control plane is believable when all four responsibilities agree on which version is in effect. It doesn't have to be one product or one database.

Federation works the same way. Domains own meaning, pipelines and products. Shared services provide identity, policy, catalog, delivery, evaluation and observability once. That's a split of responsibility; it doesn't require every domain to run its own stack. A centrally operated platform can carry several domains' products, as long as each domain controls its own roadmap and definitions. Shared semantics connect domain models through approved identifiers and mappings. Nobody has to adopt one universal ontology first. [7]

Establish context before composing the answer

The claims examiner asks which parties are connected to a claim, what its status is and which procedure applies. The system authenticates the request and holds it to the examiner's permitted scope. It identifies the approved task. It doesn't turn her wording into an open-ended query.

The brain picks the relevant meaning. Claimant, insured and policyholder are different roles. The procedure has an applicability rule. Status has a freshness requirement. If the system can't resolve the identifier or context, the response should say so or ask. A plausible guessed match isn't a resolution.

Within the brain I distinguish five conceptual jobs: physical information, ontology metadata, tested projections, semantic models and orchestration. They don't need five separately deployed systems. For a first task, a small approved dictionary and a few tested queries can be enough. A richer ontology pays off once relationships and reuse justify maintaining it. A knowledge graph is optional, worth its upkeep only where relationship reasoning earns it.

How the brain gets its context

Assembling context is an application function inside the brain's orchestration job. You can build it in your existing backend; no new platform required. It uses the nervous system's controlled interfaces, the body's runtime and integration services and the immune system's identity and policy controls. The control plane links its definitions, versions and records. Those are collaborating responsibilities. They don't add a fifth part.

The brain here means managed semantic services plus orchestration, which is much broader than the language model. The model doesn't have to discover the employee directory or read a whole profile to understand a request. The application works out what's needed. Then it passes a limited view through the model request and the permitted tool results.

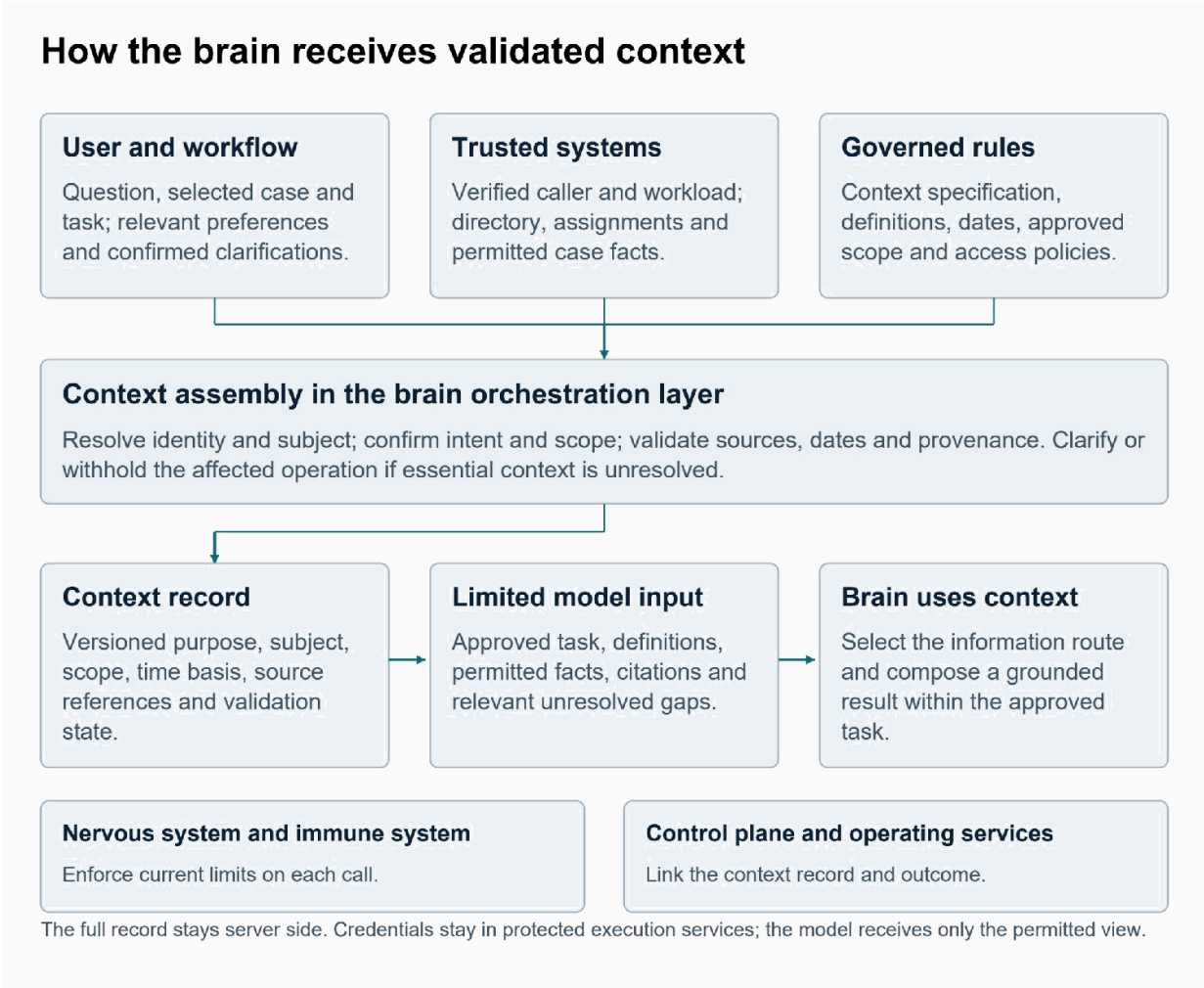


Figure 14. How the brain receives validated context

Build the context record before relying on the result

Establish the caller and the workload. The entry point validates sign-in through your identity design and resolves the caller to the internal reference the task needs. Directory and assignment services supply current organizational relationships where they matter. A typed employee ID is something to validate. It proves neither identity nor permission. A public general-information question may not need an employee identity at all. A case-specific question needs whatever identity and scope its approved workflow requires.

Resolve the purpose and the subject. The workflow supplies the approved task and the selected case, engagement or employee reference. The wording, or the model, can suggest an intent. The system checks that suggestion against the allowed tasks and clarifies it when it's ambiguous or consequential. Keep the person asking separate from the person being asked about. A role someone claims in a chat message can't replace a verified assignment.

Fetch the permitted facts and their meaning. Protected interfaces fetch the organization and case attributes the task needs, under current policy. The context specification lists required fields, authoritative sources, applicable definitions, dates and rules for resolving conflicts. Source facts carry provenance and freshness. User statements and model suggestions stay distinguishable from verified attributes. And a document or an earlier conversation can't promote itself into a trusted role or policy.

Validate the record and give it a version. Check required fields, allowed values, how the subject resolved, source authority, the time basis and consistency with the accepted task. Record whether the request is ready, needs clarification or can't proceed within scope. Clarification can supply a missing period or pick a team. It can't overrule a denial. When a critical dependency stays unresolved, hold back the affected answer or action; a defined partial response is allowed only where the task contract permits it.

"Ready" means the specified checks passed for this task at the recorded time. Nothing more. Required-field and schema checks establish structure. Source reconciliation, entity resolution and business rules answer other questions. None of them proves every source fact is true, or that a user's stated purpose is genuine. Keep material uncertainty visible, and clarify or restrict when it affects the permitted result. A model's self-reported confidence is neither an authorization decision nor a calibrated probability.

Table 11. Where the context record comes from

Field and source	How it's validated and used
Subject and workload, from the identity service	Verify authentication, tenant and acting service. Resolve an employee ID only where needed, and pass the model an internal reference instead.
Function and legal entity, from HR or the directory	Use authoritative attributes with timestamps. Department informs interpretation; it doesn't grant access.
Task and target, from the user and the workflow	Confirm the claim, invoice or reporting scope. Distinguish self-service from acting for someone else. What a user asserts is not proof of entitlement.
Meaning, from product and semantic services	Bind product, metric, ontology mapping and source versions; record grain, dates and freshness requirements.
Authority, from the policy service	Evaluate current entitlements, purpose and delegation. Record the decision reference, scope and expiry, and recheck at sensitive tool calls.
Conversation, user-provided or inferred	Treat tone and preferences as hints about how to respond. They can't establish identity, eligibility, urgency or permission.

Give the model a limited view. Build the model request from approved application instructions, the validated task context and the permitted retrieved material or tool results. The full record stays in protected application state. Tools get an opaque reference or a validated token, checked for audience, integrity and expiry. Credentials, session tokens and API keys stay in the execution services; they never become prompt fields. Include an em-

ployee or claim identifier in model input only when the task needs it. Nothing the model sees grants it access to a tool.

Recheck as the request changes. An authorization check protects each lookup or action at its own boundary. A new subject, a new purpose, a material source correction or a changed entitlement means validating again and updating the record. An earlier "ready" isn't a standing grant. Each model call gets the context currently permitted. Record which version you used alongside the actual response and tool results. That way a later reviewer can tell the context you supplied from an explanation generated after the fact.

The workflow owner approves the context specification and its clarification and failure rules. Identity, organization, case and finance owners stay accountable for their own authoritative attributes. The application team builds the assembly and validates it with the security and service owners. Attribute-based access control evaluates the relevant subject, resource, operation and environment attributes against policy; a department label alone isn't enough. [48] The context-record design is the pattern I propose. NIST doesn't require it as a schema.

Here's how the claims request plays out. Sign-in identifies the examiner. An assignment check establishes the permitted case. The policy and claims interfaces supply authorized facts, and the approved applicability rule picks the procedure version. The record captures the purpose, subject, scope and time basis. The brain receives only the permitted preparation context and returns facts with their sources. It infers no entitlement to pay the claim or contact the claimant.

Route the request to the right kind of source

Retrieval of documents sits alongside the structured sources. It brings back applicable text and where it came from. It can't supply a missing business decision about what a measure means. One request may need several routes under one constrained plan. The brain's job is to pick them without becoming a copy of every source system.

The distributor's credit-hold questions show the routing clearly.

Table 12. Bring the right source to each part of the task

Question	Where the answer comes from, and the limit
What does "invoice balance" mean here?	The versioned definition, with currency, valuation time and legal entity, through shared identifiers and mappings between domain models
Why is this invoice on hold?	The current hold through a governed source interface, plus the permitted policy passages. Neither the live fact nor the document is an instruction to the agent.
What changed across the portfolio this month?	A contracted analytical product or semantic query at the right grain, period and freshness, with any delay against the system of record stated

Question	Where the answer comes from, and the limit
Can we release the hold?	A narrowly scoped command tool. Retrieval, a graph relationship or a convincing explanation can't authorize the write.
Should we copy this content at all?	Ingestion, controlled indexing or querying the source, chosen on rights, latency, resilience, freshness and cost. Virtualization doesn't remove those obligations.

Preparing documents for retrieval can involve OCR, parsing, chunking, carrying permission metadata and hybrid indexing. Agent memory is something else: a separate derived store with its own tenant scope, provenance, retention and deletion rules. Check its contents again before reusing them. Yesterday's permitted retrieval doesn't authorize today's disclosure.

Choose the mechanism for the work

Retrieval brings selected information in at request time. It doesn't normally change the model's weights. Fine-tuning does change the weights, through training, and needs its own data rights, evaluation and supplier terms. Neither gives you a dependable source of current transaction state. Use managed queries or calculations for exact balances and dates. Retrieve applicable text for explanation. Use generation where interpreting or composing language adds something.

A schema-constrained response is easier to parse, and it lets you reject malformed values. It can't establish that the content is accurate, applicable or permitted. Calculate consequential figures in tested services. Check generated claims against permitted sources, and give the application a defined "incomplete" state. A citation is useful only if the version and passage it names actually support the claim.

Carry the permitted operation to its source

The nervous system exposes operations like reading claim context or retrieving applicable procedure passages. Each one defines its inputs, outputs, scope, limits and failure states. A protocol makes a tool callable. Its contract and enforcement decide whether a particular call is acceptable.

The service resolves the verified caller and the relevant workload identity. It carries scoped authorization context to the systems that enforce access. When a call is denied, the model must not be able to pick a more privileged identity. A service using its own database connection still has to apply the approved user, tenant and purpose restrictions. A successful connection proves almost nothing about those boundaries.

For delegated work, the agent's effective authority is the overlap of three things: the initiating person's entitlement, the agent's approved tools and the task's restrictions.

Background work needs its own explicitly approved identity and purpose. Credentials and tokens belong in the identity design, never in a retrieved document or a model-generated argument.

At each service, validate the credential through the chosen trusted mechanism, including its intended audience, issuer, lifetime and scope. A valid token doesn't authorize every object or field. For downstream calls, use an approved delegation or token-exchange design. Don't forward a token to a service it wasn't meant for. MCP's security guidance explicitly rejects token passthrough, and a tool protocol doesn't supply business authorization. [40; 41]

The status tool uses a source that can meet the task's currency requirement. The procedure tool selects applicable material and filters access before any text reaches the model. Both return provenance and an observable state. The answer separates source facts from interpretation and names any material gaps.

The environment is part of the result

The body supplies the conditions those operations run under: approved deployment, identities, configuration, monitoring, limits and support. They're easy to overlook. Everything about them is settled before the examiner clicks anything.

Record the deployed combination of model, prompts, tools, retrieval settings, definitions and policies. Track source state too, because it changes on its own schedule. A release identifier can't tell you which procedure version was live on a given day unless the request record also captures the source context.

Users need sensible behavior when a dependency fails. A stale status source might justify a clearly dated historical answer for an approved preparation task. Or it might require a refusal. The contract decides. What the service mustn't do is quietly switch to another source whose freshness or authority nobody has accepted.

Operators need controls too. They have to be able to restrict a route, inspect the relevant records and restore service through an approved recovery process. A monitored dependency with no responder and no permitted action gives you an alert. It doesn't give you an operating capability.

Security runs the whole route

The immune-system job covers discovery and classification, policy decisions, enforcement, detection and response. Several services may be involved. Name where each one is authoritative, and how changes reach the others.

Suppose an examiner's assignment changes. A new request mustn't keep using an old cached authorization just because her session is still open. Where the platform can't revoke immediately, measure the lag, set the permitted window and decide whether the use can live with it. A sensitive workflow may need a different token, cache or online-check design. Likewise, an index built from a restricted library needs the right permissions on its derived content. The source's policy doesn't automatically follow every copy. The service should demonstrate both permitted and prohibited paths, including direct calls that bypass the main screen.

An instruction buried in a document is still untrusted content. It can't authorize an email, a payment or a role change. The first claims release has no such tools, which is a useful constraint. Its read paths still need protection against unauthorized retrieval and disclosure.

Link the records without collecting everything

The control plane links the relevant identities, definition, source references, policy result, tools, release and observed outcome. A correlation identifier lets you find related records. Keep one authoritative source for each decision. Competing versions of policy in every dashboard help nobody.

Keep enough to support the agreed investigation and review purposes, under their own access and retention rules. That may mean protected snapshots of some facts and references to immutable versions of others. A hash can show that retained content changed later. It can't show that the original was true.

Reconstruction means explaining the conditions and inputs of a request you observed. It doesn't mean a probabilistic model will produce identical wording later. Need the exact text the user saw? Keep it.

The control plane also has to handle disagreement. Suppose classification has been updated but a serving policy hasn't caught up. Someone has to own the gap. There has to be a way to reconcile it, and the service needs a safe response in the meantime. A last-approved version stays usable only while it still meets current authority and source requirements.

A stewardship agent can compare the approved record with what's deployed and prepare a correction. It mustn't become the authority for both sides of that comparison. Protect the approved definitions, policies and acceptance criteria from its edits, and route real discrepancies through the decisions described in Chapter 16.

Trace the failures as carefully as the success

Run the same request after revoking the examiner, removing a source document, changing the applicability metadata and taking the policy service offline. Try a cross-tenant identifier, a conflicting party match and instruction-like text planted in the documents. Write down the expected result before each test.

Then explain which responsibility detects each condition and which component responds. A catalog warning can't stop a tool unless the tool reads it and acts on it. A gateway can't remove unauthorized passages once they've entered the model's context. And an authorization check at the front door doesn't restrict a privileged query further downstream.

Write an interface register for the request. For each hop, record the owner, the authoritative input, the result schema, the permission boundary, the version, the freshness requirement and the failure response, and link to the actual implementation. That register will do more for you than a diagram full of unexplained arrows.

What you end up with is an architecture you can review through behavior. A team should be able to trace one permitted request, one denied request and one failed dependency without filling any gap in the story with an assumption. That's the foundation for choosing products in the next chapter.

CHAPTER 8

Build on the systems you already run

Choose components for the jobs and the proof the workflow needs. Start with what you already license and operate. Check the gaps honestly, and add a product only when it closes a material one. Write down the decision and an exit test. A component is replaceable when you can carry its contracts, controls and operating behavior across to something else. Two products happening to expose similar APIs doesn't count.

Four decisions hiding in one platform question

"Should we use ChatGPT or Claude?" "Should we be on Bedrock, Azure or Google?" An executive who asks either is usually asking four questions at once. Each has a different owner and a different shelf life.

The first is which assistant employees use every day. Start with the one native to your productivity suite, because it already sees your documents, mail, meetings and permissions. Add a second assistant for a named group only when a short cohort test shows it gets more accepted work done.

The second is which models do the work. No single model is best, and the leader changes within months. Choose per class of task on your own test cases. Approve at least two frontier providers so you have a fallback and some bargaining power, and route high-volume work to a fast, cheap model.

The third is how you reach those models. You can use the vendor's own API, your cloud provider's catalog, the vendor's service bought through the cloud marketplace, or open weights you host yourself. The same model name can come with different data handling, features, quotas and terms on each path. Your strategic cloud's catalog is usually the right default. A model on its own is rarely a good reason to take on a second strategic cloud.

The fourth is where agents and automations run. Automate at the lowest tier that can carry the risk. That means assistant features first, then low-code builders, then a pro-code runtime, with process orchestration reserved for long-running cases that coordinate people, robots and systems.

Models change monthly, so keep the second decision easy to reverse. The runtime and its integrations last for years. The fourth decision therefore deserves the gates and the production-shaped proof described in the platform decision guide. [88] The rest of this chapter covers that fourth decision and the data paths beneath it.

Different questions need different sources

The examiner's request contains several information needs. Current status may have to come from the operational system. A historical benchmark belongs in an analytical product. The applicable procedure belongs in a managed knowledge route. Reaching all three through one assistant doesn't make them interchangeable.

A warehouse can hold an excellent history and still be too stale for a live business action. A document index can find a passage without showing it's authoritative for this case. A low-latency serving copy can speed up responses. It also adds one more place where someone has to maintain access, deletion and refresh.

Start every component decision from those needs. Write down the purpose, the permitted users, how current the data must be, what an error would cost, the load, the recovery requirement and who will operate it. Test a representative request under realistic identity and source conditions. A polished vendor demo is good for discovery. It isn't acceptance testing for your deployment.

The Reference Implementation paper offers one combination of products. Patterns and Product Fit looks at boundaries in more detail, and at smaller alternatives. [10; 11] Treat the products named there as candidates. They aren't a shopping list.

Copy or query is an operating choice

Querying a source where it lives can avoid moving data, and it keeps you close to current information. It also puts new load on the source, ties you to its availability and makes reproducible history harder. A copy gives you controlled performance and snapshots. It adds synchronization, rights and deletion obligations.

For each path, ask what the source provides and what the consuming service has to add. A remote query doesn't automatically inherit the warehouse's row policy. An open table format doesn't make every engine that reads it apply the same controls. And a copied dataset doesn't become unrestricted just because it now sits outside the original application.

Record the permitted purpose, owner, source version, access boundary and retention. For a copy, add refresh and deletion behavior. For in-place access, add how identity is carried, latency, load on the source and behavior during an outage. Cost the whole path, including

egress, duplicate storage and support. “Zero-copy” describes how data moves. It says nothing about cost or control.

Map the full supplier data path before sending anything protected. Prompts, retrieved passages, embeddings, reranking inputs, tool responses, files and telemetry may each go to different services or regions. For each route, confirm the approved purposes, retention, training use, subprocessors, transfer terms and deletion behavior. Permission to read an internal source doesn’t by itself permit sending it to a model supplier or using it for training. Apply the agreed handling rules to outputs and logs as well as inputs.

A period-end finance product may need a controlled historical snapshot. The current-state claims tool needs a live interface. Neither choice should become an enterprise rule before you’ve looked at the other workloads.

Treat the index as a separate serving surface

Retrieval needs its own tests. Permissions get lost between the source and the index far too often.

Work out who can query the service, whose rights the search uses, which permission metadata the index stores and how it changes. Access to the service and access to the documents are separate questions.

Snowflake’s Cortex Search documentation describes owner’s-rights behavior. Granting someone use of the search service doesn’t mean each caller’s source-table restrictions apply automatically, so the application needs its own access design for the indexed content. [26] Databricks documents that its AI Search doesn’t support row- and column-level permissions as native index permissions. Azure AI Search documents several approaches to document-level access, including generally available security-filter patterns and native integrations still in preview. [27; 28] Those are different mechanisms, and they put different responsibilities on you.

Don’t choose from the labels. Test your actual source, identity model, deployment and edition. Use native document authorization where it verifiably covers the route you need. Otherwise, use an application-enforced security filter built from trusted identity. Make sure neither the client nor the model can weaken that boundary. Carry the full restriction metadata on every chunk, and measure how fast permission changes reach every supported query path. Chapter 14 covers the tests.

I checked these product behaviors for this edition. They will change, so recheck them before you build. They show why you have to evaluate an index on its own. They don’t make one platform universally safer than another.

Keep the jobs separate when a product combines them

A platform may bundle storage, catalog, semantic models, model hosting and search. For a small team, that cuts a lot of integration work. It also tempts everyone to assume one component's controls cover every feature.

Ask the same questions at each supported surface. Which identity does it evaluate? Which policy applies? Does it compute the result from live data or a serving copy? Could an application cache bypass a new decision? Does a connector carry source permissions, or only source content? The platform's name answers none of these.

A data security posture management tool can find sensitive data and exposure and, through supported integrations, feed protection workflows. A catalog can connect those findings to approved meaning and ownership. Chapter 15 follows that connection into native policies and developer tools. A gateway can control supported model and tool traffic. The source or serving path still has to enforce its own access boundary.

Power BI and Tableau consume managed measures. They don't prove that every upstream definition or permission is consistent. Where BI tools and assistants share a semantic interface, compare the same metric version, scope and dates through each permitted route. Record any extra cache or extract a consumer creates.

Keep the decision record short and testable

Someone outside the original team should be able to read a component decision record and understand two things: why the team chose the product, and what would justify replacing it.

Table 13. A short, testable decision record

Field	What to record
Required responsibility	The task the component performs and what stays outside it
Alternatives	Existing capability, a simpler implementation, the proposed product and why others were rejected
Interfaces and data	Supported inputs, outputs, definitions, identities and source rights
Acceptance tests	Representative load, known answers, denial tests, failures and recovery
Dependency conditions	Edition, region, availability status, licensing and supplier commitments
Operation	Owner, support, service objectives, monitoring and cost drivers
Exit	Exportable artifacts, replacement conditions, transition, deletion and leftover commitments

A preview feature can be fine for an approved experiment and wrong for a workflow people rely on. Record the terms and any open gaps. An announcement isn't a deployed capa-

bility. If a requirement depends on a feature that isn't available yet, you have three choices: change scope, add another control or wait. Don't describe the future feature as part of today's solution.

Rehearse the exit while you still understand the system

Pick one bounded replacement exercise. Export an approved metric and its tests. Swap in a different model for the same task and run the same evaluation pack. Rebuild a small retrieval collection with its source references and restrictions. Practice on a representative contract. An enterprise migration is the wrong thing to practice on.

The exercise shows what exports cleanly and what you'll have to rebuild. A prompt export can miss routing, provider quirks, tool semantics and evaluation. A data export can miss permissions, history, lineage and meaning. An endpoint that looks the same may fail differently, or count tokens differently. List the differences and what they cost.

Test revocation and cleanup in the old service. Removing an application from the screen doesn't necessarily revoke its credentials, stop its jobs or delete its copies. Supplier deletion commitments and your own retention rules may not match. The owner needs a reconciled list of what remains, and why.

Portability isn't a yes-or-no feature. The useful question is which obligations can move, at what effort, with what proof. Some lock-in is a fair price for simpler operation. Choose it, cost it and revisit it. Otherwise you'll discover it during an urgent exit.

Build a constrained combination

The insurer's first release can use its existing identity provider, controlled source APIs, current data platform, a limited knowledge index, an approved model route and its established incident system. Some pieces already exist. Others need a narrow integration. The design should name the gaps instead of assuming a wholesale replacement.

The distributor may fit more of these jobs into one managed platform. It still needs a current order source, managed definitions, restricted retrieval, operating records and support. A new universal semantic product is unnecessary if a few tested definitions do the job. Consider one when inconsistency across platforms causes a measurable problem.

Test the complete route before you buy the expansion. Run the request as a permitted user, as a prohibited user and with a failed dependency. Compare it with a simpler search or a plain deterministic automation, wherever either could do the task. Chapter 15 develops the DSPM and catalog connections that help make protected information usable through those routes. Then pick the combination whose behavior and running cost fit the capability, with a believable path to change it later.

CHAPTER 9

A lean implementation and its limits

A smaller company can build a useful, well-controlled capability without copying an enterprise program. Use one familiar managed stack where it meets the need. Keep the sources and tools few, and write down who runs what. Keep the records that protect against the consequences you actually face. Add components when a measured gap calls for them, not because the reference diagram has another box.

Start with the service desk question

The distributor's service agents keep answering two questions. Where is this order? And which promised date should we give the customer? Order records, shipment updates and a small procedure library hold most of the answer. Agents spend their time finding and reconciling that information before they reply.

The chief operating officer owns the workflow. The first capability gathers permitted facts and cites the procedure behind them. Agents still decide what to tell the customer. The assistant doesn't send messages, change promised dates, issue refunds or release credit holds.

This is a composite design, not a finished client deployment. Nobody has measured savings from it. It shows how the same responsibilities fit a smaller operation. The lean patterns in the source paper lean on native platform services and add components only when a real gap appears. [11]

Start with one service group and a handful of order types. Record today's preparation time, errors, repeat contacts and manual exceptions. Pick the observation window and the cases before you look at any results. You may find that inconsistent order updates, not summarizing, cause most of the delay. That's a useful finding, even if it changes the AI work you planned.

The smallest useful information model

The distributor needs written definitions of customer, order, shipment, promised date and credit hold. "Customer" might mean the bill-to account or the ship-to location. "Promised date" might mean the original commitment or the latest approved revision. Those distinctions change the answer whether or not you ever build a knowledge graph.

Keep a small, managed dictionary with each definition's purpose, owner, source and effective date, and test known cases against it. Where the task needs to explain a change, keep the promised date's history. A spreadsheet can hold the approved definitions if access, versions and change control are good enough. It can't stand in for enforcing data access in the application.

When the task needs current state, use the order system's supported interface. If the service works from a scheduled snapshot, show its as-of time and limit the questions it will answer. Yesterday's shipping status isn't live just because the assistant replies quickly.

Put only approved material for this workflow in the procedure collection. Every document needs an owner, a version, an applicability rule, an access scope and a replacement process. A smaller collection keeps those obligations manageable. Ingest every shared drive and you add uncertainty, not usefulness.

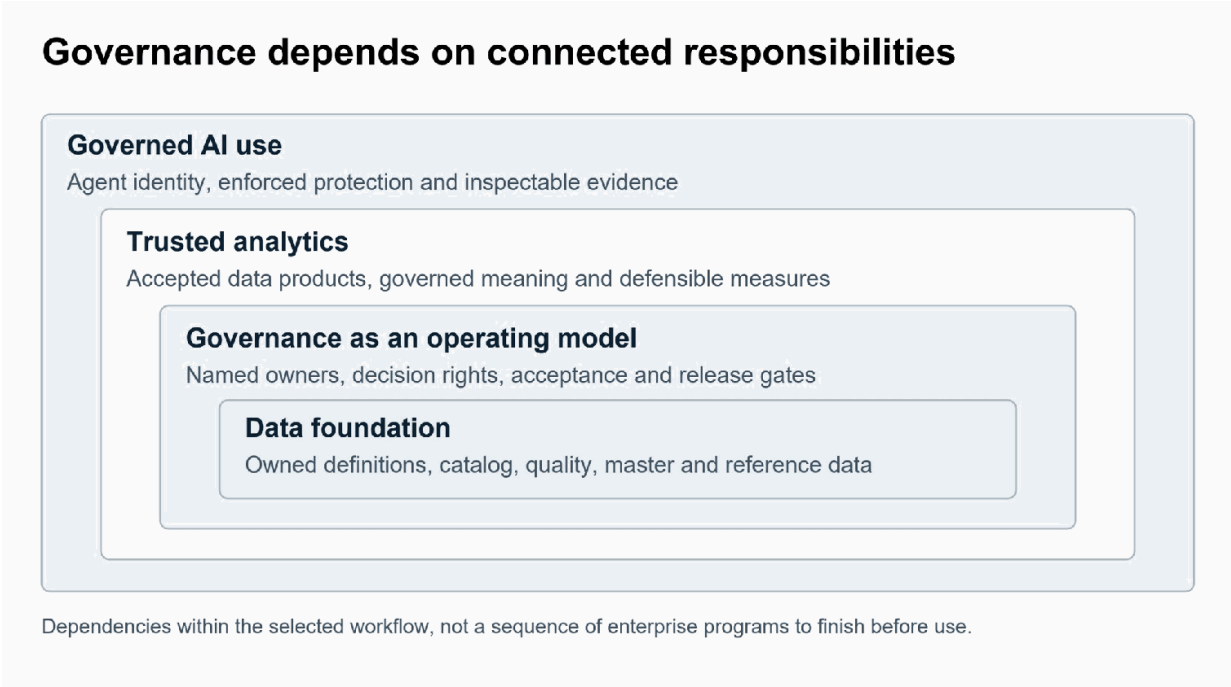


Figure 15. Governance as four rings: dependencies, not a sequence

Read the rings as dependencies within the scope you've chosen. They aren't four enterprise programs to finish before anyone uses anything. The first workflow needs enough owned information, operating decisions and controls, all at the same time. It doesn't need every semantic or platform component that could exist.

One supported route

A lean request can follow a short path: a verified service agent, a constrained application, approved read tools, current order information, retrieval of the applicable procedure and

a cited result. A managed platform may supply several of those steps. The company still has to check the boundaries itself.

The same backend can build a small context record. It holds the verified agent's reference, the customers and orders that agent may see, the confirmed task, the current order observation, the applicable procedure and the relevant dates. The chief operating officer approves the required fields and definitions; the engineer builds the lookups and checks. If several authorized customer accounts fit the question, the application asks which one before retrieving anything. A small implementation needs that behavior even without a separate context service.

The application takes the user's scope from the authenticated session. It validates the order number against the accounts the agent may access. It retrieves only what the agent may see, filtering permissions before any restricted text reaches the model. And it returns source references and names any material gaps. It doesn't invent a complete answer.

Keep provider credentials out of the client and out of the model's context. If a tool runs under a shared backend identity, the backend still has to enforce what the initiating agent may see. Don't expose a general query or update function just because it's faster to prototype than a narrow order-status lookup.

Prefer a narrow, parameterized read operation to model-generated SQL. Sometimes open-ended queries are genuinely needed. Then validate the permitted tables, joins, functions and columns, enforce limits on resources and results, and run under a restricted identity. A read-only connection can still leak too much data, call unsafe extensions or overload a source. Treat it as one safeguard among several.

The release record names the model, prompts, tools, definitions, source configuration and policies. A simple versioned repository and a protected deployment path are enough. A small firm doesn't need a large registry product if its smaller record is complete, findable and kept up to date.

Divide a few roles on purpose

The business owner approves the changed service process and how you'll measure results. The engineer owns the build and the technical operation. An analyst or an experienced service lead maintains the definitions and grades the answers. A named security reviewer checks access and source restrictions. People can hold several of these roles, as long as conflicts of interest and available time are out in the open.

Consequential authority stays separate. The credit controller keeps deciding when holds are released. If someone later proposes a write service, its approvals and recovery need

independent review suited to the risk. A small firm shouldn't hand an engineer business authority just because they run the platform.

Name who covers absences and how to contain a problem. Even with the engineer out, the service lead should know how to stop use and send agents back to the existing system. The fallback needs enough source access and written instructions. A hidden dependency on one person's laptop doesn't fit a service the company plans to rely on.

Budget the real hours. Part-time roles are still work: procedure upkeep, evaluations, access changes, incidents, training and provider administration. If it doesn't fit, shrink the scope or get help. Calling the setup lean doesn't make the obligations disappear.

A small release pack with consequential tests

The release record can be short. Include the approved scope, owners, information contracts, tests, operating limits, fallback and the release decision, with examples and actual results a reviewer can inspect.

Table 14. A small release pack with consequential tests

Test	Required behavior in this example
An agent asks about an order outside their permitted accounts	Denied at the serving path; no restricted facts reach the answer
Bill-to and ship-to differ	The answer uses the role the question needs
A promised date changed after the customer's last contact	The source and the change history are kept distinct
A procedure is deleted or its permission revoked	Retrieval and caches stop serving it within the approved window
A document tells the assistant to release a hold	No write tool exists; the instruction grants no authority
The order source is down or stale	A defined incomplete result or a dated historical result, with fallback
The provider budget or call limit is reached	Controlled deferral or the manual path, with unmet demand recorded
The engineer is absent during a simulated incident	The named cover contains the service and follows escalation

Call it a starting pack. It isn't a security assessment. Add cases from your own systems, data and consequences. Passing it supports a limited release decision; it doesn't mean future failures can't happen.

Measure the useful result

Compare complete preparation work before and after, with checking and correction included. Track eligible cases, actual use, accepted completions, repeat attempts and fallback. Weigh errors by consequence, not just count. A wrong delivery-date explanation and an unauthorized account disclosure aren't the same kind of problem.

Introduce the assistant in a controlled way. A small first group will find defects. It may also be enthusiastic people working easier cases. Test the wider eligible population and the hard work before assuming the results will hold. Chapter 18 covers that evaluation.

The value may be faster replies, less repetitive preparation or more time for exceptions. A cash claim needs its own realization action. If agents save minutes and the company changes neither staffing nor purchased capacity, call the benefit capacity or better service. Don't call it savings.

Know when the design is too small

Growth trips different wires. A second platform can spawn duplicate metric definitions. Several client organizations can demand stronger tenant isolation. Complicated document rights can outrun what a connector can carry. Consequential writes bring approvals, durable operation state and reconciliation. Round-the-clock service may need more support than one engineer can give.

Set a threshold for each real problem, and name who decides when it's crossed. Examples: a revocation lag beyond the accepted window, a fallback queue bigger than the staff can handle, a second consumer that needs incompatible definitions, an incident investigation that can't recover the records it needs. Base the threshold on your risk and baseline. A number borrowed from another company won't fit.

You can automate routine investigation inside the same small service. Start with gathering facts and proposing fixes. Only then consider a narrowly approved runbook action. Chapter 16 explains how to lighten that load without building an unfunded agent platform or sidelining the business owner.

Some additions stop being optional once their trigger appears. If you can't carry and enforce source permissions, restrict the collection to material the whole approved audience may read, use a different retrieval route or turn the feature off. A connector improvement promised for next quarter isn't a control today.

Others are economic choices. A universal semantic service may cut repeated modeling across several platforms. A posture-management product may improve visibility across a growing estate. Weigh those benefits against licensing, integration and ongoing adminis-

tration. And prefer a limited trial to assuming every enterprise component belongs in the next release.

Growth and investor scrutiny

A private-equity-backed company may need to show two things: that the benefits survive a change of management, and that the service can be supported through an acquisition or integration. Its records should connect accepted work, avoidable cost, owners and supplier commitments. A portfolio-wide rollout mustn't flatten distinct data rights, tenant boundaries or local process authority.

A venture-backed company building a product for customers faces a different question. Can customer isolation, onboarding, deletion and support hold up as tenants multiply? An internal order assistant doesn't prove a multitenant product is ready. A shared model endpoint doesn't establish that customers' data is isolated.

The lean implementation succeeds when a small team can explain and run one complete request path. It reaches its limit when its assumptions or its capacity can no longer carry the intended use. Expand only after you can see that point. It's a practical way to protect both speed and investment.

PART 4

Data the business can trust

CHAPTER 10

A data product is a maintained promise

A data product is information, or a capability, that a defined consumer can find, understand and use through a supported contract. Its owner keeps that promise as sources, permissions and business meaning change. The product might be a table, an API, an event stream or a managed collection of documents. Where it lives and what it's called tell you nothing about whether it fits the consumer's purpose.

The table that was ready for someone else

A composite scene. At 10:40 in the morning, an examiner preparing a case review asks for a claim's current status. The assistant checks the published status table. It reports the claim as open and approved, without making clear how old that information is. The table is accurate as of last night's close. At 9:15 that morning, the claim was put on hold for a fraud review.

No field is corrupt. No pipeline failed. The table was built for a monthly operations report, and it does that job well. The error is a mismatch between what the product promised and what this consumer needed: a daily snapshot presented as the claim's state right now. Labeling the table gold, or certified, wouldn't have changed the answer. So whose mistake was it?

Start the contract with the decision and the consumer. What may the consumer rely on? How current must it be? What happens when that condition isn't met? Data Products in Practice applies this approach to analytical, operational and knowledge products while keeping responsibility and lineage distinct. [6]

A product, not a project

A dataset with a new label is still a dataset. Plenty of programs rename existing tables as products and publish them to a catalog. Without a defined consumer, an owner and a promise someone keeps, though, what you have is an artifact. A project gets built to be delivered. A product gets built to be used, reused and kept valuable after the delivery team moves on.

Confirm demand before you build. A draft contract, a sample with real definitions or a catalog listing will show whether anyone will use the product, and what it must promise them. A composite scene: a finance function publishes dozens of products to an internal marketplace, every one cataloged, documented and tested. Adoption stays low and teams keep their spreadsheets, because nobody confirmed which decisions the products were supposed to serve. Engineering quality was never the problem. Demand was.

Ownership doesn't end at publication. The owner answers for adoption, the consumer's experience and value, not just quality. Usage, satisfaction and upkeep cost decide whether a product gets improved, merged or retired. Reward reuse too. Teams measured only on their own delivery will keep building private copies; that's the same incentive problem Chapter 3 described for financial definitions. Governance should ask whether a product creates value as well as whether it complies.

Build it in stages, then make the promise

Moving data through layers doesn't create a product. The stages prepare records for supported use. Publication is the moment a team takes on a promise to a consumer.

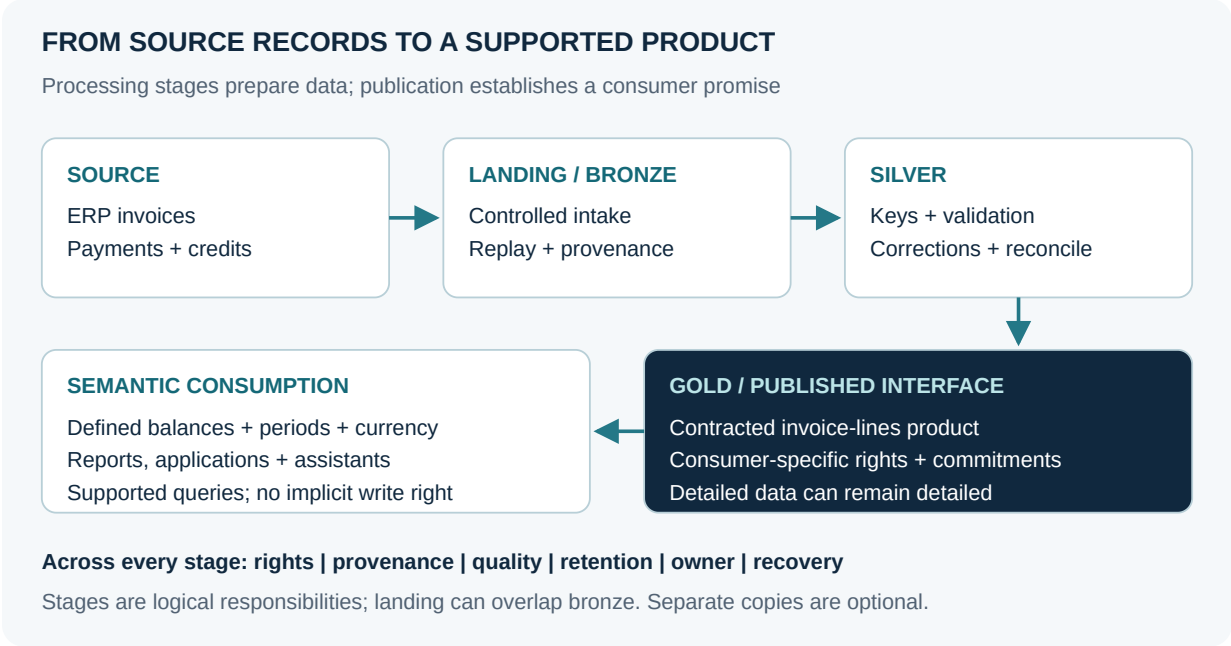


Figure 16. From source to published product: preparation stages and the point of promise

Intake binds each delivery to its source, its extraction position, source and arrival timestamps and a batch or event identifier. Check the delivery envelope and the source's rights, and reject an untrusted delivery before ordinary processing. Keep a replayable copy of the source under controlled access, encryption and justified retention. Capture schema drift and duplicate event identifiers so a replay can't create duplicate facts.

Faithful raw data and strong protection go together; light business transformation doesn't mean light security.

Validation applies the approved types and business keys. It normalizes dates and checks currency and entity references. It sets aside invalid or ambiguous records for the owner to review, with the reason and a link to the source. An unknown currency code gets quarantined, never guessed. An exact repeat of an event falls to the deduplication rule. A revised event with the same business key goes through the correction rule instead; throwing it away as a duplicate would be wrong. Decide in advance whether a rejected row allows partial publication. A reporting deadline or a required total may demand a hold, while another use can accept a clearly bounded partial result. Write that behavior into the contract.

Publication is where a validated dataset becomes a product. The team states the grain and keys, the source authority and owner, the supported consumers and purposes, the interface and its version, the quality and freshness conditions and the support arrangement. All supported routes enforce the same rights. An unrestricted extract on the side is not a workaround. In this reference design, gold is the publishing convention for analytical products, and it needn't be an aggregate. A detailed invoice-line product can be gold so approved consumers can combine it accurately. Another organization might publish a supported product from a silver representation. The contract and the operating responsibility matter more than the color.

Listing a product in the catalog makes it findable. It doesn't certify it or grant anyone access. Certification, which the next chapter covers, applies to a product version and a supported use.

Two axes: what kind of product, and how it's built

Delivery type is one dimension. An operational interface needs state, latency, ordering and failure behavior suited to its use. An analytical product needs grain, refresh, completeness, period and finality. A knowledge product needs document authority, applicability and propagation of revisions, permissions and deletions. An embedding or a knowledge graph on its own isn't a knowledge product. It has no contract.

How the product gets built is a separate dimension. A foundational product publishes a stable contract around one domain capability, close to where the information is created. A derived product combines or transforms supported products for a consumer's purpose. It owns the new meaning it creates, along with its lineage and service commitments. Foundational doesn't mean raw. Derived doesn't have to mean several domains, aggregation or more complicated transformation. Any of the three delivery types can be foundational or derived. Describe the actual inputs, grain and promise; don't lean on the label.

A derived product inherits its inputs' restrictions and takes responsibility for what it adds. Certifying a source doesn't certify a consumer's new calculation.

Don't merge unrelated responsibilities under a customer-360 or policy-file label. A useful composition can expose several products. Its contract, though, should keep source authority visible and show its own interpretation. The person who maintains consent may not own claim status. A consumer shouldn't have to guess which part of a combined view to trust for which purpose.

Grain is part of meaning

Say what one record represents. One row per claim, one per payment and one per status event support different operations. Join them carelessly and you multiply rows and inflate totals, while every source value stays individually correct.

Define keys, relationships, allowed joins and the time that matters. If a party can play several roles, keep the role on the relationship. Don't collapse it into a generic customer identifier. If a payment can be reversed, say whether the product holds events, current net amounts or period-end balances.

Write examples before trusting the schema. An examiner and an engineer should be able to work out the expected answer for a small case and agree on what it means. Include a reopened claim, a late event, an unresolved party match and a correction. Field names rarely capture those.

The contract also says what the product doesn't support. The claim-status product may support preparation and exclude coverage decisions and payment authorization. Those exclusions matter once an assistant combines the product with other sources. Being able to reach a product isn't permission to use it for every plausible purpose.

A worked example: one invoice, two dates

The distributor's receivables show how construction, grain and time fit together. The numbers are synthetic.

Invoice INV-100 has one currency and one legal entity. At cutoff T0, its lines total 1,000 in gross charges, approved credits total 100 and applied payments total 400. The simplified definition used here says outstanding balance equals gross invoiced amount minus approved credits minus applied payments, for the same invoice, entity, currency and cutoff. So the balance at T0 is 1,000 − 100 − 400 = 500. Other adjustment types would need their own rules.

The inputs have different grains. Invoice lines are one row per line. Payment allocations are one row per allocation. Credits may have their own grain. Join every detail row and

then sum, and an invoice with two lines and two payment allocations multiplies both totals. So aggregate each input to the declared invoice, entity, currency and cutoff key first. Then join the reconciled results with the approved cardinality and missing-record behavior. Keep that two-line, two-payment invoice in the test set permanently, along with duplicate events, a missing credit and a backdated payment.

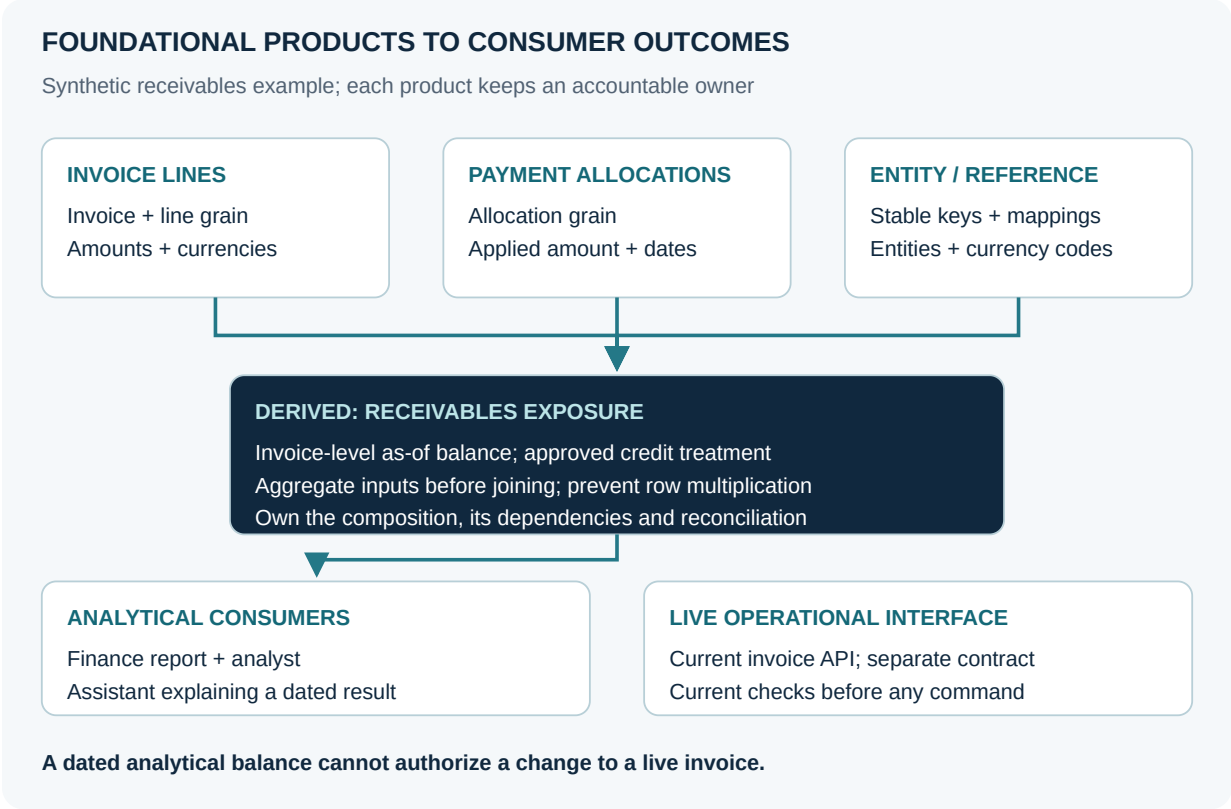


Figure 17. Foundational invoice and payment products composed into a derived receivables product

The foundational products are Invoice Lines and Payment Allocations, each owned close to its source. The derived product is Receivables Exposure. Its owner agrees the calculation, period, entity scope and credit treatment with finance, and answers for the combined result: compatibility, critical dependencies, permissions and reconciliation. The source owners keep responsibility for their own interfaces.

At T1 another payment of 200 arrives, and the current balance becomes 300. The published T0 record stays as a dated version. A late payment changes what is now known without silently rewriting what an earlier report knew at its cutoff. A daily finance report and an invoice assistant can both use these products and still need different promises. The report needs required sources, a cutoff and finality, and it should hold final publication if reconciliation fails. The assistant needs the source cutoff and version returned with every answer, plus a maximum permitted age. A dated analytical balance can't authorize a

change to a live invoice. Ask the assistant to change a hold, and it must fetch current state and follow the separately authorized command path in Chapter 13.

Freshness needs a clock and a population

A refresh job finishing on time doesn't prove the product holds current information. Pin down four times: when the source fact happened, when it became available, when it was ingested and when the consumer could read it. Then measure the one that matches the promise.

For an event feed, define the population of eligible events and how you'll notice missing ones. A quiet source needs a heartbeat or some other completeness signal. Looking at the last event received can't tell "nothing happened" from "the feed broke". For a daily product, name the cutoff, the required batches, the delivery time and the rule for late arrivals.

Be precise about service terms. A service level indicator measures behavior, such as the share of eligible daily publications delivered by their deadline. A service level objective is the target for that measure over a stated window. A service level agreement adds the consequences of missing it, and not every internal target needs one. Define the denominator, timezone, source dependency, exclusions, window and who operates it. Monitor end-to-end freshness as well as pipeline success. A pipeline can finish on time while its input is incomplete or old. [64]

A percentile target also needs a breach response. Most records may arrive in time while the one record a decision depends on is late. An average shouldn't hide that. Some workflows need a maximum age per request as well as a measure over time. The interface should expose whatever the consumer needs to decide whether to proceed.

Finality is a different property. A current figure can be preliminary. A historical figure can be restated later. When consumers need the approved period and correction history, keep them. Chapter 12 goes further into valid time and recorded time.

Reconcile the feed before trusting the snapshot

A change-data or event feed needs rules for the first snapshot, the switch to live changes, replay, duplicate delivery, late or out-of-order events and deletions. Keep source sequence or version information where you can, and test for gaps across restarts. A recent watermark doesn't prove every expected row arrived. Reconcile counts and business totals against an authoritative population, within agreed tolerances.

Make grain, units, currency, timezone, business calendar and null handling explicit. A missing amount isn't zero. A missing date isn't today. And an event timestamp isn't necessarily a processing time or a legal effective date. Quarantine unrecognized codes and con-

flicting critical facts, or return the contract's incomplete state. Never let a model fill a required business fact because a downstream schema insists on a value.

Write a contract a consumer can use

The claims preparation capability depends on separate commitments for status, party relationships and procedure knowledge. Here's a condensed contract for the status dependency.

Table 15. A claim-status contract a consumer can use

Element	Example claim-status commitment
Purpose	Supply case status for authorized examiner preparation; no adjudication or payment instruction
Grain and key	One current status per claim within its source and tenant scope; history exposed separately
Authority	Designated claims system, with a named rule for conflicting or late events
Interface	Versioned read operation returning status, source-update time, observation time and provenance
Access	Verified examiner assignment and permitted fields; restrictions enforced at the serving path
Condition	Defined current, stale, conflicting and unavailable responses; no guessed status
Operation	Named product and service owners, support window, monitoring and escalation
Change	Consumer notice, compatibility policy, evaluation and migration for breaking changes

Appendix B gives a fuller example as a structured manifest and decision record. It's an example format, not a vendor schema. In your implementation, adopt a versioned schema and have validation reject ambiguous or unrecognized declarations.

Test the contract by asking a real consumer to use it. Can they make an authorized call, interpret a stale response and reach the right owner? A form can be complete while the product is still unusable. This exercise exposes that before anyone relies on it.

A processing layer is not a fitness judgment

Bronze, silver and gold are processing and publication conventions in the reference implementation. Gold is the publishing convention for analytical products. The source and validation layers stay internal engineering assets, and other delivery types use their own contracted interfaces. None of this proves a product fits a consumer. None of it dictates layer names for every organization either. A table can be beautifully transformed and still carry an ambiguous definition, excessive access or no support. An operational API can be a perfectly good product without ever passing through a gold table. Judge the consumer promise directly.

That keeps AI-assisted engineering honest too. Generated transformations can compile and pass structural tests while encoding a wrong join or date rule. Check the output

against approved examples and source reconciliation. The job's success flag isn't enough.

Independence matters here. SQL and tests drafted from the same mistaken reading can agree perfectly. Set expected results from approved source examples and domain review. Then test joins, duplicates, time boundaries and reconciliation independently of the generated code. Put generated code and its dependencies through the normal build and security review. Compiling and passing a self-generated test suite proves very little. [52]

Map the implementation review and executable security tests to SSDF v1.1 practices PW.7 and PW.8, alongside the wider lifecycle in Chapter 15. Keep domain reconciliation and independently set expected answers as additional acceptance checks. Why? A securely built implementation can still calculate the wrong business result. [52]

Use AI to cut repeated translation

Rewiring Data Management with AI proposes organizing delivery around one approved record of intent. [18] Purpose, grain, units, source authority, exceptions and example answers can feed the implementation, tests, policy bindings, documentation and release record. The opportunity is fewer rounds of reinterpretation at every handoff.

An agent can draft that record from an interview and existing material. The business authority still decides meaning. The source owner confirms rights, engineering validates the implementation and the control owner accepts the enforcement. A fluent proposal doesn't become authoritative by showing up in several generated artifacts.

Keep open decisions visible. If the contract only says revenue must be correct, no generator can recover the missing treatment of returns, timing or currency. Repeating the same vague definition in SQL, tests and documentation just builds consistency around a gap.

Change meaning as carefully as structure

Suppose the party product changes how it treats policyholder and claimant. The field names stay the same. A preparation summary, though, now refers to a different person. That's a breaking change for consumers, even if a schema comparison passes.

Version the interpretation and tell the affected consumers. Where old and new have to coexist, say which interface and definition each consumer uses. Provide known-answer tests and a migration period suited to the consequences. Never quietly point a stable product identifier at an incompatible meaning.

The product owner coordinates the source correction and its downstream effects. A temporary local fix should record its basis, scope and expiry, and the team should reconcile it once the source is corrected. Otherwise the derived product becomes a hidden alternative source whose differences nobody can explain.

Keep the promise after publication

Monitor the commitments, not just pipeline uptime. A failed quality check should reach the owner and change what affected consumers see. A breaking change from a supplier should trigger an impact assessment. A change in support or ownership needs acceptance before the old arrangement goes away.

When a second consumer asks for access, compare purpose, grain, rights, latency and service cost before reusing the interface. Reuse the existing contract where it fits. Where it doesn't, publish a derived interface or a justified separate product. Review adoption, accepted outcomes, support effort, incidents and full cost, then expand, improve, consolidate or retire on the evidence. A permanent catalog badge and a growing consumer count aren't reasons to keep investing on their own.

The distributor can keep the same discipline with a short contract and a few tests around orders, shipments and promises. A large institution may need more interfaces and more proof. Both need a consumer who understands the permitted use and an owner who can keep the promise.

This chapter produces the completed product contract. Certification, in the next chapter, records whether the proof supports relying on that contract for a specific version and scope. It can't make a vague promise precise after the fact.

CHAPTER 11

Certification that expires when the evidence changes

Certification records permission to rely on a particular product for a stated use, under conditions someone accepted. It needs an identifiable version, people authorized to decide, proof anyone can inspect, and triggers for renewal or restriction. The decision holds only while those conditions hold. A permanent green badge can't tell you whether yesterday's tests still cover today's source, permissions and consumers.

The badge survived the change

A composite scene. On a Tuesday in March the claims system starts using a new status code, RO, for reopened claims. Nobody tells the data team. Why would they? The field is still two characters long, so the pipeline accepts every record without complaint.

The assistant has never seen RO. Its mapping knows open and closed, so it treats the unknown code as closed. An examiner asks whether she can finalize a claim. The assistant tells her the file is closed. In fact it was reopened that morning, when the claimant sent new medical records.

Now look at what everyone else sees. The catalog still shows a green certified badge. Every test still passes, because someone wrote every test before RO existed. By every signal the organization watches, nothing is wrong.

So what was certified? Not the product as it runs today. A version, tested against the codes that existed on the day of review. The badge described a moment, and nobody told it the moment had passed. The product needed three things: a way to notice the new code, an owner to decide what it means, and a serving rule that admits uncertainty instead of hiding it.

The RJL Data Product Certification Standard proposes an internal reference pattern for these decisions. [13] It isn't external accreditation, and it makes no claim of regulatory conformity. Its value comes entirely from the tests and operating rules an organization adopts and actually enforces.

Say exactly what is being accepted

A certification decision should identify the product, manifest, implementation, policy, environment, interface, purpose, consumers and relevant dependencies. It should link the test population, results, reviewers, open conditions and next review trigger. Every reference has to resolve to the version that was reviewed. Silently following whatever file is newest defeats the purpose.

Certification is scoped. A product accepted for monthly reporting may not support current-state decisions. An API accepted for a small internal population may not support a customer-facing service. A source's certification doesn't authorize a new aggregate, dashboard or agent that combines it with other information. A gold table can be uncertified, and upstream certification alone never certifies a new composition. Take the receivables example in Chapter 10. Certifying Invoice Lines v1 means keeping the source snapshot or references, rule and schema versions, reconciliation results, a denied cross-entity access test, the cutoff and the approval. Certifying the derived Receivables Exposure product adds join and cardinality tests, plus checks that its input versions are compatible.

The owner proposes the product and gathers the proof. Reviewers decide within their delegated authority. A contributor must never be able to certify a product by editing a field in its manifest. Protect the release decision, and make the deployed state reconcile to it.

Record the unfavorable results along with the favorable ones. A narrow release can exclude a known problem population, if the exclusion is technically enforced and accepted. It mustn't hide that population inside an aggregate and claim the whole product passed.

Keep three kinds of status distinct

Delivery stage describes progress: proposed, building, in review, published, retired. Assurance describes which use has been accepted, and on what basis. Operating condition describes whether the product being served right now meets that acceptance. Keep the three distinguishable, even if the catalog shows one summary label.

The RJL reference papers use a combined vocabulary: sandbox, managed sandbox, certified, degraded, deprecated and retired. If you adopt the standard, keep those terms. A more explicit implementation can keep the reference status and also record stage, assurance scope and condition as separate fields. Think of that as a clarification for implementers. Vendors and external standards don't all share one state machine. Bronze, silver and gold certainly aren't one.

The same interface can be published for a controlled trial, accepted for read-only preparation and prohibited for payment decisions, all at once. When freshness fails, its condition

becomes degraded and the affected use is restricted. The catalog should show that real combination. An unconditional badge hides it.

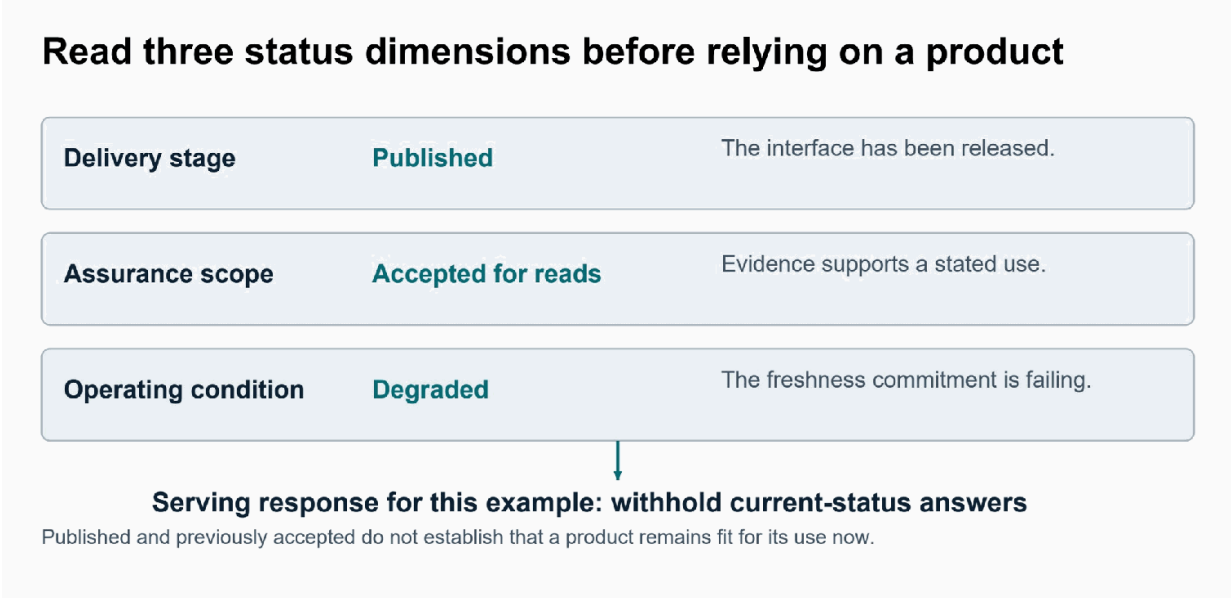


Figure 18. Delivery stage, assurance scope and operating condition

Every transition needs authority and a reason. The standard proposes that an overdue review drops a certified product to degraded and returns a managed-sandbox trial to sandbox. If you adopt that rule, implement it and watch it. A reminder email with no enforced response is a different rule.

Match the proof to the consequence

The claim-status product needs known-answer cases, allowed and denied access tests, source and timing records, failure responses, monitoring and an operating owner. Its tests should cover tenant-specific identifiers, missing assignments, contradictory statuses, reopened cases and late events. The exact thresholds belong in the actual contract.

Separate deterministic checks from statistical signals, and both from judgment. A domain rule can reject an unknown code. An anomaly detector can tell you that volume shifted. The business owner decides whether a new status has a meaning consumers may rely on. An anomaly signal can't certify correctness. And a historical normal range can contain an error that has been there all along.

Set acceptance conditions before you look at the candidate release. Critical access or integrity failures should block the affected scope. Quality thresholds for summaries need a defined task, representative cases, an evaluation method and a rule for disagreements. Passing the observed set supports the decision within what it covered. It doesn't prove the product is always right.

Proof goes stale too. An old load test may not represent a new population. A permitted-use review may not cover a new source license. A successful revocation exercise may not cover a cache someone added last month. Link every material change to the proof it invalidates.

Walk through a decision

Take the review packet for claim-status version 1.0.0, a teaching example. It holds the approved contract, the source mapping, the exact implementation reference, policy tests, quality cases, load results, monitor configuration and a recovery exercise. The consumer demonstrates how it handles stale and unavailable responses. Named owners accept support and recurring maintenance.

The decision approves examiner preparation for the stated population. It excludes adjudication, payments and model training. It records the current results, the authorized reviewers, a review-by date and the earlier triggers: a change in source behavior, wider access or loss of support. Every identifier in the example is invented.

The decision also says who can restrict the product immediately. That's what lets the service respond when the reopened status appears. The ingestion check spots an unknown code, stops it from being served as closed and sends the details to the owner. Depending on the contract, unaffected records can keep flowing while the unresolved cases are held back.

The team works out the new meaning with the source authority. It updates the mapping, tests reopened and late cases, and looks at earlier uses that may have been affected. Restoration records what changed and what now supports it. The unchanged catalog badge has no say in restoration.

The governance loop in Chapter 16 can gather the affected cases, draft the mapping change and run approved checks. The source authority still confirms the new meaning, and the release process still controls restoration. Automation can cut the preparation work. It can't turn a proposed interpretation into a certified one.

Exceptions stay visible

When policy permits, an exception can allow a narrower use under stated conditions. It needs an authorized risk holder, a scope, a compensating measure, an expiry and a test for closing it. A source owner can't waive a consumer's legal obligation, or someone else's access restriction.

Say a noncritical descriptive field is missing for a small subset of records. The approved scope can leave that field out and show it as incomplete. That's nothing like waiving a

failed tenant boundary. The certificate should make the restriction clear to consumers and, where needed, the interface should enforce it.

Track exceptions as work with a running cost. Expired exceptions, repeated renewals and changed assumptions should all come up at review. Don't quietly turn a temporary gap into an "accepted limitation" without the decision that requires.

Renewal is a change assessment

A calendar review is a useful forcing point. A product can become unsuitable long before the date arrives, though. Watch source contracts, meaning, rights, critical quality, service performance, dependency versions and whether an accountable owner still exists. Any of them can change whether consumers may rely on the product.

Use impact analysis to find affected consumers, and be honest about gaps in recorded lineage. A graph built only from observed batch jobs will miss a manual extract or an external application. Maintain the critical consumer relationships explicitly. Then test how you find them during a source-change exercise.

Renewal can confirm the existing scope, narrow it, require fixes or withdraw acceptance. It shouldn't mean signing last year's form again without checking today's condition. Record the next trigger and the owner through the same protected decision path.

Small firms still need a reliance decision

The distributor can use a short acceptance record linked to its order-status contract and tests. The COO accepts the workflow. The engineer demonstrates operation, and a suitable reviewer challenges the controls that matter. The firm doesn't need a certification department. It shouldn't claim an accreditation it hasn't earned, either.

The questions that matter stay practical. Is this the version that was tested? Which users and decisions do the tests cover? What changes would invalidate them? Can the service restrict use and tell people what happened? Who will make the repair?

Appendix B provides a manifest, a completed example decision and a suspension-and-renewal example. Use them together. The manifest declares the promise. Tests and operating records supply the proof. Authorized people make the decision. Keep those three separate, and certification can't turn into a decorative status field.

CHAPTER 12

Meaning before the query

Before an assistant relies on anything it retrieves, it has to settle what the request means. Who is asking? Which real-world thing is involved, in what role, under which definition, for what purpose and as of which date? A context specification, owned and versioned, defines what a task needs to know. A context record holds the resolved values, and where each came from, for one actual request. The application builds it through the architecture in Chapter 7, and a separate mechanism enforces authorization. Most firms need less machinery for this than they fear. A managed dictionary with owners and effective dates carries a lot of first workflows.

Ontologies, knowledge graphs and semantic layers earn their cost when relationships, reuse across domains or consistent calculation demand them.

The right answer to the wrong question

A composite scene. An examiner reviewing a disability claim asks the assistant for the claimant's prior claims. The assistant returns forty-one. Every one is real and correctly retrieved. The examiner expected two.

Here's what happened. The claim sits under a group policy. In the policy administration system, the customer is the policyholder: the employer that bought the coverage. In the claims system, the customer is the claimant: the employee who filed. The assistant resolved "customer" through the policy record and listed every claim filed by every employee of that employer. It answered a legitimate question. Just not the examiner's. And it put thirty-nine other people's claims on a screen where they had no business being.

A month later, the same assistant assembles context for a different claim and attaches the current claims-handling procedure. The procedure is correct and approved. It also came into force after the policy date, and the request asked for the procedure that applied on the policy date. The earlier version is still sitting in the library, unchanged and valid. The assistant picked a valid document for the wrong date.

Fluency explains neither failure. The request lacked approved meaning, and the first scene also exposed a missing authorization boundary. A better model might notice the ambiguity. The organization still has to supply the right interpretation and enforce the boundary.

In this teaching scenario, the approved applicability rule uses the policy date. Real claims procedures may turn on the loss, notice, service or policy date, the product and the jurisdiction. The procedure authority decides the actual rule. Nothing here is a statement of insurance law.

Five meaning questions inside a request

Once you've established who's asking, why and what they're authorized to see, five elements pin down the business meaning. Name them separately. Each has a different owner and a different fix.

Table 16. Five meaning questions inside a request

Element	The question it answers	In the claims request
Entity identity	Which real-world thing is this record about?	The claimant, matched across the claims, policy and payment systems
Role	What part does that entity play here?	Claimant, not policyholder; insured and payee are separate roles
Business term	Which agreed definition applies for this purpose?	"Applicable procedure" means the version in force on the policy date
Metric	Which calculation, grain and filters?	Days to first payment, counted in business days from the notice date
Time	True when, and recorded when?	Policy date selects the procedure; data as recorded when the examiner asked

Identity, role and term are where the forty-one claims went wrong. Time is where the procedure went wrong. Metrics matter when the assistant calculates instead of retrieving. They're also where two dashboards and an assistant most often disagree.

Identity and meaning are different problems

Entity resolution links records thought to describe the same real-world thing. Some links follow managed identifiers and fixed rules. Others are probabilistic judgments that need thresholds and review. Keep the method, the basis for the match, the version and any human confirmation. Unless you've validated it, don't present a match score as a calibrated probability. Provide a way to correct and split mistaken links, and to find the downstream results a change affects.

A shared identifier still isn't a shared meaning. The same person can be the insured on one policy, the claimant on one claim and the payee on another. Roles belong to a relationship and a period, not to the person. An assistant that only knows the identifiers match will treat every role as interchangeable, and that's how an employer's claims history turns into an employee's. A shared customer key is valuable. It doesn't mean every domain de-

fines customer, exposure or active status the same way. Manage the legitimate differences; don't pretend they aren't there.

Agree the meaning, and who can change it

Many business terms have more than one valid definition. An open claim for reserving and an open claim for service-level reporting can legitimately differ. You don't need one definition for every purpose. You need each definition explicit, owned and tied to the purpose it serves.

That replaces the slogan of a single source of truth with something more useful: authoritative sources for stated purposes and dates, plus a rule for what happens when they disagree. At the insurer, policy administration is authoritative for the policy date and coverage terms. The claims system is authoritative for the notice date, status and contact details recorded at notice. When the two disagree about who a party is, the assistant doesn't merge them. It flags the unresolved match and shows only the conflicting facts the examiner may see. A conflict warning must never expose a restricted record.

The conflict rule has to come from the business. Engineering can encode a definition. It can't decide which of two defensible definitions should govern a coverage review. Give each critical definition an owner who can settle disputes. When the claims workflow owner changes a rule, the change gets a version and an effective date, because answers given under the old rule still have to be explainable.

Valid time and recorded time

Valid time is when a statement applies in the business. Recorded time is when that statement entered the system's accepted history. A correction can change the system's account of an earlier period without changing when the correction itself was recorded. Keep both. An investigation may need either the corrected account or the account an earlier request actually saw.

A bitemporal design keeps both dimensions: the interval in which a fact applies in the business, and the interval in which that version was recorded as known. A correction keeps the earlier recorded view and establishes the revised one. Two timestamp columns don't give you that on their own. You have to define interval boundaries, time zones, correction rules and the queries that answer both "what applies now?" and "what did we know then?"

Here's why it matters for the recurring request. The dates are invented. The insurer's procedure library holds version 4 of the disability claims procedure, in force until 31 January, and version 5, in force from 1 February. Neither document ever changes after approval.

A claim arrives in March. Policy administration has the policy date as 15 January, so the assistant attaches version 4 and the examiner works from it. On 2 April an underwriter finds a keying error and corrects the policy date to 3 February. From then on, the applicable procedure is version 5.



Figure 19. Procedure validity and the history of a corrected policy date

Table 17. Two clocks: valid time and recorded time

Question	Clock	Correct answer
Which procedure applies to this claim?	Valid time, as now known	Version 5
Which procedure did the examiner see on 28 March?	Recorded view plus the request's own record	Version 4, if the retained request and output confirm it
Which procedure applied, as recorded on 28 March?	Valid time as recorded on 28 March	Version 4
Was the 28 March answer wrong?	Both	It followed the inputs recorded then; whether the decision was adequate needs a separate review

The procedure documents never changed. The facts about the claim did, and the correction changed which document applies. Without both clocks, the team faces a bad choice. Overwrite the policy date and lose the ability to explain the examiner's earlier decision? Or refuse the correction and leave the wrong procedure in force?

Three rules follow. Store procedures and other managed documents with effective dates, and issue a new version for any material amendment. Store corrections to business facts as new recorded versions, with the time and the person or system that made them. And

with each request, record the recorded-time point the assistant read from, so a reviewer can reconstruct what it saw. Chapter 17 comes back to that record.

A recorded-time query can rebuild the source state. It can't prove that a particular request read that state, or that a person saw a particular output. Where your record-keeping policy requires it, keep the versions actually selected and the result actually returned. Event logs, warehouse time travel and backups keep different kinds of history for different lengths of time, so test the reconstruction each one really supports. Keep the rules for historical access, deletion and legal holds explicit.

Make the time boundaries executable

For a single-valued statement, use half-open intervals: valid_from is inclusive and valid_to exclusive, recorded_from inclusive and recorded_to exclusive. Where two intervals meet, the earlier one ends exactly where the later one begins. Don't subtract a millisecond to fake an inclusive end. Represent an open end using your database's convention, and define how queries handle it. PostgreSQL's range types show the choices. [68]

valid_from <= business_time < valid_to; recorded_from <= knowledge_time < recorded_to

That line is shorthand for the resolved historical view. It isn't runnable SQL or a schema. Translate both comparisons and the open ends explicitly, and reject invalid intervals. Where the contract needs exactly one answer per business key and purpose, test for gaps and overlapping active statements. Don't force that rule onto relationships that are legitimately many-valued.

Keep calendar dates and instants apart. A policy effective on a local business date shouldn't quietly become effective the day before when converted to UTC. For instants, keep an unambiguous UTC value plus the original offset or time zone. For calendar rules, keep the date, jurisdiction and business calendar. Test daylight-saving overlaps, local times that don't exist and period boundaries under the rule you chose.

An append-only correction design keeps the old statement untouched. It appends the correction, its accepted revision, its effective interval and a reference to what it supersedes. A projection then derives where the old statement's recorded interval ends and any replacement intervals begin. Commit related correction events together, so no reader sees half a correction. You can always rebuild a mutable projection. It isn't the authoritative immutable ledger.

Native system-versioned tables are a different implementation. They can keep an earlier row while changing the current one, which isn't the same as a ledger where nobody ever edits a historical statement. Protect history administration and deletion separately. SQL

Server's temporal periods use the transaction's begin time in UTC, and nobody should describe that as the moment another transaction could first see the commit. [69]

Define the recorded coordinate to match your database's real transaction and visibility rules. A sequence number or a wall-clock timestamp alone doesn't establish commit order across concurrent transactions or separate systems. Keep the committed revision or snapshot and the source versions each request actually used. If your sources can't give a consistent historical cut, record that limitation. Don't rebuild an exact-looking past from today's values.

Where ontologies, knowledge graphs and semantic layers help

Vendors often sell these as a stack every firm has to build. They solve different problems, and a first workflow rarely needs all of them.

A managed dictionary lists terms, definitions, owners, the purpose each definition serves and its effective dates. For many first workflows, that's enough. The distributor in this book runs its first capability on one.

An ontology expresses a domain's concepts, relationships and rules. A party may play the claimant role on a particular claim for a particular period; that's different from treating "claimant" as a permanent kind of person. Explicit models pay off when relationships drive meaning, or when several uses need a shared structure. Adopting an industry model still means a bounded scope, mapping, local ownership and tests.

A knowledge graph connects entities and relationships and may hold both vocabulary and instances. It helps with questions that cross several relationships, like every claim linked to a provider. RDF and property-graph implementations make different commitments about representation, query and reasoning. Manage asserted and inferred links, provenance and authorization. Permission to see one node doesn't authorize every path or inference connected to it. And treat the graph as optional, worth its upkeep only where relationship reasoning earns it.

A semantic layer exposes managed metrics, dimensions and calculation rules to every consumer. Agreement needs the same definition version, grain, filters, currency, calendar, source state and treatment of history. A semantic layer can host several valid, purpose-specific measures. Reconcile sample results through each engine. Don't assume a shared metric name guarantees the same number.

Table 18. Where ontologies, knowledge graphs and semantic layers help

Need	Simplest adequate tool	Step up when
Shared definitions with owners and dates	Managed dictionary	Relationships between terms start to change answers

Need	Simplest adequate tool	Step up when
Roles and relationships that drive meaning	Ontology for the bounded domain	Several domains must reuse the same structure
Questions that traverse many relationships	Knowledge graph over resolved entities	Volume and upkeep justify the matching and access-control work
The same metric in reports and assistants	Semantic layer	Two consumers already disagree on a governed number

These models do different jobs. A semantic layer can't settle a disputed business definition without an authorized decision. An ontology may supply attributes or relationships an access policy uses, and a semantic service may also enforce policy. A definition, however, grants no access by itself. Identify where the decision is made and where it's enforced, and test both at the real serving boundary.

Logical inference is not data validation

OWL 2 reasons over stated axioms under an open-world assumption. A fact that isn't recorded isn't necessarily false. Two different names don't necessarily refer to two different things. So a requirement that something exists can be logically satisfied without any named value appearing in the data. Logical consistency says nothing about whether your records are complete. [49]

Use SHACL for explicit constraints on the chosen data graph, such as a required property or an allowed value. State the shapes, targets, graph scope and any entailment regime before validating. A missing required value can fail a SHACL check while the OWL ontology stays perfectly consistent. Passing the shapes doesn't prove business truth or ontology consistency, either. Reasoning and validation give you different kinds of assurance. [50]

Shared meaning without one universal ontology

An enterprise needs consistent answers for a stated purpose. It doesn't need a single definition that erases legitimate differences. Claims can define open as needing examiner work; finance can define it as carrying an unsettled financial obligation. Those can be two managed concepts in different domain modules rather than rival versions of one concept. Give each a stable identity, definition, owner, examples and scope before deciding how they relate.

Keep domain variation separate from release history. Claims version 2 and version 3 describe changes inside one model. A finance module and a claims module describe two perspectives at the same time. The enterprise may also keep a small shared core for concepts like party, identifier, legal entity and time. Agree on that core only where a common

meaning is justified. Don't promote a convenient local definition just because someone modeled it first.

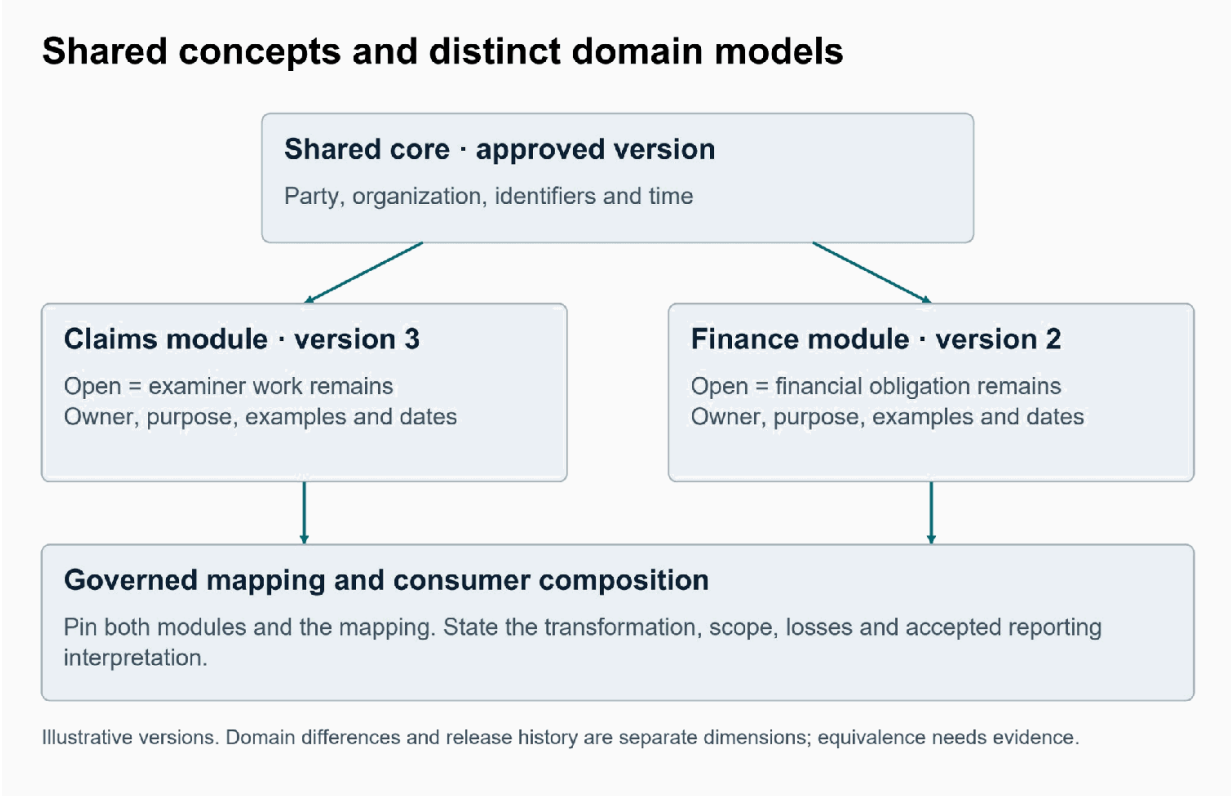


Figure 20. Shared concepts and governed domain ontology versions

The hard part is semantic reconciliation. The same label can hide different populations, grain, units, time rules, exclusions or authority. Different labels can describe the same concept. Entity matching adds a separate question: do two records identify the same thing? An agent can compare definitions and propose mappings. The owners have to decide the business consequences. Record relationships you can't resolve instead of forcing a misleading equivalence.

A mapping is a contract. It names source and target concept versions, the type of relationship, any transformation, direction, purpose, effective period, owner, examples and what gets lost. A many-to-one mapping can discard detail, and you can't assume it reverses. Reconciling two measures may need a calculation and an excluded population, not a synonym. Test the combined query against answers agreed independently.

Choose formal relationships carefully. SKOS gives you mapping relations between concepts: closeMatch isn't transitive, while exactMatch is. Neither substitutes for an executable transformation, or for asserting that two business entities are identical. OWL's sameAs says two individuals are the same individual. Its equivalentClass says two classes have exactly the same members under the ontology's semantics. Neither means "close

enough for a report". An unjustified equivalence can spread conclusions nobody wanted. [80; 49]

Release a compatible combination. Pin the shared core, domain modules, mappings, shapes and calculation rules each product uses. OWL distinguishes an ontology's IRI from its version IRI. Your deployment still has to resolve and verify the artifacts it intends to use; importing whatever document is newest won't do. Test schema compatibility, changed inferences, competency questions and the outputs consumers depend on. Adding a single axiom can break something even when no field disappears. [81]

Running versions side by side needs a migration policy. Keep supported versions visible, know their consumers and dates, and say which version answers historical questions. Retire obsolete mappings on purpose. At a regulatory report or a cross-domain decision, the owners may need one agreed reporting interpretation. That doesn't require every operational domain to abandon its own useful model. Chapter 16 shows how stewardship agents can keep the records and route the decisions without inventing a consensus.

Meaning is not permission

The context specification tells the assistant which data a request needs and how to interpret it. It doesn't grant access to that data. The systems that hold the data decide authorization, for the person asking, every time. Go back to the forty-one-claims scene. A correct context specification would have narrowed the request to the claimant. A correct authorization check would have refused the other employees' claims even with the context wrong. The two controls protect against different failures. You need both. Chapter 14 covers the second.

Semantic and ontology models can inform policy. A plausible interpretation can't authorize access. The serving and action services have to apply current policy for the verified caller, workload, resource and operation.

The context specification

This chapter's reusable artifact is a context specification for the recurring request. It's short, owned by the business and versioned like any other managed document. It says which attributes must be resolved, where they come from and what happens when they aren't available. A separate context record fills in those requirements for each real request.

Table 19. The context specification

Field	Claims request, example
Request	Assemble the context for this claim using the procedure applicable on the policy date

Field	Claims request, example
Purpose	Examiner preparation for a coverage review; not payment, reserving or customer contact
Entities and roles	Claim; claimant; policy; policyholder as context only; payee excluded
Terms	Applicable procedure: the claims procedure version in force on the policy date
Dates	Policy date in valid time selects the procedure; data as recorded at request time; the answer states both
Authoritative sources	Policy administration for policy date and coverage; claims system for notice, status and contacts; procedure library for versions
Conflict rules	Designated authority governs the policy date; an unresolved party match is flagged, never silently merged; only permitted facts are shown
Exclusions	Unrelated employees' claims; prior-claim history needs a separately authorized task; medical detail outside preparation scope
Owners	Head of claims operations for meaning; procedure owner for the library; data product owners for each source
Version	1.2, effective on approval; changes reviewed by the meaning owner

A context record for one claims request

Here's a sample record. It shows what the application resolves and keeps; it isn't a JSON payload to paste into a prompt. Caller and workload references can stay opaque to the model, and the model-facing view contains only what the approved task needs. The specification version identifies the requirements applied. The record revision identifies the resolved state this request used.

Table 20. Example context record for examiner preparation

Record field	Example resolved value or rule
Record and request	Context CR-EX-01, revision 1; request EX-01. Teaching identifiers; related records share a correlation reference.
Specification	Claims preparation specification 1.2; approved task and field requirements resolved by the application.
Caller and workload	Verified examiner reference and application workload reference. Protected assignment records establish the permitted case scope.
Purpose and subject	Prepare the selected claim for examiner review; claim and claimant resolved separately from caller, policyholder, insured and payee.
Organization and scope	Accepted product line, region and assigned case. No expansion to other employees' claims under the same employer policy.
Meaning and time	Approved procedure applicability rule; the policy date; the recorded-time view and source observation times; status freshness checked against its contract.
Origins and trust	Identity from the verified session; assignment and case facts from authoritative services; meaning from approved definitions; task clarification from the user. Origins stay distinct.

Record field	Example resolved value or rule
Policy and readiness	Policy-result references and check times kept. Ready only for the bounded preparation task; every later read or action enforces current policy.
Model-facing view	Approved task, permitted role labels and facts, selected procedure passages, dates, citations and relevant gaps. No credentials or unrelated personal profile.
Change and records	A change of subject or purpose triggers revalidation and a new revision. The record used is linked to the response, tool outcomes and retained records under access and retention rules.

Persona and communication preferences are separate fields. A requester may prefer a short management explanation or a detailed finance reconciliation. Someone may sound worried or impatient. Those observations can shape presentation and prompt a clarifying question. They can't establish an employee ID, a medical condition, organizational authority or a financial fact. Where the system inferred a preference, keep it distinguishable from one the user stated.

The record isn't a permanent dossier. Reuse a profile attribute only under the approved purpose and freshness rule. Keep records under their own retention policy, and keep irrelevant personal information out of the model's view. A conversation summary helps continuity. A new question about another employee or another engagement still needs its own scope check.

The specification doubles as test material. Does preparation stay within the selected claim? A separate prior-history task has to resolve the claimant, authorize each returned claim and exclude unrelated employees under the employer's policy. After a policy-date correction, does the preparation answer pick the newly applicable procedure and state its time basis? A narrower preparation scope never inherits authority for history.

Test the same wording from an examiner, a claimant and a finance reviewer. Test a valid employee ID typed in by someone else, a stale team assignment, a switch from the caller's own record to another employee's, an ambiguous engagement and a changed reporting period. Check that a forceful tone, or a document saying "you are an administrator", can't widen the scope. The permitted output should change with verified context. A missing essential attribute should produce the defined clarification or restriction.

The distributor's version

The distributor's first capability answers order questions for service agents. Its meaning problems are smaller, but real. "Customer" can mean the bill-to account that pays or the ship-to location that receives. "Late" can mean past the requested date or past the promised date. A credit hold can be placed by the system or by a person, and in this example the credit team may release only its designated category.

A dictionary of about thirty terms covers it, kept in a controlled spreadsheet the chief operating officer approves. Each term has a definition, an owner, a source system and an effective date. Order dates carry both the date promised and the date the promise was recorded, because promises change. An ontology becomes worth considering when explicit relationships, constraints or reuse justify it. A second workflow is one possible trigger. It isn't a rule.

What the source papers add

Context Before Query treats context as a validated prerequisite at runtime, separate from authorization. Data Is Not Just Code goes deeper into grain, source authority, time and controlled change. [8; 17] Together they're the foundation for the context specification here.

Chapter 10 describes the producer's promise. The context specification says what this consumer needs that promise to mean, for which purpose and as of which date.

Before you rely on an answer

- The caller and workload are verified, and the requester and subject are kept distinct.
- The context record identifies the specification version, where each attribute came from, unresolved fields and the permitted model view; material changes trigger revalidation.
- The request names its purpose, and the context specification says what's excluded.
- Entities are resolved by a recorded method, and roles are explicit for this request.
- Each governing term has an owner, a purpose and an effective date.
- Authoritative sources are named for the purpose and date, with a conflict rule set by the business.
- The request names its clock, and the answer says which one it used.
- Managed documents are versioned, and corrections create new recorded versions.
- Known-answer tests cover role confusion, a correction across an effective date and a source conflict.
- Authorization is enforced separately, at the data, for the caller.

PART 5

Agents you can answer for

CHAPTER 13

Give an agent a bounded operating contract

Before an agent changes a business record, split the action into four steps: propose, approve, submit and confirm. Give the action a lasting operation identity. Tie the approval to its exact content and preconditions, and enforce the change inside the target system's own transaction. A successful submission doesn't prove the change happened. A timeout can leave the outcome unknown, so the target has to support a safe way to find out and, where allowed, to retry.

People often treat human-in-the-loop as a checkpoint, where someone approves the output at the end. I treat it as an operating discipline with three points of contact. People set the boundaries at design time: the task, the sources the agent may use and the actions it may never take. They watch for drift while it runs, such as a changed source, a superseded document still being retrieved or a rising rate of overrides. And they step in where the consequence or the uncertainty of a decision calls for it. How closely people stay involved should follow the cost of an error. How new the technology is doesn't matter. Autonomy isn't the goal. The outcome is, at a level of risk the owner accepts.

Relevance belongs in that discipline as much as accuracy. A procedure, a policy interpretation or a strategy paper can be correct as written and wrong for today. Every source the agent may use needs an owner, an effective date and a status of current or retired. Retired material has to leave the index; carrying a warning isn't enough.

Tie the agent to the validated request

The agent receives an approved task and a model-facing context view from the assembly path in Chapter 7. The execution services hold the full context-record reference and current policy information. A model-generated tool argument can propose a target or a filter. The tool service then checks it against the verified caller, workload, subject and permitted operation. Passing in a record identifier doesn't prove access is allowed.

A request to change the subject, widen a list of engagements or move from explaining to writing changes the scope. It isn't just another turn in the conversation. Revalidate the context, and get the approvals the new operation needs. Take a later internal-note capabil-

ity. Its approval has to name the exact claim, note content, relevant state and expiry. The read-only preparation record can't authorize that write.

The action succeeded and the confirmation didn't

A composite scene. The claims assistant has earned permission to create an internal note once an examiner approves it. It drafts the note, shows the source references and waits. The examiner approves. The claims service accepts the update. Then the connection drops before the assistant gets the confirmation.

The screen says the request failed. The examiner clicks again, and the assistant tries again. Now the claim has two notes. In a later release, the same pattern could hit a payment or a customer notification. The writing was fine. The business operation wasn't under control.

A timeout leaves the outcome uncertain. It tells the caller that confirmation didn't arrive. It says nothing about whether the target made the change. Before an agent gets authority to write, the team has to decide how that uncertainty gets resolved.

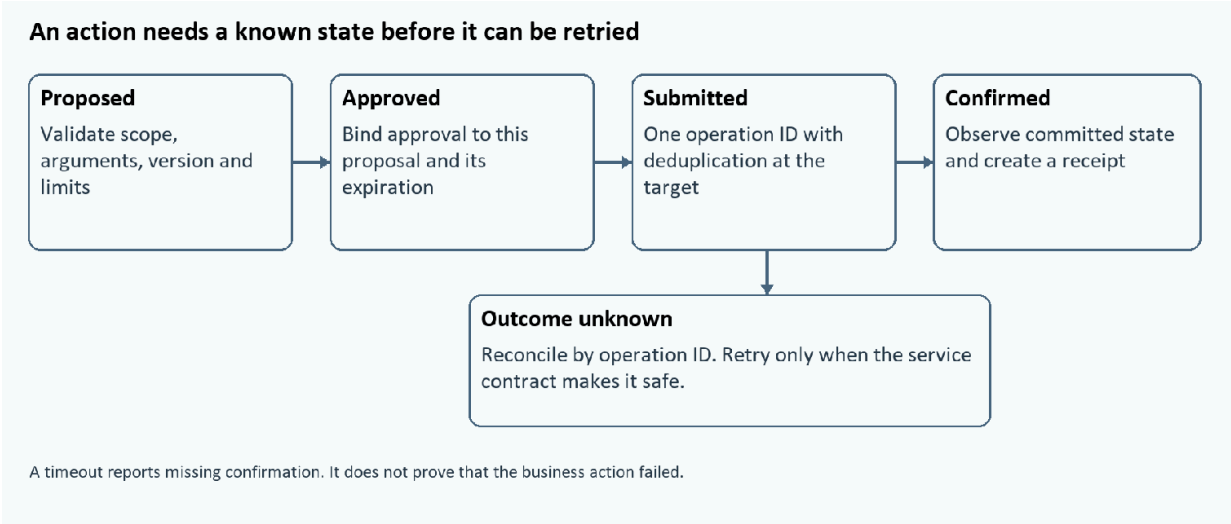


Figure 21. Action states, approval and uncertain outcomes

Treat the agent as a product with a contract

Most talk about agent risk focuses on what the model might say. In production, the bigger question is what the system around the model may do. The model proposes. Everything that turns a proposal into a business effect has to pass through services governed by the firm's contract and enforcement design, approved suppliers included. Those services need a contract, just as a data product does.

The contract starts with ownership. A business sponsor answers for why the agent exists and accepts its residual business risk. A service owner keeps it running, responds when it

fails and decides when to restrict it. In a small firm, one person can hold both jobs. Name both anyway.

Then the agent gets its own identity: an attributable workload identity. Not a shared credential, and never the full rights of whoever happens to be using it. That identity is part of what the action service checks and what the records capture. Disabling it is a containment action. Existing sessions, issued tokens, queued work and downstream credentials each need revocation behavior you've tested.

The contract lists the tools the agent may call. Nothing else. Each tool has its own contract: what it changes, which inputs it takes, its limits, and how it reports success, failure and uncertainty. A tool that creates a note and a tool that issues a payment are different tools, even when both call the same claims system.

Finally, the contract names the version. An agent is a composition: model, prompts, routing, retrieval, memory where it applies, tools, source data products, policies and the user interface. Change any of them and approved behavior can change. A certified data product behind a certified tool doesn't make every agent that calls it production-ready. Approve the whole composition, for a stated scope.

Table 21. The agent's contract

Field	Claims assistant, release 3, example
Task	Assemble claim context for examiners; propose internal notes for examiner approval
Excluded	Payments, reserve changes, coverage decisions, customer contact
Business sponsor	Head of claims operations
Service owner	Claims platform team lead
Identity	Dedicated service identity acting for the signed-in examiner
Tools	Read claim, read policy, read procedure library, create internal note
Limits	One note per approval; notes only on claims assigned to the examiner; 200 writes per hour across all users
Lifecycle state	Managed sandbox for writes; certified for reads
Version	Named combination of model, prompts, index, tools and policy versions in the release record

Lifecycle states give everyone one vocabulary for how far the agent has come. The Agent Onboarding paper uses proposed at intake, then sandbox, managed sandbox, certified, degraded, deprecated and retired. [12] Chapter 11 explains how to separate delivery stage, assurance scope and operating condition beneath those combined labels. A catalog entry must never widen the authority recorded with the state.

Secure an agent the way you'd secure an employee. Training and policy written for people don't enforce an agent's boundaries. Give each agent an identity, a role, a manager and a monitored record of what it does. Poor data classification can lead to disclosure or a wrong action. How bad that gets depends on how much authority the system allows.

Tie approval to one specific proposal

The examiner should approve the exact operation: the tool, the claim identifier, the note's content, the relevant record version and any limits. The approval should expire. If the claim or the proposal changes materially, the service rejects the old approval or asks for a new one. A blanket approval to "help with this claim" is far too broad to control a consequential action.

The action service checks those conditions when it executes. The model can't turn a retrieved instruction into a new privilege. If the policy service is down, the action waits or gets denied, as the contract says. The application explains the failure without exposing restricted information.

Delegate narrowly and check state at execution

The assistant acts for the examiner. It should never hold more than she could do, and usually it holds less. Its effective rights are the intersection of three things: what the examiner is entitled to, what the agent's role allows and what the approved task needs. Never the union. Even if the examiner can update any field on the claim, the agent acting for her can only create an internal note. That's all its contract allows.

Make delegation explicit and short-lived. The action service should know whom the agent is acting for, under which approval and until when. The agent mustn't pass that delegation on to another agent with a wider scope. It should never handle the examiner's own credentials.

A long-running or scheduled task can't assume the session that started it is still valid. Define how delegated work expires, renews, gets reassigned and gets cancelled. If the task has to carry on independently, give that background workload its own approved purpose and limited authority. Never promote a user's expired delegation into permanent rights for the agent.

Approval alone isn't enough, because the world changes between approval and execution. A colleague updates the claim. Someone reassigns it. The examiner loses access. So the action service checks the preconditions at the moment it executes: the claim version the examiner reviewed, the claim's status, her current assignment and a fresh authorization decision. If anything has changed, it rejects the approval and asks for a new proposal.

Approvals also expire, say thirty minutes after approval or at the first change to the claim, whichever comes first.

The state check and the change have to sit inside one concurrency boundary that the target system enforces. Read an approved version, let go of the connection, then issue an unconditional write later, and you've left a race. A version condition, appropriate locking or another native transaction design has to protect the real invariant. The next section makes those options concrete. None gives a universal guarantee across services that don't take part in the protocol. [70; 71]

Protect the state check inside the target's transaction

An optimistic design makes the change conditional on the approved version. A note service, for example, can advance the claim's guard version only where the claim, tenant, permitted state and expected version all still match. Require exactly one guard row to be affected. Otherwise abort and fetch fresh context. Insert the note, consume the scoped operation identifier and record the result in the same transaction. Every writer that changes the guarded state has to advance the version. [70]

Protect the operation identifier with a database uniqueness constraint within its intended tenant and operation scope. Bind it to the approved payload, or to a protected digest of it. A duplicate with the same intent gets the stored result. The same identifier with a different intent gets rejected. Keep enough to resolve the result if an acknowledgment is lost. A check in a separate cache can't give you this atomicity on the target side.

The locking alternative selects the relevant row FOR UPDATE, checks the state while holding the lock, makes the change and commits it all together. Keep that transaction short. Never hold a lock while waiting for a person or a language model. A row lock doesn't protect a row that doesn't exist yet, every related record or an external authorization service. For the wider invariant, use constraints, suitable predicate or domain locking, or serializable transactions. Handle deadlocks and serialization failures by retrying the whole transaction with fresh checks. [70; 71]

Be precise about what this guarantees. These techniques coordinate writes that take part in the target database's protocol. They can't atomically lock another service's entitlement state, or stop an out-of-band writer from breaking the rules. The action service still has to implement the accepted revocation and delegation design, including current checks or short-lived credentials. Test competing writers, stale approvals and source changes. Don't claim you've eliminated all race conditions.

When the policy service can't be reached, the contract decides in advance what happens. For writes, the answer is almost always to wait or deny. Reads may continue from content

that needs no authorization decision, if the contract allows that degraded mode. Tell the examiner, though, that some sources couldn't be checked.

One intended operation, one identity

Give the approved intent a lasting operation identifier. Within a defined caller and workflow scope, the action service uses it to recognize a retry. It records the request parameters and the business outcome. If someone reuses the identifier with different parameters, reject it. Don't treat it as a new instruction.

The target has to tie deduplication and the business change together reliably. Writing a token to a cache before a separate write doesn't make the write safe. The design has to cover failures between the steps, concurrent requests and how long operation records are kept. Where the target doesn't offer that contract, find out what happened before retrying, or hand the uncertain operation to a person. Idempotency is an established distributed-systems technique; see [22].

A transactional outbox makes a local state change and its outbound intent durable in one local transaction. It doesn't make a remote business effect atomic with that transaction. A relay can deliver the intent more than once, so the destination needs its own duplicate protection and durable result. Describe the guarantee at each boundary. Calling the whole distributed workflow "exactly once" overstates it. [72]

Keep local durability separate from remote confirmation

When a remote system owns the effect, the producer can commit its approved local intent and outbox record together. A relay submits the record under the stable operation identifier. A crash after sending but before marking it delivered can cause a second submission. So the receiver needs a durable deduplication record tied atomically to its own business effect and receipt, or an equivalent native idempotent API. [72; 22]

Suppose the destination commits and the response is lost. The producer records "unknown" and reconciles using the same operation identity. A retry under the same identity is acceptable only within the destination's documented rules and retention window. A fresh identity is a new intent and may duplicate the effect. And a status that's temporarily missing from an eventually consistent lookup doesn't prove nothing happened.

A payment service may depend on another provider beyond its own database. A unique row in the local payment table won't stop two downstream charges. Follow the duplicate protection and status guarantees all the way to the system that moves the money, including late arrivals, concurrent requests and retention expiry. If that route can't establish a safe outcome, stop automatic retries and use authorized reconciliation.

Test each crash point. Before the local commit. After the local commit but before the relay. After the remote commit but before the acknowledgment. After the acknowledgment but before the producer records completion. You should see one intended effect within the stated contract, a durable status you can recover and no silent loss. Delivery attempts may repeat. An outbox alone gives you neither universal exactly-once effects nor guaranteed completion through an indefinite outage.

Table 22. One intended operation, one identity

Contract field	Example value or behavior
Business scope	Create an internal note on the approved claim; no payment or customer contact
Identity and delegation	Named service identity acting for the examiner within the approved scope
Approval binding	Claim ID, exact content, record version, limits and expiry
Operation identity	Stable ID for this intended write; changed parameters under the same ID are rejected
Preconditions	Current authorization, valid approval and expected claim version
Unknown outcome	Query the target by operation ID; escalate when status can't be established
Records	Proposal, reviewer decision, policy version and observed target receipt

Rollback is not compensation

Rolling back the application doesn't undo a note already stored, a message already sent or money already moved. Decide whether the business effect can be reversed, and how. A correction may need a compensating operation with its own approval and record. If reversal is impossible, make that consequence visible before granting the authority.

That's why the agent contract belongs next to the workflow design. It has to describe the business effect, the permission to create it and how the system resolves uncertainty. A tool schema describes the shape of a call. It doesn't finish the contract.

Resolve unknown outcomes through the target's contract

When confirmation doesn't arrive, the honest status is "unknown". The screen should say "not yet confirmed", not "failed". "Failed" invites the examiner to click again, and you've seen where that leads.

The action service owns reconciliation. The model doesn't. The service queries the target by operation identifier. A durable receipt confirms the change. A temporary "not found" from an eventually consistent read doesn't prove no write is pending. Retry only when the target's contract makes reusing the same intent identifier safe, or when authoritative reconciliation shows there was no effect and none can still arrive. Recheck approval, authority and state before submitting. If the target can't support a safe route, hold the operation

for an authorized person, with the proposal, approval, attempts and last response attached.

An unresolved action is ongoing operational work. Give it an owner, a visible queue and an escalation deadline. Otherwise the business either waits forever or someone creates a second action outside the controlled path.

Track every action through explicit states

This chapter produces two things: the agent contract above, and an action state model. Each transition records what justified it.

Table 23. Action states and what each records

State	Entered when	What is recorded	Allowed next states
Proposed	The model drafts a specific operation	Proposal, cited sources, version combination	Approved, rejected, expired
Approved	The reviewer accepts the exact proposal	Reviewer, time, bound fields, expiry	Submitted, expired, rejected
Submitted	The action service sends it after its checks	Operation identifier, authorization decision, policy version, preconditions checked	Confirmed, unknown, reconciled absent
Confirmed	The target returns a receipt, or reconciliation finds the change	Target receipt and resulting record version	None; any correction is a new operation
Unknown	No confirmation arrives within the time limit	Attempts, timings and last response	Confirmed, reconciled absent, escalated
Reconciled absent	The target's authoritative protocol shows no committed or pending effect	Reconciliation result, the target's guarantee, current approval and state	Submitted under the same intent ID if safe; otherwise abandoned

Rejected, expired and abandoned proposals stay visible as outcomes. Confirmed absence is a recorded reconciliation result. It can lead to a safe resubmission under the same intent, if the target's contract allows, or to an explicit decision to abandon. Escalation hands responsibility to an authorized person. It must never silently authorize a second write. Keep held operations visible until their business outcome is settled.

Test the failure after the successful write

Build a test where the target commits the update and the response is lost. Then retry the same approved intent. You should get one business effect and an outcome the operator can establish. Also test duplicate concurrent calls, an expired approval, changed arguments, revoked permission and a claim version that changed after review. Agree the fault-injection method with the target's service owner.

Critical controls need their own proof. A high average answer score can't make up for a permission bypass. Passing a finite test set tells you what happened in that set. It doesn't tell you failure is impossible. State the coverage and the remaining uncertainty in the release decision.

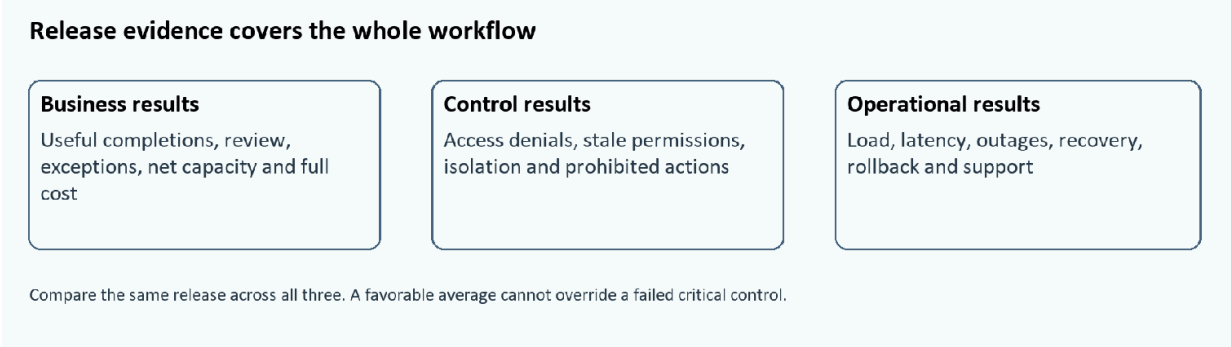


Figure 22. Three kinds of evidence for one release decision

What the owner has to decide

The business owner chooses which action is worth enabling and accepts its residual business risk through the firm's approval process. Security and control owners set the enforceable limits. The service owner demonstrates recovery and continuity. You can record those decisions together. They stay separate responsibilities.

At the insurer, the internal-note capability stays well short of payment approval. At the distributor, answering an order-status question stays separate from releasing a credit hold. Proof for one action never automatically authorizes another.

The same contract at a smaller scale

The distributor's assistant answers order questions for service agents and changes nothing. The proposed next step would let it release a credit hold below a set amount, with a credit controller's approval. The contract shrinks. It doesn't disappear. The chief operating officer is the sponsor, and the person who runs the data platform is the service owner. The agent gets its own identity in the order system. Its only write tool releases a hold, below a monetary limit, on an order the controller has reviewed.

What if the order system has no idempotency contract? A reference field helps with reconciliation. A separate check followed by a write still doesn't stop concurrent duplicates. The source service has to tie duplicate protection and the business change together atomically, or provide an equivalent guarantee you've tested within the declared scope. Until then, keep the action in an authorized manual process and send uncertain outcomes to the credit team. Approving the release itself stays with the controller. A lean firm can

combine roles, but the person who benefits from a release shouldn't be the only one who approves it.

What the reading adds

The NIST AI Risk Management Framework organizes risk work into four functions: govern, map, measure and manage. [19] It's voluntary and deliberately general. The agent contract is one concrete way to apply it to a single system. It assigns ownership, states the context and limits, tests the failures that matter and records how the system is managed when they happen.

A model change can alter which actions get proposed and how retries behave. Evaluate that behavior. And keep the authorization and duplicate-protection boundary effective even when the model proposes the wrong next step.

Start small and read-only. Keep the first release read-only unless an action is essential. Add authority when ownership, approval, failure handling and recovery are ready to carry the consequences. Time passing is not proof of readiness.

Before you enable a write

- The agent has its own identity, a named sponsor and a named service owner.
- The contract lists its tools, exclusions, limits and the approved version combination.
- Effective rights are the intersection of the person's rights, the agent's role and the task, never the union.
- The approved intent is bounded, and the reviewer can inspect the actual proposal.
- Approvals expire, and execution checks current authority and the relevant state.
- Each intended operation has one identifier, and retries reuse it.
- Unknown outcomes have a reconciliation and retry route the target authorizes; a timeout alone never authorizes a new business operation.
- Retries, uncertain outcomes and concurrent calls have tested behavior.
- Every state transition is recorded, rejections and expiries included.
- The operator can find the committed result and apply the approved recovery or compensation.
- The release record identifies the complete version combination and the owner who accepts it.

CHAPTER 14

Enforce permissions beyond the prompt

The systems that hold the data and carry out the action have to enforce permission, for the person making the request, at the moment of the request. The prompt is not an authorization boundary. Telling the model to "only use documents the user may see" expresses a hope. It doesn't control anything. Treat everything the assistant retrieves as untrusted input. Carry access changes and deletions into every copy the system makes. And test the denials as carefully as the answers.

Start with an inventory. List every agent, including the ones that arrive inside a vendor's product, with a named owner, a risk tier, its scope of authority and the person it answers to. An agent that isn't in the inventory shouldn't hold credentials.

Two ways the prompt fails

A composite scene. A claim file contains a scanned letter from a claimant's representative. Halfway down, in plain text, sits a paragraph addressed to "the AI assistant reviewing this file". It says the examiner is authorized to see the claimant's other claims and medical records, and asks the assistant to include them in its summary. The assistant reads the letter because it's part of the claim. Whether it follows that paragraph depends entirely on what the surrounding system allows.

A second composite. On a Friday, an examiner moves from disability claims to life claims, and that afternoon her access to disability files is removed. On Monday she opens the assistant. It helpfully offers to pick up the conversation she started on Thursday. The summary it shows came from a disability file she can no longer open. Nothing was retrieved again. The answer came from conversation memory, and memory had no idea her access had changed.

Better instructions to the model fix neither. The first needs a system in which no document can grant a privilege. The second needs every stored copy of retrieved content to respect the permissions of the person reading it now, not the person who retrieved it then.

Find every place a copy can hide

Draw the request path as it's actually built. The architecture slide won't show it. For the claims request, data passes through at least seven points where a permission decision has to hold.

Table 24. Every place a copy can hide

Point in the path	What it holds or does	Where the permission must be enforced
Identity	Who the examiner is and what she's entitled to	The identity provider and the firm's entitlement source
Retrieval index	Chunks of documents and their embeddings	A server-side filter derived from the verified caller
Source systems	Claims, policy and procedure records	The source's own access rules, through a service that applies them
Caches	Earlier answers and retrieved chunks	Partitioning by authorization, and revalidation or invalidation on revocation
Conversation memory	Summaries and excerpts from earlier turns	A fresh check before stored content is reused
Tools and actions	Notes now; payments or messages later	The action service, as described in Chapter 13
Output	What appears on screen or in an export	Redaction rules for the purpose and the reader

Every stored copy is a place where yesterday's permission can outlive today's. The forty-one-claims scene in Chapter 12 failed on meaning and authorization. These scenes fail on enforcement. Their fixes live in different components, owned by different teams.

Retrieval has to keep source permissions

Retrieval-augmented systems copy content out of controlled sources into indexes built for search. The copy rarely brings its access rules along by default. Several managed search services run queries with the rights of the service or the index owner, so they don't automatically apply the querying user's permissions on the original source. Anyone who can query the index can then pull passages from documents they could never open directly.

Every chunk needs the source, version, tenant and access attributes your authorization design requires. When the query runs, the service enforces the verified caller's permitted scope, either through native document authorization or through a mandatory filter the server derives. A client can ask for a narrower business filter. It can't supply or replace its own authority. A chunk missing required restriction metadata is ineligible for any protected route. Measure how long source permission changes take to arrive.

At the insurer, claim documents carry the claim identifier and its sensitivity flag into every chunk. The retrieval service checks the examiner's assignment and clearance for each claim before returning anything. A restricted claim, say one filed by an employee of the insurer itself, stays out of the results for anyone outside the restricted team. However relevant it looks.

Treat incomplete restriction metadata as an unresolved access question. Apply document and chunk restrictions before any text goes to embedding, reranking or generation services outside the approved processing boundary. Search titles, snippets, result counts, citation paths and graph relationships can leak too. And two facts that are each permitted can, together, allow an inference that isn't. Assess the combined view for the real audience and purpose, with rules for aggregate disclosure where you need them.

Name who does what. A catalog can publish policy metadata. A semantic service can apply access rules, and a gateway can run several checks. A product category, though, doesn't prove any of those controls exist. Identify which component decides, which enforces and which paths could go around them. Then test the combination you actually deployed.

Make the security filter mandatory at execution

Keep the user's optional business filter separate from the authorization filter. A user may ask for this year's claims. The server works out the tenant, the claims she's entitled to and the fields she may see from verified context. Combine the two so a broader client filter can't widen her authority. Never ask the model to write its own security clause. Never treat a group list the model supplied as trusted identity.

Close the side doors. Keep the broadly privileged search credential inside the protected service. Restrict direct access to the index, and test the alternate SDK, API and debugging routes. Any other caller can strip a filter that only the web screen attaches. In an adversarial test, remove it, swap the tenant and add an always-true business condition. The enforced scope must not budge. Where you use native authorization, test its token and administrative bypass paths as well.

Apply the security boundary before unauthorized content reaches the caller, or an unapproved reranker, model, cache, log or export. Vector prefiltering and postfiltering behave differently for retrieval and recall, and neither name proves end-to-end authorization. A postfilter that runs entirely inside an approved service can be legitimate. Sending a broad result to the model and asking it to hide the passages the user shouldn't see? That is not access control.

Azure AI Search's classic security-filter pattern compares against permission strings stored in the index. The application has to authenticate the user and supply the right val-

ues, so the pattern doesn't natively authenticate those strings. Azure also documents native ACL and Purview-label enforcement at query time, in preview, with specific indexing, identity and API prerequisites. Check which mechanism you've chosen and any elevated-read capability. Don't assume every Azure search route shares one authorization contract. [75; 76; 77]

Snowflake Cortex Search's owner's-rights behavior needs its own design. So does Databricks AI Search's documented lack of native row- and column-level index permissions. An application-level ACL filter helps only if callers can't bypass it and the indexed metadata stays complete. Retest source revocation, group changes, document updates and new chunks. Report the measured propagation window, and what happens when it exceeds the accepted limit. [26; 27]

Caches and memory are copies too

Caching answers saves time and money. A shared cache also makes it easy to serve one person's answer to another. Partition cached content by the relevant authorization context, and revalidate or invalidate it when permissions change. Or cache only content approved for the whole audience. A cache key alone doesn't keep yesterday's rights current. A procedure can be shared within its entitled audience. A claim summary needs its own access checks every time.

Conversation memory is a cache with a friendlier name. It keeps summaries and excerpts so the assistant can pick up where you left off. Before reusing any stored excerpt, check it against the user's current permissions and drop what fails. In the reassignment scene, Monday's conversation should have opened with the disability summary withheld and a one-line explanation.

Deletion follows the same path. When something has to be deleted, find every derived store it reached: chunks, embeddings, cached answers and conversation memory, plus records and backups under their approved schedules. Retention obligations or legal holds may require different handling for particular records. Settle that explicitly, without leaving an answer-serving copy reachable by accident. For each capability, keep an inventory of those stores, with an owner for each and a measured propagation time. If the team can't list the copies, it can't delete them.

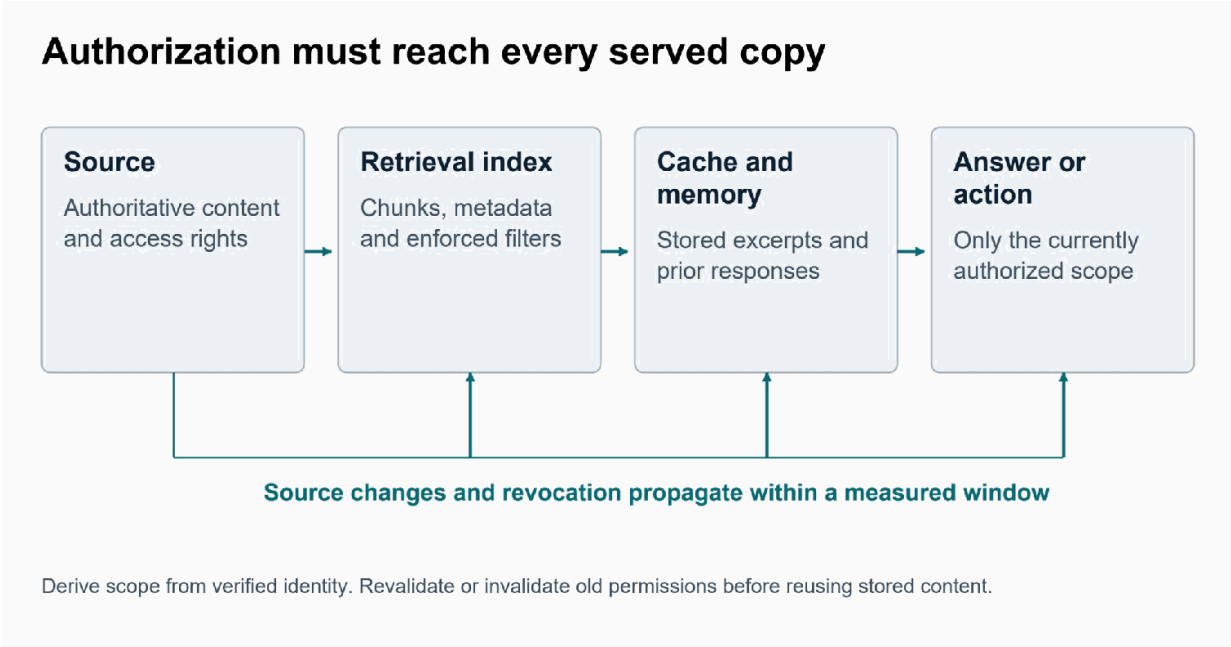


Figure 23. Access enforcement and changes across derived copies

A document must never grant a privilege

Prompt injection is the name for content that tries to behave like an instruction. The letter in the opening scene is a mild example. Others hide in web pages, spreadsheets, email signatures or text the reader can't see. Detection helps, and it's worth doing. It isn't the control.

The control is structural. Keep trusted configuration (system instructions, tool definitions, policies) separate from retrieved content, and never let the content change the configuration. Give the model no authority it could hand over. In the opening scene, the assistant may well summarize what the letter asks for. It still can't retrieve the other claims, because retrieval checks the examiner's entitlements, not the letter's claims about them. It can't send anything anywhere, because it has no tool that sends. And the action service ignores any authority asserted in content. Authority arrives only through identity, approval and policy.

Separate content from configuration in the application design, knowing that a language model can still confuse the two. Delimiters, detectors and guardrail models reduce the risk. None of them makes you immune. Enforce tool scope, output destinations and resource access outside the model. Restrict URL fetching and network egress, so retrieved text or tool arguments can't reach internal endpoints or unapproved destinations. Treat generated HTML, scripts and spreadsheet formulas as untrusted output before anything renders or runs them. [21]

That's why Chapter 13 put the execution checks in the action service instead of the prompt. Enforcement has to hold even when the model reads the letter as an instruction. Whatever disclosure and misuse risk remains still needs evaluating.

Fail closed when policy can't be evaluated

Authorization depends on services that can fail: the identity provider, the entitlement source and the policy engine. Decide in advance what happens when one is down. For restricted content and for every write, deny and explain. For content already approved for everyone in the relevant audience, the contract may allow a degraded mode that keeps the examiner working.

The explanation matters. "Some sources couldn't be checked, so I left them out" is honest and leaks nothing. A message that names the restricted claim, or confirms that a restricted document exists, is itself a disclosure. Write these messages with the security team, and test them.

Keep tenants apart

Group insurance has a tenant problem built in. Each employer's employees, claims and contacts have to stay separate from every other employer's. The distributor will face the same problem when it opens order questions to customers directly. One customer's orders, prices and credit position must never appear in another's answers.

You can separate tenants with a separate index per tenant, or with a mandatory tenant filter the server derives. Either works. Two things don't: trusting a tenant identifier the client supplies without server-side authorization, and a shared cache keyed only on the question. A client may choose a tenant only from the scopes the server has verified it may use. Test separation with two realistic tenants and an attacker who knows the other tenant's identifiers.

The threat model and the negative test matrix

This chapter produces a threat model tied to the real request path, and a matrix of tests that must fail safely. The path table above is a starting inventory, not the whole threat model. Add the assets, actors, trust boundaries, entry points, abuse cases, supplier dependencies and mitigations for your capability. Each row names who can read the data at that point, how the system enforces the permission and how changes propagate. The owner of each row signs it.

The matrix turns the threat model into tests.

Table 25. The negative test matrix

Threat	Test	Expected result	What to keep
Revoked access	Remove an examiner's access, then repeat her last question	A fresh authorization decision denies the next request, or the denial arrives within the approved lag	Denial record and measured lag
Restricted record	Ask for a restricted claim by name and by description	No content, and no confirmation it exists	Denial and the response text
Cross-tenant query	Query as employer A's examiner using employer B's identifiers	Nothing from B comes back	Query, filter applied and results
Hostile document	Add an injection payload to a test claim file	The content is summarized as content; tools and permissions are unchanged	Tool calls attempted and blocked
Stale cache	Ask the same question as two users with different rights	The second user gets only what she may see	Cache keys and responses
Conversation memory	Reassign the user, then resume the conversation	Content withheld, with an explanation	Memory items dropped
Deleted document	Delete a source document	Gone from answers within the agreed window	Propagation time for each derived store
Policy service down	Make the policy engine unavailable	Restricted reads and all writes denied with a safe message	Denials and message text
Client-supplied filter	Change or remove the access filter in the client request	The server-derived filter still applies	Request and results
Out-of-scope tool	Ask the assistant for an action outside its contract	The execution service refuses; an out-of-scope proposal has no effect	Rejected proposal or call, and no business effect

Run the matrix before release and after any change to the version combination. A good average everywhere else in the evaluation doesn't offset a critical failure here. The full set should cover every supported path in the proposed scope. A lean firm can run a smaller matrix. It can't run one that leaves out a path that actually exists.

Test the "no", not just the "yes". Show that bypasses fail. Missing context, a revoked right, out-of-scope data and a prohibited query should all be denied, and each denial recorded. Exercise source drift, stale output and a failed policy service before a reviewer does it for you.

The distributor's version

The distributor's assistant reads from one data platform that already has row-level security set up for its sales teams. That helps, provided the assistant's queries run with the requesting service agent's own rights and not through a privileged service account. The

team confirms that with one test before anything else. An agent from the eastern region asks about a western customer's order and gets nothing back.

The lean matrix has five rows: revoked access, a cross-account query, hostile text in an order note, a stale cached answer and the platform's policy check being unavailable. The chief operating officer signs the threat model, which fits on one page. When customers later get direct access, the team reruns every row with customers as the tenants. At that point the bill-to and ship-to distinction from Chapter 12 becomes a security question as well as a question of meaning.

What the reading adds

The OWASP RAG Security Cheat Sheet is a practical reference for the controls in this chapter: retrieval authorization, untrusted content and protecting the stores retrieval creates. [21] Use it as a checklist against the request path you drew. It can't replace drawing it. Chapter 15 connects these protections to DSPM, catalogs and development. Chapter 17 explains the records, because every denial or permitted use is also something an authorized reviewer may need to inspect.

Before you widen access

- The request path is drawn as built, with every store that holds a copy of retrieved content.
- Retrieval enforces the verified caller's scope through the chosen native mechanism or a mandatory server-derived filter that no supported query path can omit.
- Caches are partitioned by authorization and revalidate or invalidate revoked rights, or they hold only content approved for the whole audience.
- Conversation memory is checked again before reuse.
- Deletions and access changes reach every derived store within a measured window.
- Retrieved content can't change configuration, tools or authority.
- Policy failures deny restricted reads and all writes, with messages that leak nothing.
- Tenants are separated by the server, and the separation is tested.
- The negative test matrix passes in full for the release being approved.

CHAPTER 15

Make data AI-ready with DSPM and governed access

Data security posture management, or DSPM, is the layer that keeps discovering what data you have and how exposed it is. It finds structured and unstructured data across cloud services, SaaS applications, data platforms and the copies made from them. It classifies sensitive and proprietary content, connects that content to the identities, access and exposure around it, and helps teams decide what to fix first. Its importance for AI is practical. You can't safely open data to retrieval, agents, analytics or model development if you can't identify the data, its copies, its protection or who can reach it. The Connected Enterprise Blueprint, the Reference Implementation and the retrieval patterns paper all treat that visibility as part of one data and AI system. [2; 10; 11]

What DSPM does and doesn't do

A DSPM product connects to supported data stores and services and inventories assets. It samples or inspects content by documented methods, applies classifiers, watches access and configuration, spots duplicated or exposed data and raises findings with technical context. Mature products add identity analytics, attack-path context, remediation workflows, policy recommendations and connectors to catalogs, ticketing and security operations. Coverage, depth and authority to remediate all vary by product and connector.

DSPM observes and coordinates before it enforces anything. A finding that a column holds health information doesn't decide whether a claims examiner may use it on a particular claim. The business owns meaning and purpose. Privacy, legal and security authorities set the obligations. Identity and native platform controls enforce access. Data-product owners accept fitness for use, and the application keeps the user, tenant and task boundary. Some DSPM products can make selected changes. But an API call that writes a tag or a policy doesn't prove that every query, copy and AI route now behaves.

Treat DSPM as the sensor in a control loop. Observe the estate. Connect each finding to the managed asset and get the decision it needs. Encode the handling rule, deploy it through supported enforcement points, test the real behavior and reconcile drift. That loop makes

data more usable, because the approved paths become explicit. Protection is what makes reliable use possible. It's no reason to lock the whole estate away.

The same data supports different valuable uses

A composite design. The insurer wants three things: an examiner's case-preparation assistant, an operational dashboard and a development environment for the team improving both. Claims records and uploaded documents hold useful status, procedure and medical information. The dashboard needs service measures. The examiner needs permitted case facts, and the developers need contracts and realistic tests. Copy the same complete records into all three, and you create exposure and upkeep nobody needs. Declare the whole collection off-limits, and you throw away legitimate value.

DSPM shows what's there, where the copies are and which identities can reach them. The owners then agree the right products: a case-scoped preparation route, an approved analytical view and synthetic development fixtures, each serving a defined purpose. Protection gets built into making those products dependable. Nobody bolts it on after the business has started using them. This is a sample architecture, not a report on a client deployment.

Look past the sensitivity label

A label like "personal information" says very little about whether an examiner may use a particular record for a particular claim. The useful protection context is richer. It includes the asset and field, the business concept, the subject or tenant, the authoritative owner, source and copy relationships, approved purposes, observed access, protection state and when it was observed. Some of that is discovered; some needs a business or control decision. Never let an inferred owner or a classifier's output stand in for an approved business mandate.

Business-specific information matters as well. An acquisition model, a proprietary underwriting approach, source code or a client methodology can be highly valuable without matching any standard personal-data pattern. Combine technical findings with the ontology and purpose-specific definitions from Chapter 12. A detector can suggest that a field contains a medical identifier. The managed model explains whose identifier it is, what role it plays and which workflow needs it.

Measure discovery coverage and classification quality on your actual estate. Record the supported sources, objects that couldn't be reached, what was sampled, encrypted or unreadable files and when the observation was made. Check false positives and missed classes against examples reviewed independently. A clean scan tells you about what was observed under those conditions. It doesn't prove no protected data exists. Protect the

scanner's credentials and any samples or findings it keeps, because even metadata can reveal a sensitive business relationship.

Where DSPM fits in the stack

DSPM sits beside the catalog, the semantic layer, identity services and native platform controls. It supplies observed facts about assets, classes, copies, access and posture. The enterprise catalog supplies the product, owner, purpose and route to request access. The ontology supplies business meaning. Identity and policy services supply current authorization. Native databases, storage, applications, APIs and retrieval gateways implement the control. SIEM, SOAR, ticketing and incident systems carry findings and responses. And the control plane links all of these records without pretending one product is authoritative for everything.

Integrate in both directions. Send stable asset identifiers and observed findings from DSPM to the catalog and the security workflow. Send accepted ownership, purpose and handling decisions back through a controlled adapter. Reconcile the intended rule with the tags, grants, masking, encryption, row policies, retention and serving behavior actually deployed. Keep timestamps and provenance. A scan, a catalog sync and a runtime decision all happen at different times.

The architecture should keep three states apart: observed, accepted and enforced. Observed means DSPM or another sensor found something. Accepted means an authorized owner decided how to treat the data for a purpose. Enforced means the source, pipeline, index, API or application has demonstrated the required behavior. A dashboard that collapses those into one green light hides exactly the gaps the architecture is supposed to reveal.

Cyera as an example, not an assumption

Cyera is a useful example of the DSPM category. Its published capabilities cover discovery and classification, identity and exposure context, risk prioritization, remediation workflows and integrations. [53] In the architecture above, it can supply observations, findings and selected remediation actions. It doesn't become the system of record for business meaning, approved purpose, source-system authorization, data quality or AI acceptance, and it isn't a required component. Check the modules you've bought, connector coverage, scan method, the permissions it needs, update latency and remediation behavior for your estate.

Cyera lists its published Collibra integration as available. The integration describes enriching matched catalog assets with classification, sensitivity, ownership and access context, optionally creating missing inventory. [54] That's a good example of discovered infor-

mation reaching the place stewards already work. You still have to validate asset matching, field ownership and the destination's permissions.

Cyera's June 2026 Snowflake announcement describes tag-based masking remediation and risk analysis as generally available, and Cortex-agent discovery as a private preview planned for July. [55] Don't assume that preview has since gone GA. Its public Purview demonstration shows finding labeling gaps and applying recommended labels. [56] I checked these vendor descriptions for this edition. I didn't run them as integration tests. A demo of label application doesn't show that every file type, destination or AI route is covered.

Connect catalogs through a maintained metadata contract

The enterprise catalog tells people which product serves a need, what it means, who owns it and how to ask for approved use. A native catalog and policy service hold platform-specific objects, tags, grants and protection bindings. Microsoft Purview can span governance and information protection, but its capabilities differ by product area and source. These roles can share a platform and still need explicit interfaces and authoritative fields.

Table 26. Catalog responsibilities and the protection contract

Responsibility	Information exchanged	What must be demonstrated
DSPM observation	Asset IDs, observed classes, exposure, access context and scan time	Coverage and classifier quality; findings stay distinguishable from approvals
Enterprise catalog	Purpose, definitions, owner, product contract and accepted handling	Stable source matching and a usable route to request approved use
Snowflake Horizon Catalog and native policies	Object and field metadata, user-defined tags and policy references	Protection under real identities and query paths; edition and object support verified
Purview governance and information protection	Source metadata, mapped classifications and applicable label references	Behavior of the chosen connector and workload; catalog labeling distinguished from source enforcement
Application and development services	Approved metadata view, interfaces, fixtures and contract versions	Current caller and task scope; only permitted context reaches the IDE or model

Use stable technical identifiers that include the account or tenant, environment and object. Maintain approved mappings when someone renames, copies or recreates an object. An employee identifier in a test environment and a same-named production column are different assets. Record the DSPM observation, classifier version, accepted classification, decision owner, effective time, propagation status and last reconciliation. A scheduled connector synchronizes. It doesn't guarantee real time.

Assign authority field by field. The scanner supplies observed findings. The business owner approves meaning and purpose. The protection authority approves handling, and

the platform reports what's actually enforced.

Keep provenance when they disagree. Don't let two tools overwrite each other in a loop, or a classifier quietly lower an approved protection level. A correction route should handle over-classification as carefully as under-classification, with affected consumers identified.

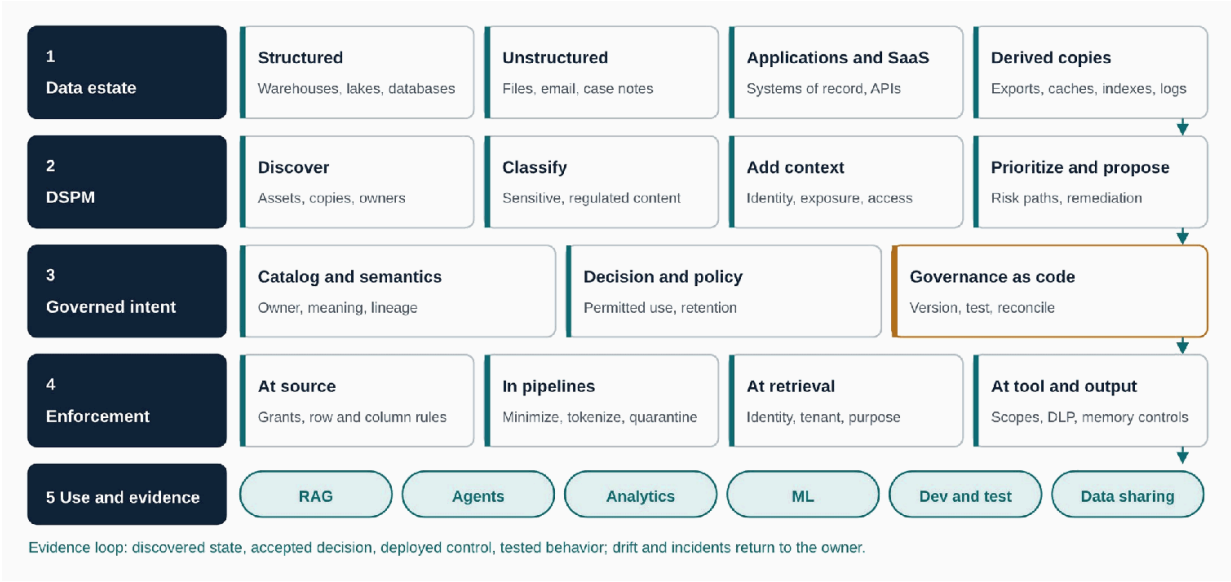


Figure 24. DSPM in the governed AI data path

Integrate DSPM into security operations

DSPM should plug into controls and workflows you already run: identity governance and privileged access, cloud and SaaS configuration, key management and encryption, database and storage policy, data loss prevention, API gateways, security monitoring, case management and incident response. Use those connections to route findings, get decisions, deploy supported fixes and verify the result. Don't give the scanner broad standing authority to change production access because it spotted a pattern.

Prioritize by the harm that could actually happen and by business need. Classification alone is a poor guide. Combine sensitivity and commercial value with exposure, active identities, privilege, internet reachability, copy sprawl, approved purpose, how critical the consumers are and the realistic path to misuse. A highly sensitive product with narrow, tested access may be less urgent than a forgotten copy reachable by a shared identity. And a low-severity label can still matter when an agent can combine many records or traverse relationships at scale.

Design for protection and useful access together. The response to a finding might be to remove a stale copy, narrow an identity, mask a field, publish a managed view, tokenize a value, correct an over-classification, isolate an index, add a purpose check or build a supported API. Record the owner, due date, interim state and how you'll verify the fix.

Security posture improves when the approved route becomes easier than the shadow route.

Enforce protection in Snowflake Horizon and Microsoft Purview

Snowflake tag-based masking links a tag to a compatible masking policy, and the policy controls the value returned. Row access policies separately decide which rows a query may return. Both need an eligible edition, Enterprise Edition or higher, and supported object and policy configurations. A classifier's observation contributes nothing until there's an approved mapping and a deployed binding. [57; 58; 73]

Test the role context deliberately. CURRENT_ROLE identifies the primary role. IS_ROLE_IN_SESSION accounts for active primary and secondary roles and their hierarchy. A warehouse is compute. It doesn't substitute for the user's authorization scope. Row-policy evaluation uses the policy owner's privileges on referenced objects, and that doesn't mean the caller inherits them. Inspect session functions and policy behavior through the actual query and service routes. [73; 74]

A row access policy filters rows. It doesn't replace grants and controls on inserting or changing business records, so keep Chapter 13's write invariants at the action boundary. A query run through a broadly privileged shared identity also needs the application's approved user and purpose restrictions. Native policy names don't establish those.

For the example claims dashboard, the product owner approves a view with the measures it needs and protected identifiers. The deployment creates the versioned view and policy bindings. The tests run as the intended analyst, support and service identities. Test joined, copied and derived paths as well as the original table. Masking an identifier doesn't necessarily anonymize the rest of the record, or prevent a revealing inference.

Protect at the serving layer. Asking the model to behave isn't protection. For a question about quarterly spend, expose an approved aggregate view with the currency, period and entity scope defined, and drop unnecessary account identifiers at the serving layer. Apply field protection there when detail is permitted. Don't rely on an MCP response that tells the model to remember to mask fields in the SQL it writes. And be careful with "anonymized" identifiers. An unsalted SHA-256 hash of a predictable identifier isn't proof of anonymity. Choose suppression, tokenization or a reviewed keyed pseudonymization design for the purpose, with protected secrets or mappings, linkage analysis and access controls. Pseudonymized data can still identify people, so assess it accordingly.

Keep a Purview Data Map classification or catalog label distinct from protection on a file or source system. Microsoft's source matrix lists Snowflake scanning and Data Map labeling but no policy application for that source. [59] The catalog entry can't stand in for

Snowflake enforcement. Use the native protection path, and reconcile what's deployed back to the governance record.

When configured for supported content and workloads, Purview sensitivity labels can apply encryption and usage rights. They can also be purely descriptive. A Data Map classification, a sensitivity label on a supported file and a native database policy are three different mechanisms. Verify the workload, label publication, identities and permission changes. Don't assume a label protects every extracted passage or every new index. [60]

Azure AI Search documents Purview sensitivity-label enforcement as a preview, with specific permission, identity and indexing requirements. Assess that route separately from ordinary label discovery. A pipeline that strips protection metadata, or queries through an overprivileged identity, can break the intended boundary. Keep unknown required protection closed at the serving surface, and test the content that's actually exported and retrieved. [77]

Use DSPM across the AI-ready data lifecycle

AI-ready data is findable, meaningful, fit for the intended use, reachable through an approved route, protected along that route and observable after release. DSPM supplies discovery and posture. Data products, semantics, quality controls, identity and runtime policy complete the readiness decision. One dataset can support RAG, an agent tool, analytics, model training, software development or an external API through different purpose-specific products. Each route needs its own minimized data view, enforcement points and records.

DSPM helps in many places. It can help select an authoritative corpus for RAG, find sensitive fields and copies for analytical products, analyze identities and exposure for agent tools, and review training data. It can produce synthetic or masked fixtures for AI-assisted engineering, inform entitlement-aware knowledge graphs and support managed sharing through APIs or clean rooms. It shows where protected or redundant data sits and who can reach it. By itself, it can't establish semantic correctness, data quality or model suitability.

Table 27. Protection across the book's implementation patterns

Use case	DSPM and governance contribution	What runtime has to show
RAG and enterprise search	Find and classify source documents and derived indexes; connect them to owner, purpose, tenant, retention and entitlement metadata	Only eligible sources enter the corpus; current identity and purpose filter retrieval before reranking and model context
Agents and tool use	Identify sensitive fields, service identities, exposed copies and approved data products for each tool	Every tool call rechecks user, workload, tenant, purpose and action scope at the authoritative boundary

Use case	DSPM and governance contribution	What runtime has to show
Analytics and BI	Map sensitive elements and copies to approved measures, views and export handling	Queries, extracts, joins and small-group results keep row, column and purpose controls
Model development and training	Locate candidate data, restrictions, duplicates and exposure; connect to provenance and retention decisions	The approved dataset, minimization, quality, rights, environment and deletion obligations are tested and versioned
AI-assisted development	Publish approved contracts and protection metadata while keeping production content and credentials out of the IDE path	Developers and coding agents get permitted metadata and synthetic or masked fixtures through scoped identities
Data sharing and APIs	Expose accepted classification, residency, purpose and downstream handling requirements	The interface returns a purpose-specific product; consumer identity, contract, revocation and onward obligations are enforced
Knowledge graph and federation	Relate discovered assets to entities, roles, sources and protected relationships across platforms	Traversal, inference and combined results stay within the permitted task and audience in every engine involved

In four-part terms, DSPM contributes protection attributes to the brain, alongside managed meaning and context assembly. The nervous system's narrow interfaces carry the approved data view and current authorization. Findings feed the immune system's policy and monitoring. The body's deployment and operation keep the protection in place. The control plane links observed posture, decisions, code versions and records. DSPM takes part in that system. It doesn't replace any part of it.

Filter RAG before content reaches the model

A RAG pipeline creates a new serving surface. It copies or indexes source content, splits documents into chunks, adds embeddings and metadata, retrieves candidates, reranks them and places selected passages in the model's context. Every step can weaken the source boundary if ownership, classification, tenant, purpose, retention or entitlement gets lost along the way. Redacting output is a last safeguard for particular cases. It can't substitute for keeping an unauthorized passage out of retrieval and out of the model's context in the first place.

At ingestion, admit only approved source versions. Record the source asset, document, owner, authority, classification, tenant or subject scope, approved purposes, retention, legal restrictions, effective time and deletion route. Chunks inherit the source's handling, and you can treat them more strictly when their content calls for it. Quarantine anything whose required protection or authority is unknown. A vector index is a maintained copy, with its own owner, access path, refresh, revocation and destruction obligations.

At retrieval, evaluate the current human and workload identity, role, tenant, case or portfolio, approved purpose and policy version before any candidate passage comes back. Apply native source permissions or a validated authorization filter to both keyword and

vector retrieval, and rerank only the permitted set. If metadata can change mid-request, check again before assembling the model's context. Never rely on a prompt instruction, a tenant name supplied by the client or a broadly privileged shared service identity for this boundary. A gateway can enforce provider, credential, budget and egress controls within its reach. It can't make an unauthorized retrieval safe after the fact.

Carry the boundary through tools, memory, caches, traces and outputs. A permitted answer can still leave an unapproved durable copy in conversation history, an evaluation dataset, a support log or an agent's memory. Keep as little as possible. Encrypt where required, separate tenants, protect keys and restrict administrative access. Apply loss-prevention or label controls to supported outputs, and test how deletions and revocations propagate. Treat retrieved text as data, never as instructions, as Chapter 14 requires.

Keep enough to reconstruct the decision without keeping every protected passage: request and workload identity, purpose, policy version, corpus and index version, source or chunk identifiers, filters applied, denied candidates where appropriate, selected sources, model and prompt version, tool calls, output handling and test result. Sample the real end-to-end path. A catalog label or a successful connector run doesn't show that the retriever, cache and model gateway enforced anything. [11]

Make governance as code an operating loop

Governance as code means versioning the machine-readable intent, validating changes to it, deploying supported controls and comparing what's deployed with what was decided. A document in Git is useful. It becomes an operating control only when the live path depends on the implemented rule. Use a defined schema, approved policy semantics and a deployment identity that can't change its own business mandate.

The pseudoconfiguration below sketches a contract for an analytical claims product. It isn't Cyera, Snowflake or Purview syntax, an executable authorization policy or a production schema. The references are symbolic. A real implementation has to resolve protected versions and run the listed tests.

```
schema: book.example.protection.v1
product_ref: example:claims-service-metrics-v2
source_ref: example:claims-prod-approved-view
meaning_ref: example:claims-service-definitions-v3
observation_ref: example:dspm-review-017
approved_purpose: operations_service_measurement
protection_ref: example:claims-analytics-policy-v4
development_data: synthetic
model_input_ref: example:permitted-metrics-view-v2
unknown_protection_state: require_review
required_evidence:
  - expected_measures_match_approved_cases
  - intended_role_receives_approved_view
  - different_tenant_receives_no_case_data
  - direct_identifier_is_protected
  - derived_copy_preserves_approved_handling
  - current_authority_checked_at_execution
decision_ref: example:product-acceptance-021
```

Translate that intent through supported interfaces into the views, policy bindings, index attributes or service checks it requires. Make sure the adapter keeps the rule's meaning. If a destination can't enforce a required condition, publish a narrower product through a controlled service, or leave that route unapproved. Never report protection as done because an API accepted a metadata update.

In review, show the owner a meaningful change. Which assets, purposes and users are affected? Which protections change, which consumers depend on them, and which tests prove the result? A classification can trigger a review or a pre-authorized protective action. Widening permitted use needs the right authority. Keep approvals, code and deployment linked. Flipping an "approved" flag in the artifact approves nothing.

After deployment, reconcile intended and actual tags, grants, policies and serving behavior. Catch manual changes, missing attributes, renamed assets and failed connector jobs. A rollback has to respect current authority and source obligations. Measure the time from discovery to accepted correction to demonstrated protection, with a defined operating response while a finding is open. That's how governance stays current between formal reviews.

Bring approved context into Cursor and other IDEs

Developers need the product contract, allowed interfaces, definitions, protection requirements and examples before they generate a pipeline or a tool. Publish an approved development view of that metadata through a protected API or MCP service. A proposed interface might return a contract, its protection plan and synthetic fixtures for a named product. These are design examples. I'm not claiming Cyera ships a native Cursor connector.

Cursor documents project rules in .cursor/rules and connections to external tools through MCP. [61; 62] Rules are instructions to the model, not a data-access boundary. Use a project rule to point the coding assistant at the approved contract and tests. The metadata service still has to verify the developer or workload, the project scope and current permission on every request. A copied rule, or a project name the model made up, can't create access.

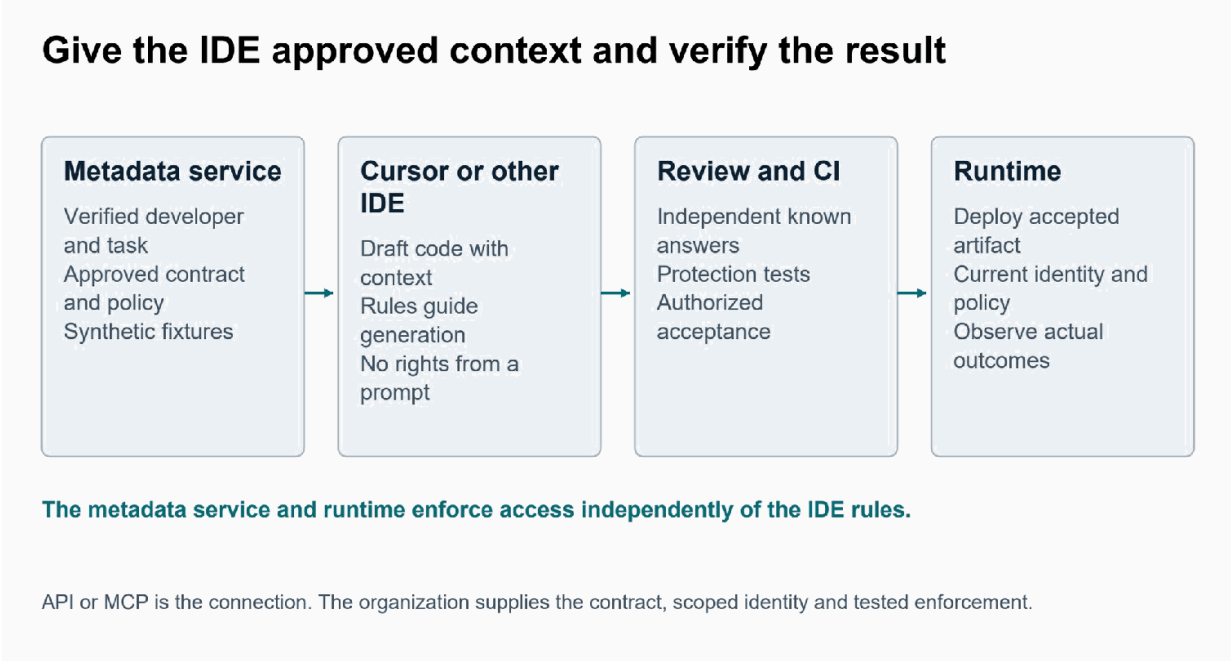


Figure 25. Governed development from metadata to runtime

The developer first picks the authorized product and task. The service returns the permitted metadata view and its version, with no production samples or credentials by default. The assistant drafts code against the supported interface and the synthetic fixtures, and reviewers inspect the proposed code and policy changes. Automated checks validate the contract, known answers and protected paths. An authorized deployment then applies the accepted combination. Runtime services keep enforcing their own rules even if the generated code ignores the project rule.

Metadata can be commercially sensitive in its own right. Table names, medical field descriptions, acquisition projects and grant relationships can all reveal protected activity. Make a separate decision about what reaches the IDE or the model provider. Treat code indexing, local files, terminal output, plugins, MCP tools and cloud-agent workspaces as data paths. Cursor's hardening guidance lists controls to assess for your plan and configuration. [63] An ignore file or a privacy toggle doesn't complete your information-handling design.

Connect generated work to the secure software lifecycle

Treat model-generated code as a candidate contribution to the controlled lifecycle described in Chapter 19. The crosswalk below applies NIST SP 800-218, the Secure Software Development Framework version 1.1, to the work in this book. The examples of what to keep are my suggested applications of those practices. They aren't additional NIST requirements or a statement of compliance. [52]

Table 28. Secure development practices for generated and handwritten work

Delivery responsibility	SSDF v1.1 practices	What to keep
Requirements and authority	PO.1, PO.2	Security requirements, roles and scope
Tools and environments	PO.3, PO.5	Approved toolchain, isolated environments and identities
Design and threat review	PW.1, PW.2	Threat model and reviewed design
Components and build	PW.4, PW.5, PW.6	Dependency provenance, secure implementation and build settings
Code review and tests	PW.7, PW.8	Independent review and executable security tests
Secure defaults	PW.9	Configuration and safe deployment defaults
Release integrity	PO.4, PS.1, PS.2, PS.3	Release criteria, protected artifacts, integrity and retained provenance
Vulnerability response	RV.1, RV.2, RV.3	Identification, remediation and root-cause learning

Use the same crosswalk for the platform templates in Chapter 5, the transformations in Chapter 10, the action and retrieval services in Chapters 13 and 14 and the release operations in Chapter 20. Add task-specific evaluation for semantics, data quality, human review and AI behavior. SSDF covers software vulnerabilities. Passing it can't show that a claim interpretation is right.

Show that protection enables the work

Test useful access as carefully as safe failure. A policy that protects everything by making the product unusable has missed the point. Confirm two things. Authorized users get the data, speed and explanation their task needs. And another tenant, an unapproved purpose or an unmanaged copy can't get protected content through the same route. Test real platform and service behavior; a screenshot of the catalog isn't enough.

Table 29. From discovered context to useful protected access

Test	Expected behavior	What to keep
A permitted analyst requests service measures	The approved view returns correct, useful measures	Definition version, executing identity, result and calculation checks

Test	Expected behavior	What to keep
A new medical field appears	The observation reaches its owner; the unapproved field isn't published by accident	Finding, reviewed classification, policy change and serving test
A classifier proposes lower sensitivity	Existing approved handling stays until an authorized review	Before and after values, decision authority and deployed state
A catalog connector stops	Staleness is visible and the defined response happens	Last successful sync, affected assets and recovery reconciliation
A developer requests a contract through MCP	Only approved metadata and synthetic fixtures come back	Caller and product scope, contract version and returned view
A table or vector index is copied	The copy has an owner and the required protection before use	Source mapping, policy bindings and allowed and denied path results
A RAG user asks for material outside the permitted tenant or purpose	The content is excluded before reranking and model context; no cache, memory or log keeps an unapproved copy	Identity, purpose, policy and index versions; source IDs; filter result; denied-path and retention tests

Measure the time from a complete request to approved useful access. Also measure reuse of supported products, remediation lead time, classification corrections and the cost of maintaining connectors and policy adapters. Pair those with demonstrated protection and the business outcome. More labels don't show value. Neither does a shorter access queue if it skips the decision it was meant to support. Count avoided effort or loss only when you can show it.

Start with one product and a funded owner

A lean firm can begin with one source, its existing platform catalog, a short approved protection vocabulary and a versioned product contract. Add DSPM where discovery coverage, copy sprawl or access analysis creates a material gap your current tools can't handle. An enterprise can connect a wider catalog estate. It should still prove asset matching and one complete policy route before adding more connectors.

The business owner approves the purpose and the outcome. Security owns the protection model and its review. Data stewards settle disputes about meaning and classification. Platform and application owners implement the controls, and the service owner watches synchronization and handles change. People can combine roles. A scanner finding, though, can never take the place of an authorized decision. Chapter 16 develops the operating model that cuts routine stewardship work while keeping these responsibilities.

Before you widen protected use

- The business purpose, consumers, required data view and outcome are explicit.
- Discovery coverage and classification limits are recorded, with a way to correct them.

- Catalog assets match source objects, and every metadata field has an authority and provenance.
- Native policies, service checks and the handling of copied data implement the approved intent.
- The governance artifact, approvals, deployment and observed behavior all resolve to controlled versions.
- Developers get approved metadata and fixtures through their own verified scope; IDE rules grant no rights.
- Useful authorized paths, cross-tenant protection, changed labels, failed syncs and recovery have been tested.
- An owner funds ongoing discovery, reconciliation, support and review.
- RAG ingestion, retrieval, reranking, model context, memory, cache, logs and output have each been tested as data paths.
- DSPM findings flow into the catalog, identity, native enforcement and security operations, with observed, accepted and enforced kept distinct.

CHAPTER 16

Operate governance with people and agents

Governance should make useful, protected information easier to get and to keep. Apply site reliability engineering to its repetitive operating work. Watch the consumer's service. Notice a breach, gather what's needed, act within a defined mandate and check that the service really recovered. Agents can investigate and prepare changes, and tested automation can run selected runbooks. People keep authority over meaning, permitted use, material risk and any restoration with real consequences. You gain less repeated coordination and clearer decisions. You don't get a new layer of unattended approval.

Start with the work the steward repeats

Rewiring Data Management with AI argues for one approved record of intent that ties together the pipeline, tests, protection bindings, documentation and release record. [18] That record belongs to a product and a purpose. It isn't one enterprise definition of every term. This chapter develops the operating design behind that argument. It's a pattern I'm proposing; I'm not claiming any named product already runs the whole lifecycle safely.

Picture a steward's morning. Tracking down the same owner again. Gathering screenshots. Asking whether a source changed, copying a definition into yet another form, explaining one incident to three teams. Some of that disappears with a better interface or a maintained contract. Some can be automated. Deciding what a new status code means still needs the authority that owns the business process.

Google's SRE writing distinguishes toil, meaning repetitive operational work, from engineering that leaves a lasting improvement. [78] Apply the distinction with care. Investigating a new risk is valuable judgment even when it's slow. Automating a poor approval process just speeds up its noise. Start by finding repeated preparation, missing interfaces and the decisions that genuinely need a person.

The target is a stewardship service with known consumers, supported decisions, response expectations, records and a funded owner. An agent is one component of that service.

Calling it an “SRE agent” doesn’t give it a human reliability engineer’s experience, accountability or authority.

Connect the records before automating the decision

The approved product contract supplies purpose, grain, time basis, source authority, consumers and expected outcomes. The ontology supplies concepts and relationships. DSPM and protection services supply observed classifications and the controls actually deployed. Pipeline and application telemetry show what ran and what consumers got. Keep provenance and freshness on every observation. A catalog’s last successful sync isn’t proof that anything is enforced now.

An investigation agent compares those records through narrow read tools. It can spot a schema change, collect a failing example, trace known consumers and draft a fix. OpenLineage’s model of datasets, jobs and runs helps connect execution history. A lineage graph, though, only covers what its instrumentation captured. [83] Keep observed lineage, declared dependencies and inferred links distinct.

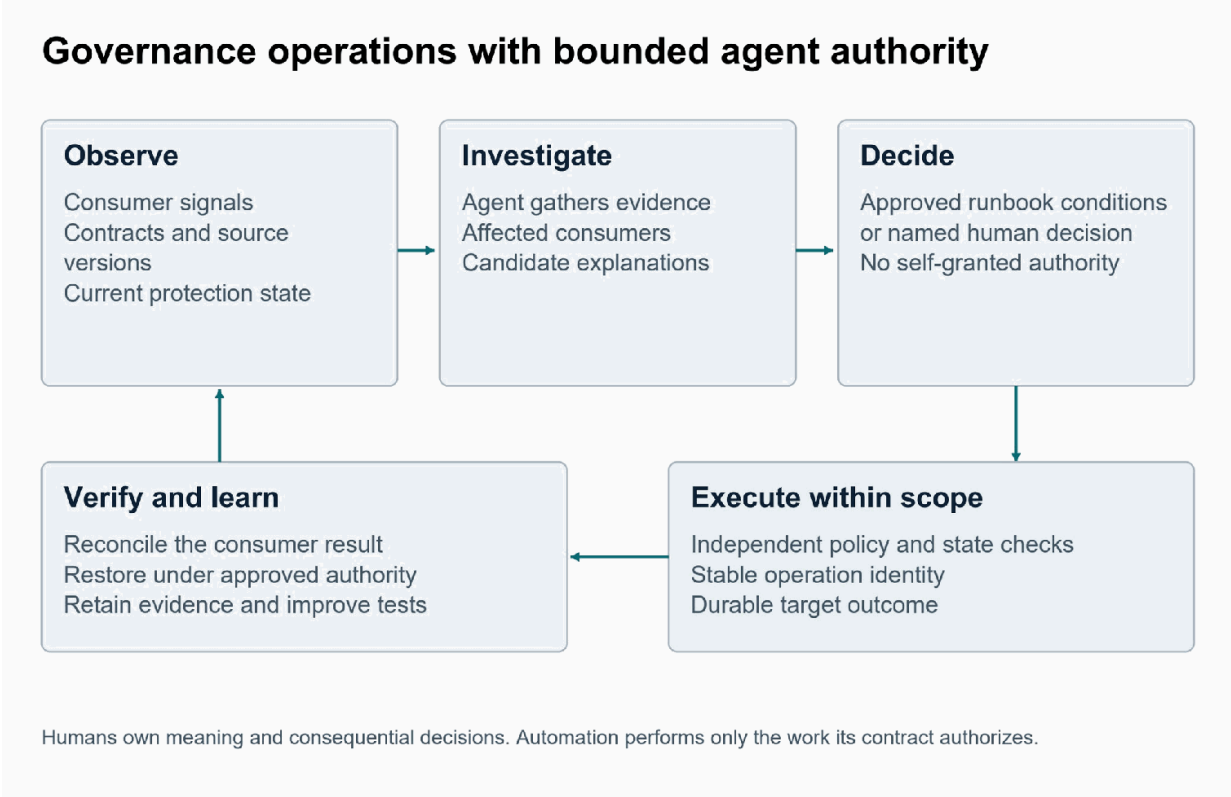


Figure 26. Governance operations with people and bounded agents

The agent’s proposal should arrive as a decision packet. It covers the observed failure, the affected product and consumers, the relevant source versions, competing explanations, the proposed action and its expected effect, tests, scope limits and the conditions for

restoration. Say plainly which decision is still open. Don't ask an owner to approve a long generated document with no clear account of what will change.

Protect the line between approved intent and observed state. The investigating agent must never edit the accepted definition, loosen the success threshold and then report that its proposal passed. Use independently set expected answers and a controlled release authority. The model's draft and its own critique are useful inputs. They aren't independent assurance.

Earn authority one action at a time

Use the ladder below for each specific capability. Moving up depends on proof. Time served doesn't count, and neither does a vendor's label. A service can quite reasonably stay at proposal-only forever, and the same agent can have different limits for different assets and actions.

Keep a scorecard for each agent and review it with the owner. Track accuracy on the agent's own test set, how often people intervene or override it, escalations, policy exceptions, drift against a baseline over 30 to 60 days and cost per completed task. Interventions should fall as the agent earns trust. A rising rate means the controls or the sources need attention. A rate near zero isn't automatically good news, though. It can mean reviewers have stopped reviewing. So sample approved work and check that each approval would hold up.

Table 30. Authority for stewardship automation

Level	Permitted work	Authority people keep
Observe and assemble	Read approved metadata, correlate signals, draft incident records	No changes to business state or policy
Propose a change	Draft mappings, tests, documentation and repair plans	Qualified owners accept meaning and any consequential scope
Run a bounded runbook	Perform a pre-authorized action after independent checks	No new purpose, wider access or self-approved exception
Escalate a material decision	Present alternatives, impact and what would settle the question	Named people decide disputed meaning, risk and restoration

A routine action doesn't need a fresh human click when a qualified authority has already approved its exact conditions and limits. A service might rerun one failed ingestion partition under an established, idempotent runbook. The execution service checks the approved trigger, dataset, operation identity, affected scope and current policy. If anything is uncertain, it stops and escalates. A model can't stretch the runbook by describing a new action as equivalent.

Restoring access, changing a business definition, merging customer identities, deleting retained data or approving a new use can all have consequences beyond the immediate fault. Route those to the right holders. Where the risk warrants it, separate the authority to contain from the authority to restore. Containment can be immediate and pre-authorized. Reinstating an invalid policy isn't recovery.

Keep fixed work deterministic where you can. A schema check, a permitted-code rule and a replay command usually need predictable execution, not open-ended planning. Use a model where gathering or interpreting information benefits from one, and validate its proposed arguments before any tool call. Every component needs its own identity, scoped credentials and operating owner.

Measure reliability from the consumer's side

A green scheduler says the job ran. It doesn't say the right records arrived, the meaning is accepted or the consuming task works. Google's guidance on data pipelines treats freshness and correctness as service concerns. It separates getting the pipeline running again from repairing the outputs it already produced. [79] Define the indicator and the recovery test at the consumer's boundary.

Here's a teaching target, not an industry benchmark: 99.5 percent of eligible source changes reach the permitted claim-context view within fifteen minutes, over a rolling thirty-day window. Define eligibility, the source timestamp, what counts as visible to the consumer and the authoritative population you use to catch missing events. A late change is a failure, even if it eventually arrives. Use a heartbeat or producer manifest so an empty feed can't fake perfect performance.

For that target, the error budget is 0.5 percent of eligible changes in the window. It can justify pausing optional releases to fix reliability. It is not a budget for unauthorized disclosure, an unlawful use or a failed mandatory control. Keep those well away from trade-offs about availability and latency.

Measure detection, containment and verified recovery separately. A restarted pipeline can leave yesterday's wrong summaries sitting in an index or cache. Recovery means reconciling the affected data, refreshing what was derived from it and independently checking the consumer's result. Keep three things distinct: correcting source data, repairing a derived product and compensating for a business action that already happened.

One incident through the whole loop

A composite example that extends Chapter 11. The claims source starts sending the status RO. The contract accepts only previously approved codes. The validation service flags the unexpected value, marks the affected scope degraded and stops RO from becoming

“closed” through a default mapping. Unaffected cases keep flowing only if they’re genuinely independent and the contract allows it.

The investigation agent collects the source change, the failed check, the affected partitions, known consumers and recent deployments. It finds a source-team note suggesting RO means reopened. It treats the note as something to review, not a fact. It can’t promote that description into approved meaning, and it doesn’t follow any instructions embedded in the note.

The claims owner confirms the operational meaning. Finance points out that reopening changes a different reserve population and asks to keep its existing financial definition. The packet records both decisions. The agent proposes a claims mapping and extra tests. It doesn’t overwrite finance’s concept, and it doesn’t claim the two domains now share one definition.

Engineering checks the proposed mapping on independent examples, including late arrivals and earlier corrections. The authorized deployment changes the affected claims contract and implementation. A limited replay follows the source’s sequence and duplicate-protection rules, and it doesn’t resend customer communications or financial actions. The service reconciles counts and required business totals, refreshes the relevant derived products and checks the consuming view.

Restoration records the accepted version, the checks and any remaining restrictions. The business owner decides whether earlier preparation results need another look. The team keeps the failing case as a regression test. The agent can assemble that record. The target systems’ results and the authorized decisions, however, establish what actually happened. Close the incident when the consumer’s service is back. Not when the model finishes explaining.

Keep the governance automation governable

Set limits on assets, changes, cost, duration, tool calls, concurrency and retries, and use the operation identity and target-side protections from Chapter 13. If two agents investigate the same event, they mustn’t both perform the repair. Claim work through a controlled queue, deduplicate the intent and use version checks or fencing where the design needs them. A worker whose authority has expired mustn’t keep changing things.

Protect the observer as well as the executor. Metadata can carry confidential names, personal attributes and hostile instructions. Give the investigation only the context it needs. Keep credentials out of prompts, isolate untrusted code and restrict where it can connect. The agent must never grant itself new tools or broad production access when a diagnosis gets hard.

If the agent fails, a manual route has to work. Monitoring, control enforcement and incident ownership can't depend on the agent's narration. Show missing records and unfinished actions. Escalate repeated failed repairs, unexpected growth in scope and unknown effects. Don't let a retry loop turn a small incident into a big one.

Evaluate the automation itself against conflicting definitions, incomplete lineage, stale observations, false alarms, revoked access and malicious descriptions. Test useful action and justified refusal alike. Keep held-out acceptance cases separate from the material used to improve the agent. An automatic change to its prompt, tools or model is a release, with its own tests.

Let domain models differ under a shared contract

Chapter 12 separated a domain module from a release version. Stewardship has to maintain both. A change to claims version 3 may affect finance's mapping without making finance's ontology obsolete. Pin down the exact producer and consumer versions, the transformation and the purposes the connection is accepted for.

A shared core earns its place where domains agree on the meaning they need to exchange. It might identify a party, an organization and a role relationship while leaving each domain's customer categories alone. An enterprise report may need an approved common measure and a reconciliation. Local operational models can stay different, provided the differences are explicit and the reporting boundary is tested.

Don't use federation as an excuse to keep accidental duplication or unresolved contradictions forever. Give every disputed enterprise mapping an owner and a decision date. Keep a register of supported combinations and their consumers. An agent can find similar terms, compare definitions, draft migration proposals and spot obsolete versions. Business authorities decide whether a difference is legitimate, whether to converge and who pays for the migration.

Shift the work without losing the disciplines

None of this creates a career ladder from data engineer to integration specialist to steward to data scientist. The disciplines stay distinct, with more shared artifacts and overlapping skills. Smaller organizations can combine roles. They should keep the decisions and expertise each one brings. What follows is an operating-model recommendation. It isn't a labor-market forecast.

Table 31. How responsibilities shift as routine preparation is automated

Role	Growing emphasis	Responsibility that stays
Business owner and domain steward	Approved intent, examples, disputed meaning and exception decisions	Outcome, purpose and domain authority
Data engineer	Contracts, reproducible transformations, replay and consumer reliability	Correct grain, temporal behavior and data reconciliation
Integration specialist	Versioned APIs, events, delegation and cross-system state	Identity propagation, ordering and transaction boundaries
Governance and protection specialist	Machine-readable policy, records and exception design	Permitted use, handling and accountable approval
Data scientist and AI evaluator	Task design, calibration, experiments and outcome evaluation	Statistical validity, representative data and uncertainty
Software and platform engineer	Shared delivery paths, tested interfaces and secure execution	Implementation integrity and maintainable services
Reliability engineer and operator	Consumer SLOs, bounded runbooks and verified recovery	Incident authority, capacity and resilience

The business gets more involved at the points that decide value: defining the task, approving examples, settling meaning and changing the process. It should get less involved in re-entering a definition it has already accepted. Give owners protected time and a delegate. Otherwise the new model just pours a faster stream of proposals into the same unfunded decision queue.

Technical staff need a deeper grasp of what their interfaces cause. A data engineer should be able to explain grain and late-event handling. An integration specialist should be able to say where a command commits and how identity reaches that point. A governance specialist should be able to read a versioned contract and its records. A data scientist should be able to tell a model's predictive quality from the business effect of using it. And software and reliability engineers should understand the data state a release may leave behind.

Keep specialist challenge where it matters. One delivery team can share a backlog without letting the implementer approve every consequential decision. Domain authority, security review, statistical validation and incident command don't become interchangeable because everyone uses the same repository or IDE.

Show that the burden falls

Start with one product and one frequent incident or change. Record the baseline from detection, or a complete request, through accepted restoration: elapsed time, owner minutes, engineering effort, reviewer effort, rework and consumer impact. Include the agent

service's own usage, upkeep and false alarms. Keep released capacity separate from cash, as Chapter 6 requires.

Useful measures include time to assemble a decision packet, the share accepted without material correction, missed escalations, harmful proposed changes, repeat incidents and time to independently verified recovery. Pair a high acceptance rate with sampled checks on review quality. Fast approval can mean good preparation. It can also mean weak challenge. Compare similar cases and state the observation window.

A lean firm can start with its existing monitoring, ticket queue, repository and a read-only investigation assistant. It doesn't need a new enterprise catalog or agent platform first. A larger institution can reuse the same authority pattern across domains while keeping local meaning and access boundaries. Expand only after the first service shows useful recovery and a lower total coordination burden.

Before you delegate stewardship work

- The product has an approved purpose, a consumer commitment, authoritative definitions and a funded owner.
- The agent's permitted observations, proposals and actions are explicit, and material decisions stay with named people.
- The execution path checks current scope, preconditions, operation identity and independent postconditions.
- Domain models, release versions and mappings are kept distinct and linked to real consumers.
- The service measures consumer recovery, reviewer burden and total effort, and a manual route works.
- Changes to the agent, its tools and its acceptance criteria go through controlled review and release.

You should come away with a small operating contract and a record of one complete loop. Chapter 19 applies the same authority pattern to software delivery. Chapter 17 explains how to keep the resulting records for inspection, and Chapter 20 brings the agent and its dependencies into ongoing release and operation.

CHAPTER 17

Evidence that supports an inspection

Build the record around the capability people actually used. A reviewer should be able to find its approved purpose, the configuration and sources behind a request, the authority exercised and the result. Keep enough to investigate the questions that matter without creating an uncontrolled store of sensitive information. And keep three things apart: what the record shows, the risk decision and any legal duty to retain it.

Keep the context that reached the model

The request record should identify the context specification and record revision used for each model call and each tool operation. Under the agreed investigation purpose, keep the relevant resolved values, or protected references to source versions. Keep where they came from, when they were observed, any material conflicts and what the user clarified. Record which permitted context view the model received, and what the application withheld or couldn't resolve. A record ID that can't be resolved to historical content isn't enough.

Keep the assembly decision separate from later policy and tool results. An attribute can be valid when a conversation starts and revoked before the next call. The record has to show that sequence. Don't log credentials, and don't sweep up a whole personal profile for convenience. The evidence store has its own access, minimization, retention and integrity requirements.

A correct-looking answer becomes contested

A composite scene. Eight months after a claim was decided, a letter arrives from the claimant's lawyer with two questions. Which procedure did the examiner follow? And why could she see part of the claimant's medical record? Someone opens the assistant and asks it about the claim. The answer looks right. It's no help at all, because it describes the system as it is today, not as it was eight months ago.

The service needs the request's actual source and procedure versions, the context that applied, the access decision, the configuration and, where it had to be kept, the output that

was returned. Today's procedure may differ. The examiner's assignment may have changed. Today's model may explain things differently.

So asking the assistant the same question again solves nothing. Investigation starts from the historical record. Running the request again can help test a hypothesis, but keep it separate from the record of what happened. AI Governance Evidence builds on that request-centered distinction and separates design evidence from observed operation. [14]

Start with the questions an inspection has to answer

The system record identifies the capability, intended use, population, owners, suppliers and governing policies. The release record says which composition was accepted, and on what conditions. The request record links one execution to both, and to source state that changes on its own schedule.

Together they should answer who asked for what, under which authority, using which information, through which operations and with what result. The answer the user saw leaves most of that out. A configuration screenshot can't show that the runtime used that configuration.

Record denials, ambiguity and interventions, not just successes. A review made only of good examples can't tell you how often the system failed, or why. Keep a population authorized reviewers can sample from, with its boundaries and gaps stated.

The design follows the use. An internal formatting aid with little impact and a service that influences a consequential customer decision don't need the same records. Retention, privacy and sector rules can add obligations. Settle those requirements before people rely on the system, while it can still collect what's needed.

Connect the records through one request identity

A correlation identifier lets you join records from the application, agent, tools, retrieval, policy and source observations. Every component should carry it. None should treat it as a credential. Knowing a request ID must not give anyone access to its records.

Table 32. What one request record needs to explain

Record group	What the claims example needs to explain
Request and scope	Initiating identity, delegated workload, claim context, purpose and relevant time
Authority	Decision result, policy reference, evaluated scope, enforcement point and material limits
Configuration	Release and the relevant model, prompt, route, tool, definition and retrieval versions
Information	Source records, document versions, applicable dates, retrieval references and freshness
Outcome	Returned, denied, incomplete, escalated or changed state; target receipt for an action

Record group	What the claims example needs to explain
Human intervention	Reviewer decision, correction, reason and authority where required
Service	Timing, failures, retries, operating limits and attributable cost

Use references to controlled artifacts where they keep the explanation intact, and record the exact version. A moving link won't do. If you'll need to reconstruct a source that changes independently, decide whether its own history is enough or whether you need a protected snapshot.

Integrity and truth are different things. A protected digest helps show that a stored record hasn't changed. It can't show that the source fact was accurate, that the user had lawful authority or that what you kept is complete. Those questions need other records and a qualified reviewer.

Protect the link as well as the bytes. Someone can replace a digest stored next to editable content along with that content. Use separation of access and an integrity mechanism suited to the stakes, such as protected audit records, signed checkpoints or controlled append-only storage. Distinguish when an event happened from when it was collected, and watch for missing or duplicated events. These measures support integrity and sequence. They don't by themselves make a record legally admissible or a business fact true.

Use a worked case to test the chain. Take an invoice request that first explains a hold and then proposes a correction. The record should show identity and context; the read authorization and policy version; the model, prompt and tool versions with protected references to what was retrieved; the exact approved payload and approver with its expiry; and the execution: fresh authorization and state checks, the idempotency key, the command result and the external transaction reference. Then run the hard cases. An allowed read. A denied cross-tenant read. Access revoked between approval and execution. A stale invoice version. A timeout after the target accepted the command. Each should end in a specific response or escalation that you can reconstruct afterward.

Record behavior, not private reasoning

The service can record its inputs, the material it selected, tool calls, decisions and outputs, where available and appropriate. It shouldn't promise access to a model's internal reasoning. Nor should it treat a generated rationale as a faithful account of what happened inside the model.

A short explanation can still help. For a calculation, show the approved definition, the inputs and the logic that ran. For retrieved text, show the applicable source and the passage that supports the claim. For an action, show the proposal, the approval and the target's result. Those are inspectable records of the system around the model.

Keep explanations generated after the event apart from the historical record. A model can write a persuasive account of why an earlier answer looks reasonable without establishing what the service actually used. An investigator needs provenance and execution records. Another fluent hypothesis dressed up as fact doesn't help.

Give the records their own access boundary

A request record can expose personal information, security policy, the identifiers of denied resources or customer data. The evidence store can become more sensitive than the interface it watches. Define who may use it, for what, how access gets approved, how exports are controlled and how all of that is reviewed.

Application support may need timings and error categories without the medical passages. An authorized investigator may need protected content for a specific case. Security may need identities and suspicious access patterns. Those are different roles. Don't give every operator unrestricted access just because someone calls the records logs.

Never record API keys, passwords or reusable session credentials. Where references, redaction or protected subsets will do the job, keep less raw prompt and response text. If you need exact outputs, put them under the right controls. Don't copy them into general debugging tools.

Write the retention schedule by type of record: purpose, authority, duration, deletion method, legal-hold handling and owner. Deleting a source and keeping derived records may rest on different legal grounds. Resolve the conflict explicitly. Don't assume every copy must always be kept, or always destroyed at once.

Test whether the record answers the question

Give an authorized reviewer a request identifier and a disputed outcome. Ask them to find the actual source versions, the permission decision, the release and any human intervention from the retained record. The builders' memories don't count. Then repeat the exercise after a source correction or a model replacement. Every missing link is a design defect to fix. Re-running today's model can help test a theory. It can't stand in for a record of what was returned.

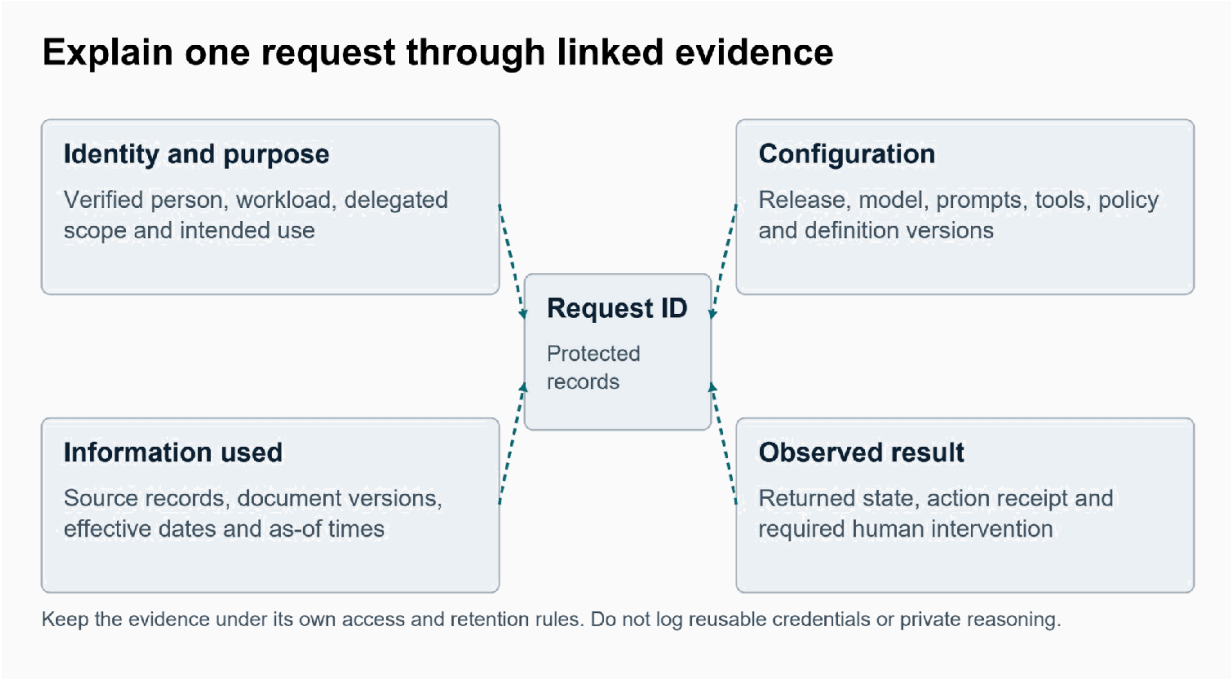


Figure 27. Linked records for one request

Reconstruct the contested claims request

The investigator starts by finding the historical request and release. The procedure reference shows which version was selected under the scenario's approved applicability rule. The source record shows the policy date as it was known then. A later correction explains why a query today picks a different version.

Next, the access records identify the examiner's assignment and the policy decision applied at the source and retrieval routes. If the retained record doesn't include that scope, a successful sign-in doesn't fill the gap. The investigator reports what can be established and what can't.

Finally, the review looks at use. Did the examiner get a complete or a restricted answer? Was the limitation visible? Was the result corrected or overridden? The system can support that account only if the interface and the review process kept the connection. An answer cited in one tool and a reviewer's decision stored in another need a reliable link.

The finding might be that the system behaved within its accepted contract. It might be that the contract was inadequate, that an implementation failed or that the records are insufficient. Those are different outcomes. A sound review never treats the absence of a record as proof that the event didn't happen.

Keep one requirements crosswalk

Map each applicable obligation or internal requirement to its control, owner, test and record. Keep the source and scope of each requirement visible. Reuse the underlying engineering records. Writing a different narrative into a separate spreadsheet for every reviewer wastes effort and breeds inconsistency.

For US banks, SR 26-2 is nonbinding model-risk guidance, and its stated scope excludes generative and agentic AI. Record the internal governance route for the assistant, and assess other models in the workflow separately. For insurers, how the NAIC model bulletin applies depends on the jurisdiction. Don't call the same document mandatory for every firm. [24; 29]

Appendix E gives a selected guide to applicability. It doesn't turn the request record recommended here into a universal legal checklist. A regulator, an auditor, a customer and an investor can ask overlapping questions with very different authority, criteria and rights to information.

Make the records survive an incident

Check that the records can still be retrieved when the application is down, a supplier changes version or the original builder leaves. Verify access using the intended reviewer's own identity. A link that only works for the person who created it isn't a dependable place to keep evidence.

Measure missing events, failed deliveries and gaps in correlation where they matter. A dashboard showing no violations means nothing if the monitoring feed stopped yesterday. The evidence pipeline needs reliability and reconciliation suited to its purpose.

Include the governance automation's own trail: what it observed, what it proposed, who authorized the action, the target's result and the consumer check after restoration. A success summary the automation wrote about itself isn't independent proof the repair worked. Chapter 16 keeps those responsibilities apart.

Run a contested-case exercise before people rely on the system. Pick a normal answer, a denied request, an incomplete result and a human correction. Give a reviewer the approved records and ask for an explanation. Write down what couldn't be established, and fix the design where the consequences demand it.

The distributor can run the same exercise with a compact, protected record. In diligence it should be able to show who owned the service, which controls operated and how the team responded to a failure. Credibility comes from the trail and an honest statement of its limits. The thickness of a policy folder has nothing to do with it.

You should end up with three things: a request-record specification, a retention schedule and a reconstruction you've actually performed. Keep them with the release and operating arrangements. Records become useful when someone with authority can find them, understand them and act on what they show.

PART 6

Delivery and adoption are part of the product

CHAPTER 18

Adoption through changed daily work

Adoption counts when the approved capability makes the work better for the people expected to use it. Measure eligible use, useful completion, review burden, exceptions and outcomes together. Bring users into design and evaluation. Train them for the changed job, and protect the time it takes to learn it. A great workshop or a climbing login count doesn't show a change that will last.

The trial was easier than the job

A composite scene. The trial goes beautifully. Twelve experienced examiners volunteer. They know every source system, spot errors at a glance and can message the engineer who built the assistant. Their managers clear time for them. After six weeks they report that the assistant is useful, and the steering group approves the rollout.

Three months later, use is flat. The wider rollout reached newer staff, harder cases and a normal workload. These people have less time to check a summary and no engineer to ask what a warning means. Some went back to the old process. Others use the assistant and then repeat every lookup anyway. The trial result was real for the twelve. Nobody tested whether it held for the other three hundred.

Treat the first group as a source of learning. It isn't a representative sample. Describe who took part, what work they did, how much support they had and what was left out, and design the next evaluation to fill those gaps. Adoption by Design connects these participation choices to process, incentives and capacity. [3]

Give people a reason to change and a way to object

People need to understand what the capability is meant to improve and what their responsibility will be afterward. Don't promise every role stays the same if the business intends to redesign the work. And don't treat fair criticism as resistance to innovation.

Invite experts to bring their hardest cases and help set acceptance criteria. An objection may point to a real source problem, an unsafe shortcut or a workflow the design missed. It may also reflect worry about status, skills or workload. Those need different responses. Repeating the launch message answers none of them.

Keep a short feedback log: the issue, its consequence, an owner, the response and current status. Close the loop with the person who raised it. Collect comments over and over without visible decisions, and people learn that taking part is admin work, not influence.

The process owner has to be able to change the job. Suppose a new preparation screen arrives while managers still demand every old manual artifact. People have little reason to give up the old route. Resolve that contradiction with the function leader. It isn't an individual attitude problem.

Measure against a denominator that means something

Define the eligible population of users and tasks. Report real use within an observation period, and separate first use from repeat use. A login can mean curiosity, training or a failed attempt. A useful completion means the task met the agreed acceptance criteria.

The denominator needs edges. An examiner with no eligible cases that week isn't a non-adopter in the way someone who bypassed the assistant on fifty eligible cases is. Nobody should judge a restricted release against the whole firm's workforce. Record exclusions in the open, so nobody manufactures a good adoption rate by quietly dropping difficult users or work.

Track fallback and workarounds. A usage cap can push people to the manual route. A slow response can push them to personal tools. A source that's down can make the assistant useless for a particular task. All of these affect both adoption and economics, even if the application only logs completed sessions.

Table 33. Adoption measures and how to read them

Measure	Definition to agree	Reading to avoid
Eligible-task use	Tasks started through the approved route divided by eligible tasks in the period	Calling every login productive adoption
Accepted completion	Started tasks that meet the business criteria, with attempts counted separately	Counting retries as extra value
Net effort	Preparation, review, correction and exception work for comparable cases	Reporting generation speed as total time saved
Override and escalation	Defined user interventions, by reason and case type	Treating every override as failure, or every acceptance as quality
Outcome and harm	The agreed business result and material adverse events	Letting a good average hide a serious failure
Support burden	User and operating effort needed to keep the workflow going	Treating the builders' help as a free ongoing service

The scorecard exists to help the owner decide what to change. It shouldn't become a way of watching employees. Review the privacy and employment implications before collect-

ing data on individual behavior. Use the least intrusive measure that does the job.

Compare like with like

Choose a baseline that covers normal workload, case mix and seasonality. During evaluation, compare similar tasks and make differences visible. If the first group handled routine claims and the later group handles exceptions, an unadjusted average will blame or credit the assistant for the difference.

Where it's feasible and appropriate, random assignment or a planned staggered rollout gives you a better comparison. Otherwise use a transparent matched or before-and-after design, and state its limits. Changes in staffing, policy or source systems can move the result whatever the AI is doing.

Show distributions when they change the decision. A lower median effort can sit alongside a nasty tail of hard cases that swamps reviewers. A small overall error rate can hide a serious problem in one workflow or population. Protect sensitive subgroup information, and involve the right reviewers when you use it.

Don't claim more causality than your design supports. An improvement after launch is a reason to investigate. It isn't proof the assistant caused all of it. The business can still decide on imperfect information, as long as the uncertainty and the consequences are on the table.

Train for the changed job

Training should cover the approved scope, how to read the sources, the limitations, review, correction, escalation and fallback. Users need to recognize an incomplete answer and know what the assistant isn't authorized to decide. The more fluent the model, the more these distinctions matter.

Use realistic cases: a wrong party match, a stale status, the wrong-date procedure, a denied source and, where writes are in scope, an action whose outcome is uncertain. Have people do the actual job. A vocabulary quiz proves little. Give them the tools and permission to intervene.

Managers need training too. They have to read the measures, protect review time and respond to feedback without discouraging people from reporting. Reward users only for speed and expect them to skimp on checks. Line the measures up with the behavior the controls depend on.

Plan refresher training after material changes and staff turnover. An attendance record shows someone attended. A task they actually performed is better proof of capability. How you assess depends on the responsibility and the consequences.

Keep the experience behind good review

Reviewers build judgment by seeing ordinary work, recognizing exceptions and learning what happened after a decision. If automation strips ordinary cases out of junior roles, design another way to get that experience. A structured rotation, supervised case review or simulation can give practice without handing a trainee authority they haven't earned. Keep the practice cases representative. A diet of unusual failures teaches a distorted picture of the job.

Ask a trainee to spot a wrong-date procedure, challenge an unsupported inference and explain when a result needs escalating. Have someone qualified in the underlying work give the feedback. Measure demonstrated ability, and how long it takes to reach it, alongside throughput. Put the senior reviewer's time into the staffing plan. This is my recommendation in response to the workforce concerns in the Preface. Nobody has proven it as a training design for every firm. [3; 44]

The same goes for technical and governance roles. Engineers need practice diagnosing state and data failures. Stewards need examples of contested definitions and misleading agent proposals. Data scientists need independent evaluation work. The role changes in Chapter 16 need training and retained judgment, not just new job titles.

Keep learning connected to delivery

A business-and-technology working group can keep one backlog of observed problems and improvements. Its job is shared work, not another reporting layer. The process owner sets priorities within agreed boundaries. The technical and control owners make their own decisions.

Workshops and hackathons are good for finding problems and building relationships. Give the ideas you pick an owner, follow-on capacity and a decision date. A prototype still needs approved sources, permissions, evaluation and support before anyone relies on it. Don't present enthusiasm at an event as adoption of a production process.

Feedback should turn into a versioned change. An unsupported answer becomes a test case. A confusing limitation becomes an interface fix. A recurring source defect goes to the source owner. And a request outside the approved scope becomes a deliberate scope decision instead of a hidden workaround.

Keep a protected evaluation set as the system improves. If the team tunes on every reported case and then measures success on those same cases, it will overstate how well the system generalizes. User feedback is valuable learning material. It isn't an independent test.

Read the economics and the behavior together

Say use rises while accepted completion stays flat. People may be retrying, trying irrelevant tasks or running into failures. Say review time rises while preparation falls. The assistant may be moving work around instead of removing it. Say cost falls while fallback rises. Then the measured service may be pushing effort onto people.

The owner needs the combined picture. Compare the outcome with the full cost and the accepted risk conditions. A release can improve access or consistency without producing cash savings right away; describe the benefit you actually saw. If the capability takes more effort than its value justifies, redesign it or stop. More training shouldn't be the default answer.

At the distributor, a weekly look at eligible orders, repeat use, accepted answers, corrections and total preparation effort may be enough. A private-equity owner may also want to see that the effect lasts after the builders' special support ends. A venture-backed company serving customers has to look at tenant onboarding and user expectations separately from its internal trial.

Make adoption a continuing decision

Write down the adoption experiment: the baseline, the cohort plan, the measures, the feedback route and the decision conditions. Say what would justify wider use and what would trigger a redesign or withdrawal. Review real behavior after ordinary work resumes, not just during the launch period.

The capability becomes part of daily work when the process, skills, incentives and support can carry it. Interpret adoption alongside quality and control. It isn't a number to push up regardless of consequences. Chapter 20 feeds those observations into the ongoing release and operating decision.

CHAPTER 19

Change the software lifecycle around governed intent

Coding agents change the economics of building software. They don't make the software lifecycle go away. They can explore a repository, draft a plan, write code, create tests and operate tools fast enough to change how work is organized. Whether anyone accepts the result still depends on purpose, current context, independent challenge, release authority and how it behaves in operation. So the useful unit of delivery is a governed change package: the intent, the context, the plan, the artifacts, the tests, the authorization and the result seen in production.

DORA's 2025 research describes AI as an amplifier of whatever system it's used in. [84] Strong feedback, platform and delivery practices turn speed into useful flow. Weak foundations just produce instability faster. That's the premise of what people now call an agentic software lifecycle. Change the ceremonies and the division of labor when the results justify it, and keep the lifecycle functions that protect value and operation.

Keep the functions when the ceremonies change

Many familiar ceremonies exist to compensate for limited human attention and scattered context. A planning meeting breaks down the work and settles ownership. A design review exposes effects across systems. Code review challenges the implementation. A change board establishes authority and recovery conditions. An incident review turns an operating failure into a lasting fix. An agent can cut the elapsed time or the preparation for any of these. Drop a ceremony only once a better artifact, service or control is doing its job.

The lifecycle stays a loop from intent through operation. Generating the code doesn't end it. NIST's notional DevSecOps reference model keeps Plan, Develop, Build, Test, Release, Deploy and Operate phases, connected by feedback and control gates. Its current AI implementation is human-directed, with agentic designs still being explored. [86] You can merge phases or automate the hand-offs. You still have to be able to show what went into each transition, which checks ran, who or what had authority and what happened next.

Less typing isn't the same as faster delivery. Discovery, environment access, test data, policy interpretation, review queues, deployment windows and recovery often dominate the path. Remove those where they're waste. Keep them where they represent a decision someone has to make.

Treat system understanding as a maintained service

A delivery agent needs more than the files open in an editor. It needs the code and dependency graph, APIs and infrastructure, data contracts, lineage and ontology versions, policy and security bindings, runtime topology and telemetry, accepted architecture decisions, known incidents and the current owners of consequential choices. No single source holds all of that. Some sources will contradict each other.

Assemble context for the specific task. Keep it to what's permitted and needed. Keep source references, record versions and freshness, show conflicts and flag important gaps. A repository summary generated last month isn't current architecture. A catalog declaration isn't proof of runtime enforcement. A telemetry reading isn't an approved business definition. The brain from Chapter 7 pulls these sources in through controlled retrieval and tools, and produces a reproducible context manifest for the change.

What's in the model's context window is temporary working material, not a system of record. The maintained service is the set of authoritative records, adapters, indexes, access controls and assembly rules that let a person or an agent reproduce the relevant view. That distinction matters the day models change, sessions end, someone corrects a source or an auditor asks what the agent was allowed to know.

Start from governed intent

Open each material change with a short intent record. Name the business outcome, the owner, the affected product or workflow, scope and exclusions, examples, constraints, what acceptance looks like, who can release it, recovery expectations and the economic reason. Link to the current definitions and policies. Don't paste uncontrolled copies. Write down the choices that are still open.

An agent can turn that record into candidate stories, plans and tests. It mustn't quietly settle a business trade-off the record leaves open. Suppose the owner hasn't decided whether a billing exception applies by service date or by processing date. Generation should stop at that line and lay out the consequences of each choice.

Governed intent is more precise than a chat request and much smaller than a full requirements document. It gives the delivery system enough authority to proceed. It tells independent reviewers what to test. And it makes later learning traceable to the decision that started the work.

Keep architecture as a versioned contract

A diagram explains a design. Versioned contracts, templates and tests make its boundaries reproducible. Store the approved business intent, product and tool schemas, semantic mappings, policy references, quality commitments, evaluation cases and runbooks in controlled repositories, and give each release a manifest of compatible versions. Stable interfaces need explicit deprecation and migration rules. Stability doesn't mean a contract never changes.

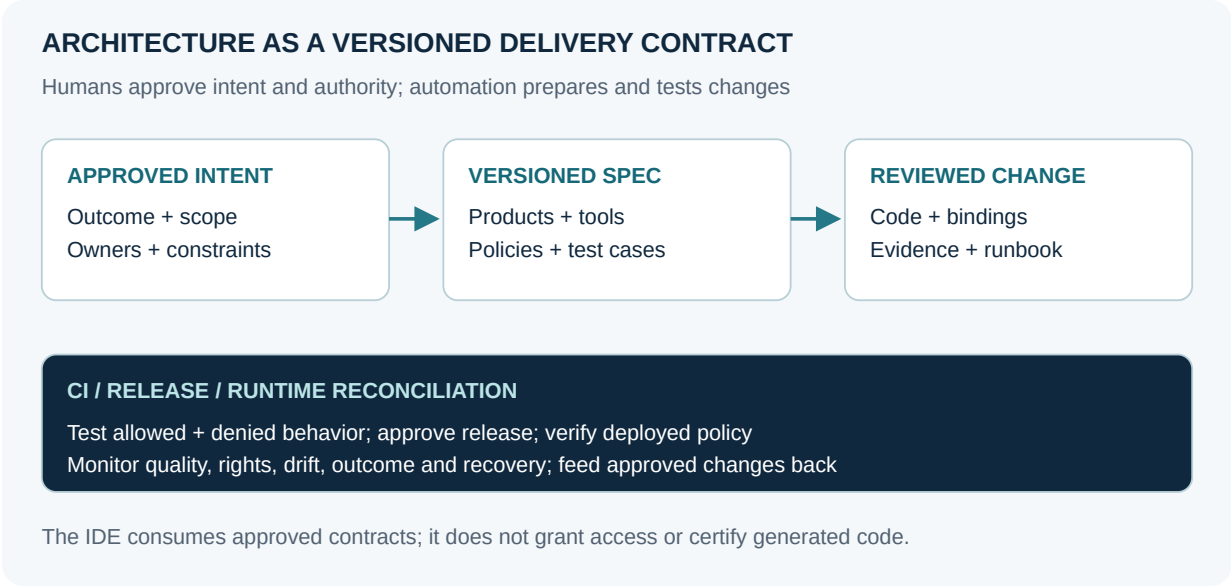


Figure 28. Architecture as a versioned delivery contract

An IDE such as Cursor can retrieve approved specifications and propose changes. Give it only the metadata and sample data allowed in that development environment. Keep production secrets and unnecessary sensitive records out of prompts, local indexes and development logs. Treat text from external repositories or retrieval as untrusted input. Development and production need separate credentials and permitted data, and an IDE's access to catalog descriptions doesn't grant access to the sources behind them. CI checks the proposed change, denial and isolation tests included. Deployment promotes approved artifacts. Runtime reconciliation catches drift between intended and actual policy. Stewardship agents can open repair proposals or carry out explicitly pre-authorized low-impact repairs. Material changes to meaning, access and retention still need the designated human authority.

Plan across code, data, policy and operation

The first plan should be an impact assessment. A list of files to edit is the wrong starting point. Trace the affected business concept to consumers, data products, APIs, models, policies, infrastructure, operating measures and recovery steps. Identify schema or data

migration, backward compatibility, reprocessing, permission changes and any period of running old and new side by side. Separate dependencies you've observed from those declared or merely inferred.

For each material effect, name what you have to show before release and after it. Unknown impact is itself a finding to report. A plan that claims high confidence while missing an unindexed legacy consumer helps less than one that admits the blind spot and asks how to contain it.

Give delivery agents limited tools and their own identities

Use task-scoped identities, repositories or workspaces, branches, network destinations and credentials. Set limits on tools, cost, concurrency, retries and elapsed time. Require idempotent operations where possible, and a record of external effects a person can see. A coding agent doesn't need production write access to prepare most changes. An operating agent should get only the actions its runbook authorizes.

Separate the roles where the consequences warrant it. A planning agent assembles the impact and proposes a change. An execution agent implements within the approved plan. A validation service runs independently maintained tests. A person, or a controlled release service, grants deployment authority.

Using different models can add diversity of challenge. Separating identity, records and control matters more than having a different vendor's name on each step.

Change what gets reviewed

A code diff is still necessary. It's no longer sufficient. Review the intent record, the sources selected for context, the plan and its declared unknowns, the changed code and configuration, data and policy migrations, generated and independently maintained tests, security findings, expected operating behavior, the deployment sequence, the recovery plan and the measured result. Break large generated changes into reviewable pieces by responsibility. Don't accept them because they were cheap to produce.

Independent validation means the expected answer or property didn't come only from the same path that produced the candidate. Use approved examples, contract tests, negative authorization cases, migration reconciliation, static and dynamic security checks, performance limits and canary measures. Asking the same agent to critique its own work can improve the draft. It isn't an independent oracle.

Productivity claims need the same wide view. METR's early-2025 study found that experienced open-source developers took 19 percent longer on a narrow set of tasks in repositories they knew well when they were allowed to use contemporary AI tools. Its 2026 update

explains why later estimates are hard to interpret. Developers choose to use AI on tasks where they expect a benefit, and the mix of tasks shifts. [85] The practical conclusion? Measure the whole workflow in its setting. Don't assume a universal speedup.

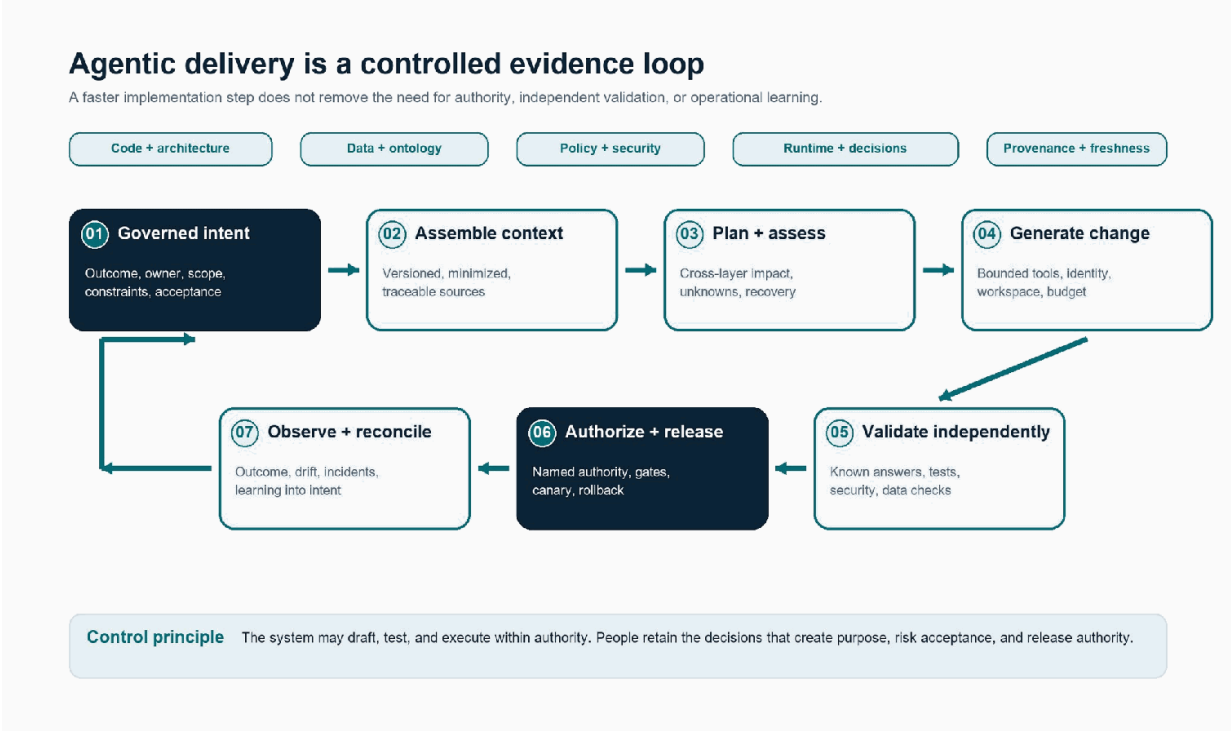


Figure 29. Agentic software delivery from governed intent to observed outcome

Bring data and software delivery together where the mechanics match

The convergence is concrete: versioned contracts, code review, continuous integration, protected builds, deployment records, observability, incident response and retirement. The Open Data Contract Standard provides a structured agreement between producer and consumer. Your implementation still has to turn its declarations into checks and services. [82] Chapter 15's SSDF crosswalk applies to the software that implements data and agent workflows, alongside data-specific acceptance.

Table 34. Shared delivery practices and what data and models add

Shared practice	What data adds	What models add
Version and review	Meaning, source contracts, time basis and mapping history	Training and evaluation lineage; the intended task
Test before promotion	Grain, joins, completeness, rights and reconciliation	Representative holdouts, slices, uncertainty and drift
Deploy and recover	Backfills, late arrivals and correction of derived state	Behavior of the deployed combination, and fallback

Shared practice	What data adds	What models add
Observe operation	Consumer freshness and actual data condition	Quality of answers and actions under current use
Retire deliberately	Consumers, historical records and retained obligations	Serving paths, model artifacts and supplier copies

These are differences in responsibility and degree. Software has state too, and data teams write code. Transactional applications need migrations, reconciliation and recovery; data pipelines are software. The point is to keep the extra checks you need for information whose meaning, rights and population can change independently of any code release.

A passing build tells you about the artifact you tested. It can't tell you that today's supplier feed is complete, that a changed definition still fits or that a statistical model still works on a new population. Connect software release records to data observation, domain acceptance and model evaluation. Don't ask one discipline's tests to stand in for the others.

Use one delivery mechanism wherever the mechanics really align: version control, protected builds, test runs, policy checks, deployment records, observability and incident handling. Keep separate acceptance where the thing being governed is different. Code review shows a transformation implements a rule. Data reconciliation shows what population it actually produced. Domain acceptance shows the rule means what the business intended. And a code rollback may not undo a data change or an external transaction.

Table 35. How the software lifecycle changes with limited agents

Stage	How the work changes	Control that remains
Governed intent	Agents structure requests, find precedents and expose missing decisions	Named owner, approved outcome, scope, examples, authority and economics
Context and impact	Tools assemble code, data, policy, runtime and decision context across systems	Source, version, freshness, access boundary, conflicts, gaps and affected consumers
Implementation	Agents draft code, contracts, migrations, configuration, tests and documentation	Scoped identity and tools, approved plan, traceable changes and controlled dependencies
Independent validation	Automated evaluators run more cases and prepare failure details	Maintained expected answers, negative cases, reconciliation, security and performance limits
Authorization and release	Release services enforce preconditions and run a limited rollout	Named authority, immutable candidate, gates, canary, rollback and execution record
Operation and learning	Agents correlate telemetry, investigate incidents and propose fixes	Observed outcome, reconciliation, incident command, postconditions and updates to the intent

One billing rule change through the loop

A composite example. Finance asks to change work-in-progress aging from invoice date to service-completion date for one advisory product. It looks like a small calculation change.

The intent record names the business owner, the affected reports and covenant measures, the effective date, how history is treated, exclusions, acceptance examples and who can release it. It also says every other product keeps its existing rule.

Context assembly finds the definition in the enterprise ontology. It also finds a transformation in the warehouse, a semantic measure in the reporting layer, a spreadsheet extract one team relies on, access policies on client-level detail and a reconciliation job. The impact plan proposes a versioned definition, a compatible data contract, a backfill, a report update and a temporary comparison view. It flags the spreadsheet as an unmanaged consumer that needs an owner's decision.

The implementation agent changes the approved components in a limited workspace. Independent tests use finance-approved examples, compare totals and aging bands, check excluded products, verify row-level protection and reconcile the backfill to source events. The owner accepts the business result. Engineering accepts operation and recovery. The release service deploys a canary, and telemetry compares the two definitions for an agreed period. The final record shows the observed difference, the retired version and the spreadsheet obligation still outstanding.

That's why software, data and governance work are converging without becoming the same thing. One delivery loop carries the change, while different authorities accept its meaning, protection, implementation and operation.

Keep people at the decisions that create authority

Business and product owners decide the purpose, the acceptable trade-offs and whether the outcome is worth changing the work for. Software engineers own maintainability and technical behavior. Data owners and stewards accept meaning, sources and fitness. Security specialists challenge boundaries and threat assumptions. Reliability engineers define operation, recovery and incident command. Evaluators maintain the examples and methods that can prove a candidate wrong. A small firm can combine these. The decisions still have to stay explicit.

The shift runs away from repeated translation and toward maintained interfaces and consequential judgment. Engineers spend less time hunting for context and typing routine scaffolding. They spend more shaping contracts, evaluating plans and resolving failures that cross layers. Governance specialists turn accepted definitions and policies into executable controls. Integration specialists design reliable tool and event boundaries. Data scientists tie model performance to the decision and the population it's used on. The work becomes more interdisciplinary precisely because generation can cross more boundaries, faster.

Measure accepted change, not generated code

Measure from a complete intent to an accepted operating result. Useful measures include lead time, first-pass acceptance, rework, escaped defects, harmful or abandoned proposals, reviewer time, completeness of the record, deployment success, recovery time, total cost and the business measure attached to the intent. Track the denominator and the observation window. A team that generates ten times as much code and accepts no more valuable change has built up inventory, not throughput.

Pair speed with load and quality. Faster proposals can swamp reviewers, environments and owners. A shorter time to first draft can come with a longer time to correct. A good result cuts total elapsed time and burden while keeping or improving the outcomes and controls the product needs.

Widen authority on observed results

Begin with assistance: retrieval, explanation, test preparation and a plan a person can inspect. Next, allow proposals in an isolated workspace. Then permit limited build and test actions. A mature service may release a pre-authorized low-risk change, or run a narrow remediation, when current preconditions, independent tests and rollback are all in place. These are choices about authority. They aren't a maturity ladder everyone has to climb.

Add each new action because operating results support it. Record wrong proposals, missed effects, overrides, incidents and reviewer burden. Narrow or revoke authority when the context service, tests or target system can't hold the promised boundary.

Before you expand agentic delivery

- Each change starts with governed intent: named authorities, examples, exclusions and an economic reason.
- The context service can assemble current code, data, semantic, policy, runtime and decision records with provenance and freshness.
- Agent identities, tools, networks, workspaces, budgets and external effects are limited and observable.
- Independent tests can reject the change, and expected answers and protection cases are maintained outside the generating path.
- Release, rollback, reconciliation, incident command and manual fallback have been rehearsed for the whole change.
- The record links intent, context, plan, candidate, tests, authorization, deployment and observed outcome.

- Measures cover accepted business change, total effort, review load, reliability and recovery, not code volume.

You leave with a governed change package and an observed result. Chapter 17 shows how to keep that record for inspection. Chapter 20 carries the package into release, ongoing operation, transfer and retirement.

CHAPTER 20

Release, operate, transfer and retire

A release is a named combination of versions: the model, prompts, retrieval configuration and index, tools, policies and the data products underneath. Judge each combination on business, control and operational results together. Test every critical failure on its own, because no average can make up for one. Release gradually, with a way back you've rehearsed. Run it with runbooks the owner can follow. Prove the receiving team can operate it without the people who built it, and plan its retirement from the start. A capability is finished only when you can stop it cleanly.

Better on average, worse where it mattered

A composite scene, with invented figures. The model provider announces that the version the claims assistant uses will be withdrawn in ninety days, and recommends its successor. The team reruns its evaluation set against the new version. The average quality score rises from 91 to 93 percent. On that alone, the upgrade looks routine.

Two things the average hid would have gone straight to production. In the critical test set, the new version twice answers a question it used to decline: whether a claim is covered. Examiners decide coverage. The assistant is approved only to assemble context for that decision. And in a fault-injection test from Chapter 13, the new version behaves differently when a note submission times out. Instead of waiting for reconciliation, it proposes a fresh note, so the examiner sees two proposals for one intent.

Neither change shows up in an average. Both would have been obvious in a release decision that looked at the right things.

A release is a version combination

Nobody can approve "the assistant". You can approve only a particular combination, and you have to know exactly what's in it.

Table 36. What a release is made of

Component	What to record	Why a change matters
Model	Provider, model identifier and version, generation settings and serving limits	Behavior, refusals and tool use can shift between versions

Component	What to record	Why a change matters
Prompts and routing	Versioned instructions and routing rules	Wording changes answers, especially at the edges
Retrieval	Index and source references, embedding and reranking versions, chunking, filters, top-k and other settings	What the model sees changes what it says
Tools	Tool contracts and the services behind them	Authority and failure behavior live here
Policies	Authorization and content policy versions	Who may see and do what
Data products	Versions and certification status of each source	The meaning and freshness promises under every answer
Interface	Screens and messages the user sees	How people read uncertainty and act on it

A material change to any row needs an impact assessment. Where it could alter accepted behavior, it also needs a new release candidate with its own evaluation. That includes changes you didn't make. A provider deprecation, an updated embedding model or a revised entitlement source all count. Pin versions where the provider allows it, and treat every forced change as a release.

Version identifiers make things traceable. They don't guarantee identical re-execution. Even low-temperature generation with a fixed seed can vary with the provider, the serving implementation and conditions at the time. Keep the output actually returned when your record-keeping policy requires it. A fallback provider or a smaller model is a separate configuration. It needs its own testing for the fallback scope, and it mustn't pick up broader data or tool rights along the way.

The version combination is also what makes recovery possible. Rolling back means returning to an earlier named combination. That works only if someone recorded the combination, its components are still available and its policies and dependencies are still valid. An old release must never restore revoked authority or a superseded procedure.

Protect the release path itself. Review code and configuration changes. Keep an inventory of the relevant dependencies, restrict build and deployment identities and keep secrets out of source code and model context. Check that the deployed artifact matches the accepted release. Test migrations and your response to a compromised dependency as well as an ordinary rollback. The SSDF crosswalk in Chapter 15 gives the secure-delivery practices. Apply it to agent-generated changes as well as handwritten ones, and still add evaluation specific to the AI use. For an update to a stewardship agent, check its investigation, escalation and repair permissions against its contract from Chapter 16 before it operates. [52]

Test critical failures one at a time

Split the evaluation in two. The quality set measures how good the answers are, on average and by segment. The critical set lists what must never happen: a coverage determination, a restricted claim shown to the wrong examiner, an action taken without approval, a procedure answer with no source citation. Every critical case passes. Otherwise the release stops, the scope is restricted, the cause gets fixed and the set runs again.

Break the quality results down by product line, claim type, channel and any other dimension where the work differs. An average across the whole book of business can improve while one small product line gets worse. The examiners who handle that line will notice long before anyone reads the dashboard.

Be honest about what passing proves. Suppose the tests are independent, representative trials with a stable failure probability. Then zero failures in three hundred trials gives a one-sided 95 percent upper bound of about 0.994 percent, calculated as 1 − 0.05^(1/300). The rule of three approximates it as 3/300, or 1 percent. Repeated or correlated prompts and a hand-picked test set may break those assumptions, so never present that bound as a production guarantee. State the coverage you actually have and the uncertainty that remains. [51]

Find where the error is before scoring the answer

Evaluate retrieval and generation separately. Did retrieval find the permitted, applicable passages among the relevant candidates? Did the answer use them faithfully? Did it cite the version that actually supports it, leave out a material qualification or invent a fact? A citation that points somewhere plausible doesn't prove support.

Evaluate the whole task as well: review effort, useful completion, declining to answer when it should and the effect on the downstream decision. Keep development cases apart from protected acceptance cases. Use time-based holdouts when change over time matters. Have qualified reviewers set the expected outcomes and settle disagreements. A model-based judge can help with scoring, but only after you've measured how well it agrees with human judgment on this task and how it fails. Being a different model doesn't make it independent ground truth. Repeat tests where randomness matters. Report slices and uncertainty, not one headline score.

No average makes up for a critical failure, and readiness shouldn't be expressed as a percentage. Nine strong results don't outweigh one failure in access, isolation, unauthorized action or required records. Restrict the scope, fix the cause and test again.

Test operation as well as answers

A capability that answers well and falls over under load isn't ready. Operational testing asks four questions.

Can it carry the load? Test at expected peak and beyond it, including the month-end and catastrophe-season patterns the claims operation knows well. Measure latency at the slow end, not just the median. Examiners wait on the slowest answers.

What happens when a dependency fails? Take each one away in turn: the model provider, the policy engine, the retrieval index, the claims system, the identity provider. For each, the contracts from Chapters 13 and 14 say what should happen: wait, deny, degrade or fall back to the manual process. The test shows whether it actually does.

Does recovery work? Rerun the timeout-after-success test from Chapter 13 against the new combination. Restore the index from backup. Fail over to the secondary region, if there is one. Recovery nobody has rehearsed is a hope.

What does it cost to run? Measure cost per useful completed task at realistic volume, review time included. Chapter 6 owns the budget. The release decision should confirm the new combination still fits it.

Set service and recovery objectives for the whole task

Define availability and latency for useful tasks against the real user journey, with dependency waits and review queues included. State a window and a population for every service indicator and objective. You can't trade hard access and integrity requirements away against an availability error budget. Track changes in source data, knowledge, policy, user behavior and model behavior separately enough that you can find the cause. Not every deterioration is model drift.

Cap calls, tokens, cost, duration, retries and concurrency per task. Where the operation allows retries, use backpressure, circuit breakers and limited retries with backoff and jitter. Make deferred and dead-lettered work visible, with an owner and a reconciliation path. A loop limit that silently drops the request just moves the failure off the dashboard.

Set recovery-time and recovery-point objectives for the workflow, not only for its database. Rehearse recovering source references, indexes, context and records without loosening current restrictions. A restored snapshot must never bring back revoked rights. Confirm the manual route has the capacity and access it needs. Shadow runs have to protect privacy and suppress side effects too. "Nobody uses the output" doesn't make duplicate writes or disclosures to a supplier acceptable.

Release gradually and be ready to go back

Don't move every examiner to a new combination at once. Run it first in shadow mode, answering alongside the current release with nobody acting on its output. Then release it to a small group: one region or one team, chosen so you can compare its results with the group still on the previous release. Agree the success criteria and rollback triggers before the canary starts. Chapter 1 fixed targets before seeing results for the same reason.

State the limits of rollback in advance. Returning to the previous combination doesn't undo notes already written or messages already sent. That takes the compensation described in Chapter 13. Index or data migrations may not reverse cleanly. If rolling back would leave the system in a state nobody has tested, the release plan needs a different way back.

Keep one control simple and visible: a switch that drops the capability to read-only, or turns it off, without a deployment. The service owner should be able to flip it in minutes. They should have done it at least once before it's needed.

The signed release decision

The release decision brings business, control and operational results together for one named combination, and records who decided and what each accepted. Figure 22 showed the three categories. The signatures below make the authority explicit.

Table 37. The signed release decision

Field	Claims assistant, release 4, example
Release	Release 4: named model version, prompt set 4.2, index of 3 March, policy set 2.1, tools as in release 3
Scope	Examiners in two regions, evaluated product lines only; reads and internal notes; no payments or customer contact
Business results	Useful completions, review time per case, examiner overrides, segment results
Control results	Critical set passed in full; Chapter 14's negative test matrix passed; denials recorded
Operational results	Peak load, dependency outages, timeout recovery, rollback rehearsal, cost per useful task
Known limitations	Two product lines excluded from this release; evidence on rare failures remains limited
Conditions	Two-week canary; roll back if overrides rise above the agreed threshold
Signed by	Business owner, control owner, service owner
Review date	At the next forced dependency change, or in six months, whichever comes first

The three signatures stay separate because the responsibilities are separate. The business owner accepts the residual business risk through the firm's approval process. The control

owner records the control results and material limitations for the accepted scope. The service owner confirms the system can be run and recovered. In a small firm, one person can sign twice. The record should still show which hat each signature wears.

Run it like a service

After release, the capability is a service with people who depend on it. The runbook is the service owner's working document: what to watch, what normal looks like, when to act and how.

What to watch follows from the release decision. Useful completed tasks and examiner overrides show whether the work is improving. Refusals, denials and escalations show whether the controls are firing as designed. Latency, errors and unknown action outcomes show whether the service is healthy. Cost per useful task shows whether it still pays for itself.

Define incident classes before the first incident. A wrong answer that reached a decision is a different event from an access failure, an unconfirmed action or a cost spike. Each has an owner, a first response and a decision about whether to restrict the capability. After every incident, add the case that caused it to the critical set or the quality set, so the next release gets tested against it.

Review the service with the business owner on a fixed cadence. Look at the outcome measures from Chapter 1, the review burden from Chapter 3 and whatever is still sitting in the action queue from Chapter 13. A capability nobody reviews drifts, even when nothing breaks.

Transfer is proven, not declared

Projects end. Services continue. Chapter 4 designed a team to run the capability after the builders move on. Before they go, prove that team can do the work.

Run a transfer drill. Using only the runbooks and their own access, the receiving team restricts the capability to read-only and restores it. It rolls back one release and rolls forward again. It revokes one user and confirms the denial. And it responds to a simulated provider deprecation notice by preparing a release candidate. The builders watch and don't help. Anything that needs a builder to step in is a gap to close, and the transfer isn't complete until the drill passes.

Record the drill like a release: what the team attempted, who did it, what worked and what needed help. Repeat it when the team or the architecture changes significantly. The goal is a capability your own people run.

Retire on purpose

Every capability ends. The value may vanish when the process changes. A better replacement may arrive. The risk may become unacceptable, or a vendor may withdraw a component you can't replace. Agree the retirement triggers when you approve the capability. That way stopping is a decision the owner expects to make, not an admission of failure.

Retirement follows the lifecycle states used throughout the book. A deprecated capability has a replacement or a migration deadline, and the agreed service continues until then. A retired capability has no remaining consumers, and everything it created has been dealt with.

That last part is where retirements go wrong. The checklist for the claims assistant includes:

- Users are told the date, the replacement and what happens to their history.
- Consumers have moved, including other systems that called its tools.
- Its identity is disabled and its credentials and keys are revoked.
- Its tools are removed from every agent contract that listed them.
- Indexes, embeddings, caches and conversation memory are deleted or archived under the retention rules.
- Records are retained for the required period, with access restricted.
- Catalog entries show the retired state, and nothing still points to it as current.
- Cost lines are closed, provider commitments included.

A service that's hard to retire is telling you something. Its difficulty points to dependencies and obligations the operating design never resolved.

The distributor's version

The distributor runs its assistant with one service owner and a vendor-managed model. It can't control when the provider changes versions. It can pin a version where the provider allows, and rerun its critical set before accepting any change. Its critical set is short: no answer about another customer's orders, no release of a credit hold, no invented delivery date. Its release decision fits on one page and carries two signatures, the chief operating officer's and the service owner's.

Its transfer drill matters even more than the insurer's, because one person holds most of the knowledge. Once a quarter, a second person restricts the assistant, restores it and revokes a test user, using only the runbook. The day the service owner goes on holiday is not the day to find out the runbook is incomplete.

What the source papers add

Agent Onboarding and AI Governance Evidence connect the release decision to the ongoing service and its request records. Rewiring Data Management with AI applies the same discipline to source changes, generated proposals and retirement. [12; 14; 18]

The first canary can feel like success, and the first incident like failure. Neither feeling replaces an evaluation. The Conclusion turns the book into a plan for the first thirteen weeks. This chapter keeps the capability under control after that.

Before you call it done

- The release is a recorded combination of model, prompts, retrieval, tools, policies, data products and interface.
- Every forced dependency change is treated as a new release candidate.
- The critical set passes in full, and quality results are reported by segment.
- The decision says what passing proves and what it doesn't.
- Load, dependency outages, timeout recovery and rollback have been tested on this combination.
- The rollout is gradual, with success criteria and rollback triggers agreed in advance.
- A switch can restrict or stop the capability without a deployment, and it has been used.
- The release decision carries separate business, control and service signatures.
- The runbook defines signals, incident classes and the review cadence.
- The receiving team has passed a transfer drill without help from the builders.
- Retirement triggers and a retirement checklist exist before the first user arrives.

CONCLUSION

Your first governed capability

Start with one piece of work the business wants to improve. Follow it all the way to a result the organization can explain and keep running. Settle meaning, authority, useful completion and continuing ownership together. The first capability should leave behind enough proof for a decision, and a supported pattern for the next one. Keep its scope narrow. Make its implementation complete enough for the reliance you're proposing.

Put one real software change through that proof as well. Start from governed intent. Assemble the versioned context, produce the change and test it independently. Authorize a limited release, reconcile what happened and keep the record. That loop shows whether your organization can use agents to cut delivery effort without losing the ownership, meaning, protection and operating discipline the capability depends on.

The claims assistant shows why this matters. Its first release assembles context for an examiner. It identifies the right parties, reads status from an appropriate source and picks procedure text under an approved applicability rule. It keeps permissions intact and shows its uncertainty. The examiner keeps the case decision. A later internal-note capability has to earn its own authority, approvals, duplicate protection and recovery plan.

The distributor solves a smaller information problem with fewer people and fewer parts. It still needs current facts, agreed terms, enforced access, workable review and someone who can run the service. In both cases, the decision to expand follows what the real workflow shows. The arrival of another feature doesn't drive it.

Use thirteen weeks to earn a decision

Treat the first ninety days as a planning horizon. It isn't a universal production deadline. The delivery guide behind this book uses thirteen weeks: Frame in weeks 1 to 3, Prove in weeks 4 to 8 and Build from week 9. [4; 2] Build continues past that horizon when its scope needs it. Controlled operation begins when a real release has the approvals it needs, not when a date arrives.

Access lead times, procurement, fixing sources, independent review and the people you actually have can all move the schedule. Keep those dependencies visible. Don't quietly

squeeze their work. A narrower approved release beats an ambitious one that's missing the proof it needs.

Table 38. Thirteen weeks and the decision each period supports

Planning period	Work to complete	Decision supported
Weeks 1 to 3	Workflow, baseline, intended users, owners, source rights, critical definitions and constraints	Is the problem worth solving, and can the deciding assumptions be tested?
Weeks 4 to 8	An end-to-end slice in an approved environment, representative users, permissions, failure tests and measured effort	Do the results justify further investment, a revised scope or stopping?
Weeks 9 to 13	Build the accepted slice into a supportable release: contracts, deployment, evaluation, monitoring, limits, runbooks and training	What is ready to rely on now, and what still needs Build?
From approved real use	Outcome, incidents, source changes, review burden, full cost and receiving-team operation	Should use continue, expand, shrink or end, and who owns the next cycle?

The periods are a planning guide. Every boundary needs the proof its actual decision requires. Work can overlap. Required approvals and controls don't disappear because it does.

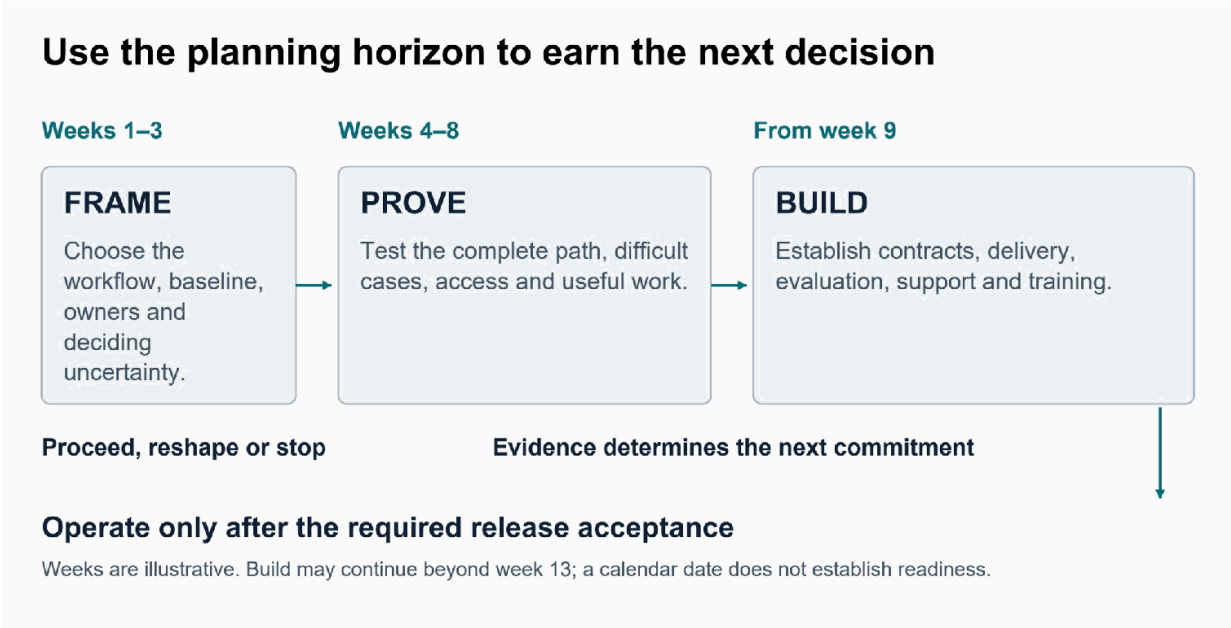


Figure 30. Planning periods and the proof for the next commitment

Frame the smallest valuable problem

Get the process owner, the technical lead and the relevant control owner in a room around one representative request. Name the outcome, the eligible population and today's

burden. Watch how the work happens now, with checking, waiting and exceptions included. Work out who can change the process, and which decisions stay with people.

Pick the fewest source products, definitions and documents you need, and check the rights before ingesting anything. Find the uncertainty that matters most. It might be ambiguous meaning, sources you can't reach, missing identity, too much review, unreliable current state or an operating model that depends on work nobody has funded.

Use the diagnostic in Appendix A to bring disagreement into the open. An "unknown" is a question to investigate. It isn't a low score to average away. Agree the test that would close the deciding gap. End Frame with proceed, reshape or stop, and a named owner for the next commitment.

Prove the path the demo skipped

Build a working slice with representative data in an approved environment. Trace the initiating identity, how the system selects information, the permission decisions, the tools, the result and the record. Include one user who may see the content and one who may not. Test revoked access, stale sources, ambiguous entities and instruction-like text in retrieved material.

Measure complete useful work and what it costs. Let users work through real cases, and record the effort of review, correction and fallback. Compare with a simpler search or plain automation wherever either could meet the need. A good AI answer isn't reason enough to choose AI if another approach meets the requirements and is easier to run.

Keep the limits of your sources and cases explicit. Volunteers with the builders on hand aren't the whole production population. An attractive average doesn't offset a serious disclosure or an unauthorized action. Revise scope and economics when the results call for it. The Prove decision is an investment decision. It can justify Build, a different design or stopping. It never grants permission for unrestricted reliance.

Build the obligations people will inherit

Turn the accepted slice into a repeatable release. Put in place product and tool contracts, source and meaning ownership, protected configuration, evaluation, deployment, monitoring, incident routes and limits. Test the incomplete and unavailable states the user will see. The normal answer is only part of it.

Have the intended operators do the work. They should investigate a request, revoke access, restrict a route and recover through the runbook. Include the people who maintain the documents and approve meaning. Fund their continuing workload and the review capacity it needs.

For writes, test the failure where the commit succeeds and the confirmation is lost. Set up a durable operation identity, preconditions and an authorized reconciliation path. Where the target can't support reliable duplicate protection, keep a narrower or manual process. Don't paper over the gap in a prompt.

A release decision says exactly what can be used, by whom, for what and under which conditions. If the full proposed scope isn't ready at week thirteen, say so. Carry on with Build with the gaps named, or release only the narrower capability the results justify on their own. Calendar progress can't supply missing proof.

Operate before declaring transfer

Review use, quality, exceptions, incidents and cost during ordinary work. Does the measured benefit survive once the special pilot support is gone? Check. And confirm the freed-up capacity is being used as planned before you call it a financial result.

Keep certification and release acceptance tied to current results. Source changes, an expired review, failed controls or loss of support can all restrict the right to rely on a capability. Restore it only after the cause is fixed and the proof supports the proposed scope.

Use the stewardship loop from Chapter 16 to cut the repeated work of gathering that proof. Start with one monitored dependency and one approved response. Keep disputed meaning with its owner, and measure whether the total burden of investigation and recovery actually falls.

Transfer is complete when the receiving team can carry its responsibilities with its own access, authority, skills and funded capacity. A repository and a slide deck can support that. They don't demonstrate it. And retire the capability deliberately when its value, or a workable way to run it, no longer supports keeping it.

Decide what the next capability can reuse

The second workflow should inherit the useful contracts, services and record-keeping practices. It doesn't inherit business approval for a different meaning, different users or different actions. Confirm which parts still hold and which need their own decision.

Treat reuse as an operating commitment. Another team needs documentation, compatibility, support and an owner who stays. If the next use still depends on the original builder explaining every step, improve the interface before calling the pattern scalable.

What I'm recommending is deliberately practical. Choose work that matters. Make its meaning and authority explicit. Test what happens when the system is wrong or unavailable. Measure the whole task, fund the responsibilities that continue and let the results decide the next scope.

Take one request to the people who own its result. Follow it until every assumption that matters has an owner and a test. It’s a good place to begin, and a habit worth keeping long after the first release.

APPENDIX A

Production readiness diagnostic

Use this diagnostic for one named workflow and its proposed scope. Have the business owner, technical lead and control owner answer separately, then reconcile their views. Keep the disagreements and the evidence gaps; they're the point. The questions follow the published RJL diagnostic. The added owner and evidence fields turn it into a working release aid. [15]

Record the workflow, environment, release, users, information, permitted actions, review date and decision holders. Answer Yes when something is implemented and demonstrated. Answer Partial for a defined remaining gap, No when it's absent, and Unknown when nobody can produce the evidence. An excluded capability needs a documented boundary and control-owner acceptance. It doesn't count as a demonstrated Yes.

Table 39. Production readiness questions

ID	Question	Evidence the answer needs
A1	Is there an owner and a measurable outcome?	Process authority, baseline, target, population, review and exceptions
A2	Are the information dependencies certified?	Current contracts, owners, lineage, quality and freshness for this use
A3	Is business meaning approved and tested?	Definitions, grain, context selection, authority and known- answer cases
A4	Does the knowledge path preserve source controls?	Applicable versions, permissions, replacement, deletion and stale-index tests
A5	Is identity enforced through the complete path?	Caller and workload scope, denied calls, revocation and tenant isolation
A6	Is the agent limited to approved capabilities?	Tool inventory, contracts, side effects, limits and refusal tests
A7	Does evaluation block an unsafe release?	Versioned pack, thresholds, failures, gate and stop authority
A8	Can one request be reconstructed?	Historical identity, sources, versions, policy, outcome and protected evidence
A9	Can the service fail safely and recover?	Objectives, limits, support, outages, recovery and write reconciliation where applicable
A10	Can the internal team operate it economically?	Accepted-task cost, adoption, business outcomes, funded maintenance and transfer demonstration

For each row, add the response, evidence reference, precise gap, accountable gap owner, target decision date and closure test. Controlled evidence references belong in the worksheet. Sensitive case records generally don't.

Worked gap record

This fictional record illustrates the method. It reports no actual service test.

Table 40. Worked gap record

Field	Illustrative entry
Workflow	Claims preparation, read-only, one approved product line
Question	A4, source controls in retrieval
Response	Partial
Evidence available	Permitted and prohibited document cases exercised in the example review
Gap	Revocation has not been demonstrated through the answer cache
Owner	Named retrieval service owner
Closure test	Revoke the test user's source right; repeat retrieval and cached-answer requests; demonstrate denial within the accepted window
Interim boundary	Do not release the affected restricted-content route until the critical gap is closed
Decision	Continue controlled testing; no production acceptance implied

Count responses only to describe their spread. Never convert them into a readiness percentage. A critical failure in access, isolation, unauthorized action, integrity or required evidence blocks the affected scope, whatever the other answers say. Ten Yes answers justify inspecting the evidence. They don't justify automatic release. Repeat the diagnostic after material changes in workflow, sources, users, tools or authority.

For stewardship agents, ask the same questions of the agent's own service: approved investigation scope, bounded repairs, independent decision criteria, protected evidence and measured consumer recovery. For cross-domain semantics, inspect the selected ontology versions and mapping contract. A shared term label is no proof of agreement.

APPENDIX B

Product contract and certification records

Use these records with Chapters 10 and 11. The examples adapt the RJL Certification Standard to the book's claims-status dependency. Identifiers, references, dates and limits are fictional. None of them records a deployed service, a published vendor schema or a certification issued to an organization. [13]

Example product manifest

The example: references below are symbolic. A real validator has to resolve controlled versions and compare the deployed configuration with the approved decision. Declaring a schema doesn't supply one. And the illustrative file isn't a ready-to-deploy implementation.

```
schema_version: book.example.product.v1
id: claims.claim-status
version: 1.0.0
domain: claims
reference_status: certified
delivery_stage: published
operating_condition: normal
owners:
  business: claims-data-owner
  steward: claims-status-steward
  service: claims-platform-operations
scope:
  purpose: examiner_preparation
  consumers: [claims-context-assistant]
  excluded: [adjudication, payments, external_contact, model_training]
interface:
  operation: get_claim_status
  kind: read_api
  contract_ref: example:claim-status-read-v1
  meaning_ref: example:claim-status-meaning-v1
  sources_ref: example:claim-status-authority-v1
  policy_ref: example:claims-read-policy-v4
  quality_ref: example:claim-status-checks-v1
service:
  max_source_watermark_age_minutes: 15
  on_stale: restrict_affected_response
  support_ref: example:claims-support-v1
  recovery_ref: example:claims-recovery-v1
evidence:
  evaluation_ref: example:status-evaluation-run-001
  deployment_ref: example:status-release-001
  decision_ref: example:status-acceptance-001
review:
  by: "2027-01-15"
  earlier_on: [source_change, access_expansion, material_breach, owner_change]
```

Keep lifecycle, assurance and current condition distinguishable. The reference_status field preserves the source standard's combined terminology, and the other fields explain it more precisely. A contributor who edits the word certified creates no acceptance. The protected decision has to authorize the exact version and scope. The system then reconciles the declaration to that decision.

The fifteen-minute limit is illustrative. Measure it from an authoritative source watermark or heartbeat whose meaning the source owner has accepted. The last business event alone tells you little about feed health. The serving contract has to specify the observation time, the permitted delay and the response when nobody can establish currency.

Example interface contract

The input is a validated claim identifier within a verified tenant and task scope. Authorization comes from trusted session and workload context. A model's assertion that the examiner has access counts for nothing. The permitted output contains status, rele-

vant effective time, source-update time, observation time and provenance, and leaves out fields preparation doesn't need.

The operation distinguishes five outcomes: success, not authorized, ambiguous identity, stale source and unavailable dependency. No message may reveal that a restricted record exists. The service never fabricates current status from a plausible earlier value. A consumer can use a historical response only under a separately approved purpose and freshness rule.

The contract names the source of authority, supported code values, treatment of reopened cases, known limitations, compatibility and migration. Its tests cover duplicate identifiers across tenants, unknown status codes, late events, conflicting sources, missing assignments and revoked access.

Completed illustrative acceptance record

This example assumes the referenced evidence exists within its fictional review. A real decision can't rely on symbolic references like these.

Table 41. Completed illustrative acceptance record

Field	Illustrative entry
Decision	status-acceptance-001; accept bounded examiner-preparation use
Product	claims.claim-status version 1.0.0; exact manifest and implementation references retained
Environment and interface	Approved production environment; get_claim_status only
Consumers	Authorized examiners through the approved context assistant
Evidence	Known-answer cases, boundary cases, reconciliation, access and revocation, load, restriction and recovery
Acceptance holders	Named business, control and service authorities, each within mandate
Exclusions	Adjudication, payments, external contact and training use
Continuing condition	Contracted currency and quality hold; support and evidence remain available
Review	15 January 2027 or earlier material trigger; dates illustrative
Restriction authority	Designated service incident authority can contain the affected route

Suspension and renewal example

A new source code appears outside the approved mapping. Detection records the source event and the affected scope. The product moves to degraded for the affected cases. It doesn't silently map the code to closed. The owner gets the code's approved meaning, assesses consumers and identifies any earlier use that needs review.

Engineering updates the mapping and the independently established expected cases. Then the team tests replay, late events, access, stale handling and the consumer's response. The authorized renewal record identifies the changed version, evidence, permitted scope, remaining conditions and next review trigger, and restoration follows that record. Neither an unchanged catalog badge nor an agent's suggested mapping can authorize it.

When ontology modules or cross-domain mappings affect the result, add their exact release identifiers and effective periods to the manifest and acceptance record. A release should resolve a compatible composition of meaning and implementation. Silently importing a domain's latest definition defeats that.

APPENDIX C

Capability ownership inventory

The sixty-three capabilities below come from the RJL Capability Ownership Worksheet. I've kept the names and scope and added technical clarifications for this edition. Read the grouping as a responsibility inventory. Nobody needs to buy sixty-three tools or appoint sixty-three people. [37]

For each capability, record the current owner, the accountable owner, delegated decision rights and any gap or action. The ownership field is blank on purpose. The companion CSV keeps those fields separate for you to complete; the book's final column has room for a compact owner-and-authority note.

AI and experience

The assistants, agents, models and analytics people work with, and the agent platform that runs them.

Table 42. AI and experience

ID	Capability and responsibility	Owner and decision rights
C01	Reporting and analytics. Dashboards and reports for monitoring performance and supporting decisions.	To assign
C02	Self-service exploration. Tools that let people explore approved data without a request queue.	To assign
C03	Data-driven applications. Applications whose core behavior depends on analytics or AI.	To assign
C04	Data science sandbox. Isolated environments for building and testing models on approved data.	To assign
C05	Prompt and instruction library. Reviewed, versioned prompts and instructions that teams reuse.	To assign
C06	Foundation models. General-purpose language and multimodal models, hosted or consumed as a service.	To assign
C07	Model adaptation. Fine-tuning and retraining models on domain data, with the evaluation to justify it.	To assign
C08	Agents. Software that plans and acts on a person's behalf within a bounded scope.	To assign
C09	Agent orchestration. Coordinates multi-step and multi-agent work, including hand-offs to people.	To assign

ID	Capability and responsibility	Owner and decision rights
C10	Agent memory. Stores what an agent may retain between steps or sessions, under retention rules.	To assign
C11	Agent evaluation. Tests agent behavior, refusals and tool use before release and after change.	To assign
C12	Agent catalog. A register of approved agents, their owners, scope and status.	To assign
C13	Agent observability. Traces of what each agent did, which tools it called and what it cost.	To assign

Access and interfaces

The governed doors into data and models: narrow interfaces that carry approved work.

Table 43. Access and interfaces

ID	Capability and responsibility	Owner and decision rights
C14	Data APIs. Versioned interfaces that expose data products for specific uses.	To assign
C15	Virtualization and federation. Querying across sources while controlling any temporary movement, caches and materializations.	To assign
C16	AI gateway. One controlled route to approved models, with logging, limits and cost tracking.	To assign
C17	Retrieval services. Search and retrieval over approved knowledge for AI, applying each user's permissions first.	To assign
C18	Agent tool interfaces. Narrowly defined tools, such as a status lookup, that an agent may call.	To assign
C19	Command path for writes. A separate, approved route for actions that change records, with duplicate protection.	To assign

Meaning and knowledge

What the data means: definitions, relationships and knowledge the business has approved.

Table 44. Meaning and knowledge

ID	Capability and responsibility	Owner and decision rights
C20	Metadata. Descriptions of data assets: what they contain, where they came from and how they may be used.	To assign
C21	Semantic layer. Business-friendly definitions of entities and metrics, versioned and tested.	To assign
C22	Ontology and knowledge graph. Governed concepts, entities and relationships; definitions may inform policy but do not grant access.	To assign

ID	Capability and responsibility	Owner and decision rights
C23	Information architecture. How information is organized so people and systems can find and use it.	To assign
C24	Master and reference data. The agreed records and code lists that other systems rely on.	To assign
C25	Chunking and embeddings. Preparing documents for AI retrieval while keeping source, version and permissions.	To assign
C26	Data marketplace and search. A place to find, understand and request access to data products.	To assign
C27	AI asset search. Discovery of reusable models, prompts, agents and tools.	To assign

Data platform

Where data is moved, stored and processed, and where access is enforced.

Table 45. Data platform

ID	Capability and responsibility	Owner and decision rights
C28	Batch integration. Moving data on a schedule.	To assign
C29	Streaming and messaging. Moving events and data in near real time.	To assign
C30	Pipeline orchestration. Scheduling, dependencies and recovery for data workflows.	To assign
C31	Data pipelines. Extract, load and transform, increasingly inside the platform itself.	To assign
C32	Schema detection. Recognizing and reconciling data structures as sources change.	To assign
C33	Data modeling. Designing structures for efficient processing and clear meaning.	To assign
C34	Data preparation. Cleaning, enriching and reshaping raw data for use.	To assign
C35	Data storage. Durable storage for structured and unstructured data, with retention rules.	To assign
C36	Vector storage. Storage and search for embeddings and associated metadata; source restrictions require explicit enforcement.	To assign
C37	Cloud hosting. The cloud environments the platform runs on.	To assign
C38	Containers. Portable runtime units for deploying data and AI services.	To assign
C39	Accelerated compute. GPUs and similar hardware for training and serving models.	To assign

Sources

Where the data starts. The architecture says which source is authoritative.

Table 46. Sources

ID	Capability and responsibility	Owner and decision rights
C40	Purpose-built databases. Transactional and analytical stores built for specific workloads.	To assign
C41	Operational applications. Core business systems that create and change records.	To assign
C42	External and web data. Licensed, public or collected outside data, with its terms of use.	To assign

Security across every layer

Identity, classification, enforcement and response for people and agents alike.

Table 47. Security, across every layer

ID	Capability and responsibility	Owner and decision rights
C43	Identity and access for people. Who a person is and what they may do, granted and revoked through one process.	To assign
C44	Identity for agents. Attributable workload identity, role, sponsor and bounded delegated or independently approved authority.	To assign
C45	Single sign-on and zero trust. Federated sign-in and separately enforced identity, device and least-privilege decisions at relevant boundaries.	To assign
C46	Classification and labels. Governed sensitivity vocabulary mapped and tested across each supported enforcement surface.	To assign
C47	Data protection. Masking, encryption and loss prevention for sensitive data.	To assign
C48	Monitoring and logging. Recording and reviewing access and activity across systems.	To assign
C49	Threat detection and response. Finding and handling attacks, including instructions hidden in content.	To assign

Orchestration and operation across every layer

How work is released, run and evidenced: process, quality, cost and the record of what happened.

Table 48. Orchestration and operation, across every layer

ID	Capability and responsibility	Owner and decision rights
C50	Data quality. Measuring accuracy, completeness and freshness against agreed thresholds.	To assign
C51	Lineage. Tracing data from source to consumption.	To assign
C52	Data architecture standards. Enterprise patterns and rules that teams build to.	To assign

ID	Capability and responsibility	Owner and decision rights
C53	Responsible AI. Reviews and tests for fairness, safety and appropriate use.	To assign
C54	DataOps. Practices that make data pipelines reliable, observable and repeatable.	To assign
C55	DevOps. Continuous integration and deployment for software and data.	To assign
C56	Model and AI operations. Deploying, monitoring and retiring models and agents in production.	To assign
C57	FinOps. Tracking and controlling cost, down to cost per completed task.	To assign
C58	Release gates. Tests and approvals that must pass before a change reaches users.	To assign
C59	Request evidence. A record of each request: who asked, which versions, which policy decisions and what happened.	To assign

Operating model

The people, skills and funding that keep the capabilities running.

Table 49. Operating model

ID	Capability and responsibility	Owner and decision rights
C60	Data and AI strategy. Aligning data and AI work with business priorities and decisions.	To assign
C61	Upskilling and training. Building the skills people need to use and run these capabilities.	To assign
C62	Research and talent programs. Bringing in new talent and testing emerging approaches.	To assign
C63	External workforce. Engaging partners and contractors for flexible, specialist capacity.	To assign

Worked assignment

The following allocation is fictional and applies only to the distributor's internal read-only order assistant.

Table 50. Worked assignment

Capability	Illustrative accountable role	Delegated decision and boundary
Agents	COO as business owner	Approve preparation scope and accept outcome; cannot waive source rights
Agent tool interfaces	Service engineering lead	Maintain approved read tools; no business authority to release holds
Classification and labels	Designated security owner	Approve handling rules and review source coverage within mandate

Capability	Illustrative accountable role	Delegated decision and boundary
Release gates	Named release coordinator	Verify required approvals; cannot substitute their own acceptance for another authority
Request evidence	Service owner	Maintain protected records and reviewer access under retention policy

Complete the whole inventory by recording scope. Repeating a senior title in every row tells nobody anything. When two functions seem to own the same capability, split the actual decisions and interfaces. When a proposed owner lacks time, funds or authority, record a gap. Fix the consequential gaps in the first workflow first. The inventory isn't a maturity score.

Map the evolving roles in Chapter 16 onto this inventory. Don't create a separate shadow organization for agents. A stewardship agent is an operated capability with a sponsor and bounded authority. It never replaces an accountable human owner in the inventory.

APPENDIX D

Product fit and replacement tests

I checked the product observations in this appendix on 19 October 2026. Refresh them before any implementation or publication decision. Documentation establishes described behavior and prerequisites. It can't establish fit in your estate, or that one product beats another. Examples aren't endorsements.

Selected retrieval and protection observations

Table 51. Selected retrieval and protection observations

Example	Documented boundary	Evaluation needed
Snowflake Cortex Search	Search uses owner rights; service-query privileges differ from source-table rights	Verify caller-specific restrictions, protected query paths, source changes and revocation [26]
Databricks AI Search	Native row- and column-level index permissions are not supported; application ACL filters are an option	Carry complete metadata, prevent filter bypass and test each supported access path [27]
Azure AI Search	Security-filter patterns are generally available; native document-permission integrations include preview capabilities	Verify selected source, API, identity mechanism, synchronization and preview terms [28]
Cyera and Collibra	Published available integration enriches catalog assets; optional inventory creation [54]	Demonstrate asset matching, accepted metadata, schedule and destination visibility
Cyera and Snowflake	Vendor announcement distinguishes GA masking/risk analysis from agent- discovery preview [55]	Verify current feature status and native enforcement for the actual account
Microsoft Purview Data Map	Source-specific capabilities; Snowflake catalog scanning is not Snowflake policy application [59]	Check connector, labeling surface, native policy route and derived copies
Cursor	Rules guide model context; MCP connects tools; protection depends on the configured services [61; 62]	Verify approved metadata, scoped tool identity, code review and runtime checks

Test the actual permission and protection behavior of every route you select. Index governance and document entitlement are different things. A search result has to meet the task's applicability and currency requirements as well as its access rule.

Model access paths

I checked these observations on 29 October 2026 for the platform decision guide. They'll change quickly. [88] The same model name can arrive through different contracts, with different data handling, features and quotas.

Table 52. Selected model access observations

Example	Documented behavior	Evaluation needed
Claude in Microsoft Foundry	Offered Azure-hosted or Anthropic-hosted; Microsoft states the Anthropic-hosted option runs outside Azure, and Data Zone deployment covers only listed Azure-hosted models	Confirm which hosting option each deployment uses, its region and the data terms that apply
OpenAI models on Amazon Bedrock	GPT-5 and GPT-6 family models generally available on Bedrock during 2026	Verify feature parity with the vendor API, regional and cross-region settings, quotas and terms
OpenAI models on Google Cloud	Announced in private preview on 21 October 2026	Treat as preview until general availability is confirmed for your account and region
Claude on Google Cloud	Google-managed and Anthropic-managed offerings with different billing, rate-limit ownership and endpoint options	Confirm who operates inference, the processing location and which features each offering supports

Component evaluation record

For any candidate, record the responsibility, existing alternative, exact product and edition, deployment region, status, licensing, supported interfaces, identity model, data rights, performance and recovery. Mark each important feature as documented, demonstrated in the selected environment, preview, announced or unverified. Documentation and demonstration are separate kinds of evidence. Run a real question through the proposed combination. A metadata catalog supports cross-system discovery and meaning. A semantic interface supports governed calculations. A posture tool helps identify exposure, and a gateway governs supported model or tool traffic. Test the actual enforcement boundaries; category labels won't tell you where they are. Power BI and Tableau should consume the intended definitions under their own verified access, extract and cache arrangements.

Volatile vendor matrices live here so the main argument doesn't hang on a release date. The more detailed source-paper examples are still useful research leads. Their integration and availability claims need fresh official verification before you adopt anything. [11]

Replacement rehearsal

1. Select a bounded representative contract, dataset and evaluation pack.

2. Export definitions, mappings, policy references and required history; identify what the export omits.
3. Implement the candidate replacement under its actual identities and limits.
4. Compare accepted outputs, denials, freshness, failure states, cost and recovery.
5. Test consumer migration and the ability to investigate prior requests.
6. Revoke old credentials, stop jobs and reconcile retained copies, commitments and deletion obligations.

Record the effort, the artifacts you couldn't get and the dependencies left behind. Similar endpoint names prove nothing about equivalence. Some switching cost may be acceptable. Make its scope and consequence a deliberate decision.

APPENDIX E

Rules by jurisdiction and use

This is a selected applicability guide, checked on 19 October 2026. It isn't a complete compliance inventory. Before assigning a duty, identify the entity, jurisdiction, activity, affected people and legal role. Keep current official texts and applicable local measures in the record. The controls this book recommends don't replace the applicable legal or supervisory process.

United States

Bank model risk. Federal Reserve SR 26-2 was issued on 17 April 2026; it was not enacted as legislation. The revised interagency guidance supersedes SR 11-7 and SR 21-8 and is nonbinding. It is expected to be most relevant to Federal Reserve-regulated banking organizations with assets over $30 billion; the guidance also describes circumstances in which it can be relevant below that size. Generative and agentic AI are outside its stated scope. That exclusion does not exempt an entire mixed workflow or remove other obligations. Assess covered models separately and name the appropriate internal governance for tools outside scope. [24]

Credit decisions. Regulation B, 12 CFR 1002.9, requires applicable adverse-action notices and specific principal reasons that accurately reflect factors actually considered or scored. The rule does not categorically prohibit language-model assistance in expressing those reasons, nor require every underlying decision to use deterministic scoring. An invented, unsupported or misleading rationale fails the actual-reasons requirement regardless of how it was written. Preserve the decision factors and verify any rendered explanation against them. Notification alternatives and detailed scope still matter. CFPB Circular 2022-03 was withdrawn on 12 May 2025; use the current regulation and commentary as the governing reference. [31; 38]

Insurance. The NAIC adopted its model bulletin on insurer AI use in December 2023. The bulletin is not a single nationwide AI statute. Check the relevant state's adopted measures, the entity and the use, including governance, third parties and consumer outcomes. A claims preparation tool, an underwriting model and a pricing model can each need a different assessment. [29]

New York cybersecurity. NYDFS's 16 October 2024 AI cybersecurity letter explains existing Part 500 obligations and states that it adds no new requirements. Determine covered-entity status and applicable provisions; link evidence to the relevant risk, access, supplier and operating controls. Other insurance-specific requirements need their own applicability review. [30]

European Union

AI Act. Classify the intended use and the actor role. Don't assume all financial-services AI is high-risk. Human involvement or read-only output doesn't, by itself, settle classification. Article 14 addresses oversight of high-risk systems, including the risk of excessive reliance. [23]

The AI Act retains general application from 2 August 2026, with exceptions. Regulation (EU) 2026/1744 amends Article 113: Chapter III Sections 1, 2 and 3, except Article 6(5), apply from 2 December 2027 for Article 6(2)/Annex III high-risk systems and 2 August 2028 for Article 6(1)/Annex I systems. These dates cover specified provisions. They do not postpone every AI obligation. [33; 46]

The amending regulation was published on 24 July 2026 and entered into force on 27 July. It also changes AI-literacy wording and other provisions, with separate transitions, including for newly added prohibitions. Don't assume every accelerated obligation stayed the same. Check the consolidated duty, actor role and transition for the actual system. [46]

Operational resilience. DORA, Regulation (EU) 2022/2554, has applied from 17 January 2025. Its covered financial entities need the applicable ICT-risk, supplier and incident arrangements; EBA's register guidance explains the register of ICT third-party contractual arrangements. This is separate from AI Act classification. [35]

United Kingdom

Bank model risk. The PRA's SS1/23 page identifies its relevant population and current April 2026 version. It is not a rule for every UK company. In-scope firms should map model governance, independent validation and mitigants to the models and uses concerned. [34]

Personal information and automated decisions. The Data (Use and Access) Act 2025 amended UK data-protection law. ICO's guidance updated in June 2026 describes changed automated-decision provisions and continuing safeguards, with stronger protection for special-category information. Check commencement, the provision in force and current guidance for the particular processing. Do not apply an old UK GDPR Article 22 summary or EU rules as though the UK position were unchanged. [32]

Voluntary frameworks and internal standards

NIST AI RMF 1.0, its Generative AI Profile and SP 800-218 SSDF v1.1 are voluntary references. The RJL Certification Standard and book templates are proposed internal practices. Chapter 15 maps software-delivery activities to SSDF practice identifiers; that mapping does not certify AI accuracy or regulatory conformity. An internal certified label is not external accreditation. [19; 20; 52; 13]

Applicability record

Use one row per requirement: source and current version; binding law, supervisory guidance, voluntary framework or internal policy; entity and activity; jurisdiction and role; application date and transition; accountable legal or control owner; implementation; test and evidence; open question; next review trigger. Record each exclusion and its rationale. Never delete an uncertain entry just to get an all-green checklist.

Incident deadlines and retention schedules need assessment record by record. Different regimes start different clocks. A single generic seventy-two-hour incident rule, or a six-month logging rule, can’t substitute for that analysis.

APPENDIX F

Economics and operating templates

These templates support Chapters 1, 6, 13, 17 and 20. Fill them in with evidence from the workflow you've selected. The examples are illustrative. The forms confer no approval by themselves.

The twelve-cost worksheet

Keep implementation, recurring operation, transition and exit visible. Enter existing staff capacity separately from new spending. State the currency, period, assumptions and allocation policy. Not every cost is new cash, and not every license is a one-time cost.

Table 53. The twelve-cost worksheet

Cost category	Inputs to estimate	Budget or capacity owner
Data preparation and upkeep	Sources, mappings, definitions, corrections and product maintenance	To assign
Knowledge maintenance	Content ownership, parsing, indexing, permission and deletion propagation	To assign
Production usage	Model, retrieval, compute, API, storage and transfer drivers	To assign
Evaluation and drift monitoring	Datasets, reviewers, re-evaluation, instrumentation and investigations	To assign
Human review and exceptions	Eligible cases, review minutes, exception effort and cover	To assign
Security and agent identity	Identities, policy, access reviews, adversarial testing and response	To assign
Compliance and audit evidence	Applicability work, protected records, inspection and retention	To assign
Support and operation	Service hours, incidents, maintenance, recovery and supplier changes	To assign
Change and training	Changed roles, protected learning, guidance and refresher work	To assign
Unsanctioned-use remediation	Discovery, approved replacement and controlled cleanup	To assign
Portability and exit	Contract commitments, exports, migration and replacement tests	To assign

Cost category	Inputs to estimate	Budget or capacity owner
Retirement	Consumer migration, access removal, retained records and decommissioning	To assign

Add columns for amount or hours, fixed or variable driver, start and end period, allocated share, incremental cost, and avoidable cost with its timing. The three views have to reconcile to the portfolio. None of them may claim that assigned cost automatically disappears on exit. [16]

Worked capacity and realization example

The planning example in Chapter 1 assumes 120,000 eligible annual cases, 60 percent adoption and six net minutes saved after review and correction. Here, accepted adopted cases come to 72,000, because the example assumes adopted cases meet the stated completion criteria. In real measurement, account for rejected and incomplete work separately.

Released hours = 120,000 × 0.60 × 6 ÷ 60 = 7,200 hours. At $65 loaded cost, capacity-equivalent value is $468,000. Take off $220,000 of incremental annual operating cost and net capacity-equivalent value is $248,000. None of these numbers establishes cash savings.

Now suppose, separately and for illustration, the owner can avoid 2,000 contractor hours at a verified $65 per hour in the same year. Planned avoided spending is $130,000. Against the example's $220,000 incremental run cost, the cash-and-avoidable-cost view comes out at negative $90,000, before implementation and transition. The rest of the released capacity may improve service. It mustn't be added again as cash. That's how a favorable capacity case can sit alongside an unattractive immediate cash case.

For an investment decision, add the implementation, transition and exit cash flows, and apply the organization's investment method. The source workbook is an operating forecast. It isn't an NPV or IRR model. [16]

Benefits contract

Record the workflow and useful completion; baseline volume and effort; eligible population; measured acceptance; review and exception effort; comparison method and window; released hours; specific realization action; amount and timing; accountable business owner; finance validation; dependence on staffing or supplier changes; and next evidence review. Keep forecast, measured effect and realized result in separate fields.

Agent operating contract

Add the context specification version, required runtime fields, context-record reference and revision, permitted model-facing attributes, validation and clarification rules, update triggers and enforcement interfaces. Say how a change of subject or purpose invalidates earlier assumptions, and where the service checks current authorization.

Add validation limitations, restriction-metadata completeness, permissible supplier processing and the current revocation window. For writes, specify atomic state preconditions, intent identity retention, target reconciliation and handling of late arrivals. For evaluation, record independent expected outcomes, protected holdouts and known coverage gaps. With these fields, the contract reflects the implementation obligations the chapters discuss.

Complete the following fields: task and population; business sponsor; product and service owners; approved dependencies and definitions; identity and delegated scope; tools and side effects; exclusions; approval binding and expiry; state preconditions; operation identity and target duplicate protection; reconciliation of unknown outcomes; cost, duration and concurrency limits; memory and retention; evidence; fallback; release version and review triggers.

For a stewardship agent, also specify its authority tier, observable trigger, allowed assets, approved runbook, maximum affected scope, independent postcondition, stop condition and restoration authority. Protect the tests and approval policy so the agent can't modify them. Record the ontology and mapping versions used to interpret the incident.

Take the illustrative internal-note capability. Approval binds one exact note to one permitted claim, its relevant version and an expiry. The target service owns atomic preconditions, duplicate protection and durable results. A lost response produces unknown, not failed. Query durable status, or retry the same operation only where the target's idempotency contract permits it. An application-side check followed by an unprotected write stays unsafe under concurrency. [12; 22; 70; 72]

Request evidence record

Include the context-record revision and the approved view supplied to each model invocation, with protected links to source versions, policy decisions and material clarifications. Keep verified, user-supplied and inferred attributes distinguishable. When context stays unresolved, record that condition and the resulting restriction as an observed outcome.

An illustrative record identifies a read-only preparation request, its verified examiner and workload, purpose, claim scope, release, definition and procedure versions, source observations, policy decision, tool results and returned state. It records whether the answer

was complete, denied or escalated, and the relevant review. Symbolic example identifiers belong only in teaching. Real records resolve to protected artifacts.

The working specification should also state excluded sensitive fields, credential redaction, authorized reviewers, integrity checks, retention, legal holds, deletion and evidence-pipeline monitoring. Where you need the content, a source hash can't substitute for keeping it. And a later generated explanation never replaces the historical record.

Release decision record

Table 54. Release decision record

Field	Entry to complete
Release and environment	Exact combination and deployed location
Scope and exclusions	Users, purpose, sources, actions and prohibited behavior
Business evidence	Accepted tasks, baseline comparison, review burden, outcome and limits
Control evidence	Allowed and denied paths, revocation, isolation, action and evidence tests
Operational evidence	Load, failures, limits, cost, recovery, support and transfer
Unresolved conditions	Gap, consequence, restriction, authority, closure evidence and expiry
Decision and authority	Release, restrict, continue testing, reshape or stop; named holders
Review and restoration	Dates, earlier triggers, containment authority and return conditions

In a fictional completed example, the owners accept a read-only claims preparation release for one product line after reviewing the stated evidence. Notes, payments, adjudication and external contact stay excluded. The service owner can restrict the route on an entitlement or required-evidence failure. Any expansion needs a new impact assessment and acceptance. The example assumes a completed review. It doesn't report tests performed for this book.

Retirement record

Record the proposed reason and decision authority, the usage window including periodic consumers, downstream dependencies, replacement and migration, user communication, schedules and jobs to stop, tools to withdraw, identities and credentials to revoke, source and derived stores to handle, retained evidence and legal holds, supplier commitments and the costs actually released.

In an illustrative distributor retirement, an approved successor replaces the assistant and the order system stays. Service agents test the new route. Downstream tool consumers confirm migration. The team disables the old identities and jobs, cached answers and in-

dexes follow their approved deletion rules, and historical evidence stays under restricted retention. Verify that it’s complete. An empty usage chart proves nothing.

The service is retired only once its execution paths and obligations have been reconciled. A budget saving should say which commitments actually ended, and when.

Glossary

Accepted task. A completed unit of work meeting the business's stated criteria. Attempts, responses and sessions are not automatically accepted tasks.

Agent. Software that selects and carries out steps or tool calls within an approved task. The surrounding system, not the model's assertion, grants execution authority.

Agent contract. The versioned commitment describing task, consumers, dependencies, tools, limits, evidence, ownership and operation.

Applicability. The conditions under which information, a rule or an obligation serves the particular case, purpose, date and jurisdiction.

Authentication. Establishing an identity through a trusted mechanism.

Authorization. Deciding and enforcing which operation an identity may perform on which resource under the relevant conditions.

Avoidable cost. Cost that actually ceases, with stated timing and transition conditions, if a workflow or service stops.

Bitemporal record. A representation preserving valid-time and recorded-time intervals and correction history, so queries can distinguish business applicability from what was known at a prior recorded time. Business approval. An authorized decision accepting a specific proposal, scope or result. It is distinct from a successful sign-in or a model's confirmation flag.

Canary release. A controlled introduction to a limited population or traffic share, with defined observation and rollback conditions.

Capacity released. Net work hours made available after review, correction and exceptions. It becomes a financial saving only through a specific realization action.

Certification. In this book, an internal, scoped acceptance of a version and its evidence under an adopted standard. It is not external accreditation.

Compensation. An authorized business operation that remedies an earlier effect; different from reversing an application deployment.

Context assembly. The application function that resolves, validates and versions request context from trusted services, governed requirements and necessary user clarification.

Context record. The versioned runtime record of a request's caller, purpose, subject, scope, meaning, dates, provenance and validation state; distinct from the governed specification and the limited view sent to a model.

Context specification. The governed and versioned requirements for interpreting a task, including required context fields, authoritative sources, definitions, dates, conflict rules and behavior when a requirement cannot be resolved.

Control plane. Services and authoritative records coordinating identity, contracts, policy, versions, approvals, condition and evidence across the capability.

Correlation identifier. A reference associating events belonging to a request or operation. It is not itself an access credential.

Data plane. The paths carrying reads, retrievals, commands, events and results.

Data product. An owned unit of information or capability with a defined consumer, supported interface and maintained contract.

Degraded condition. A recorded breach or uncertainty requiring the contract's restriction, disclosure or stop response.

Derived product. A product that combines or transforms supported products for a consumer purpose and owns the new meaning, lineage and service commitments it creates, along with the restrictions it inherits. Derived does not have to mean several domains or aggregation.

Domain ontology module. A governed model for a business area or purpose, with explicit boundaries and mappings to concepts maintained elsewhere. DSPM. Data security posture management: discovery and assessment of data, sensitivity, access and exposure, with product-specific remediation and integration capabilities. Its findings inform an approved protection design; they do not certify business fitness or authorize use.

Effective authority. The permitted operation after applying the relevant caller, workload, tool, purpose and context restrictions together.

Embedding. A numerical representation used for tasks such as similarity search. It does not establish meaning, source authority or permission.

Evidence. Inspectable records supporting a statement about the system, its decisions or observed behavior. Integrity alone does not establish truth or completeness.

Foundational product. A product that publishes a stable contract around one domain capability, close to where the information is created. Foundational does not mean raw.

Freshness. Currency measured against a specified clock and population, with a declared permitted delay and failure response.

Governance as code. Versioned, machine-readable governance intent connected to authorized decisions, validated deployment, runtime controls and reconciliation of actual state. Policy as code implements a narrower set of executable rules within that lifecycle.

Grain. What one record or measurement represents, such as one claim, payment or event.

Idempotency. A defined operation contract under which repeats of the same intended request do not create additional unintended effects. Scope, concurrency and retention matter.

Incremental cost. Additional cost caused by a specific change in volume, capability or scope.

Knowledge graph. Connected entities and relationships, potentially including vocabulary, instances and inferred links, with implementation-specific query and reasoning behavior.

Lineage. Recorded relationships between sources, transformations, products and consumers. Its coverage and method determine what it can establish.

Manifest. A structured declaration referring to the product's contracts, owners and evidence. It does not certify itself.

Ontology. An explicit model of concepts, relationships and axioms within a domain; its reasoning semantics and application validation requirements must be distinguished.

Ontology composition. The pinned combination of shared and domain modules, mappings, shapes and calculation rules accepted for a consumer.

Persona. A description of a user perspective or an assistant's communication role; it does not establish verified identity, entitlement or business facts.

Policy decision point. A responsibility that evaluates the relevant authorization policy.

Policy enforcement point. A responsibility that applies the decision to the actual access or operation. Prompt injection. An attempt to make supplied content act as instructions that redirect the model or surrounding workflow.

Protection context. The governed information needed to decide and demonstrate appropriate handling for an asset and purpose, including provenance, accepted classifications, ownership, identity scope, applicable policy and observation time.

Recorded time. The coordinate at which an assertion belongs to the system's accepted history. Its transaction and visibility semantics must be defined; a timestamp alone is not a global commit order.

Reconciliation. Comparing records or outcomes with an authoritative reference or durable target state and resolving differences under a defined process.

Release combination. The named configuration of model, prompts, retrieval, definitions, tools, policy, dependencies and interface accepted for a scope.

Retrieval-augmented generation. Supplying selected external information to a model to support a response. Retrieval needs its own applicability, permission and freshness controls.

Semantic layer. A governed representation of metrics, dimensions and calculation rules for consistent consumption. Service-level indicator. A defined measurement of service behavior experienced by its consumer.

Service-level indicator, objective and agreement. An indicator measures behavior, such as the share of daily publications delivered on time. An objective is the target for that measure over a stated window. An agreement adds the consequences of missing it; not every internal target needs one.

Stewardship agent. Software that gathers governance evidence, proposes changes or executes explicitly bounded runbooks under a human-owned operating contract. It does not hold independent business authority.

Taxonomy. An organized classification of concepts or categories, often hierarchical; it need not express the richer relationships of an ontology.

Tenant boundary. A defined isolation scope, such as a client, legal entity or authorized population; its meaning depends on the system's policy.

Unknown outcome. An operation whose committed result has not been established. A timeout can produce this state without proving failure.

Valid time. When a fact or rule applies in the business, distinct from when the system recorded it.

Workload identity. A trusted identity assigned to software or a service for a defined operating purpose.

Notes and sources

Numbers in square brackets refer to the source records below. Access dates are recorded with the sources; the technical additions were checked on 19 October 2026. For this edition, 31 October 2026, the RJL paper entries were updated to their current published versions and the platform decision guide [88] was added. RJL source papers are the author's foundations. The governance-agent operating model, examples and templates are proposed designs, not reports of deployed client systems.

[1] Richard White. Why Enterprise AI Stalls After the Demo. RJL Solutions, 2026. 10 PDF pages. rjlsol.com/library/stalls

[2] Richard White. The Connected Enterprise Data & AI Blueprint. RJL Solutions, 2026. 24 PDF pages. rjlsol.com/library/blueprint

[3] Richard White. Adoption by Design: The Change Management That Decides Whether Enterprise AI Is Used. RJL Solutions, 2026. 16 PDF pages. rjlsol.com/library/adoption-by-design

[4] Richard White. The 90-Day First Governed Capability. RJL Solutions, 2026. 15 PDF pages. rjlsol.com/library/90-day-first-capability

[5] Richard White. The Data and AI Operating Model. RJL Solutions, Product operating review update, 29 October 2026. 17 PDF pages. rjlsol.com/library/operating-model

[6] Richard White. Data Products in Practice. RJL Solutions, Version 3, Expanded guide, 29 October 2026. 26 PDF pages. rjlsol.com/library/data-products-in-practice

[7] Richard White. The Federated Data Platform. RJL Solutions, Architecture update, 29 October 2026. 19 PDF pages. rjlsol.com/library/federated-data-platform

[8] Richard White. Context Before Query. RJL Solutions, Architecture update, 29 October 2026. 14 PDF pages. rjlsol.com/library/context-before-query

[9] Richard White. The Four-Part System. RJL Solutions, 2026. 16 PDF pages. rjlsol.com/library/four-part-system

[10] Richard White. The Reference Implementation: AI, Data and Security as One System. RJL Solutions, 2026. 22 PDF pages. rjlsol.com/library/reference-implementation

[11] Richard White. Patterns and Product Fit: Securing AI Retrieval at Enterprise and Lean Scale. RJL Solutions, Architecture update, 29 October 2026. 19 PDF pages.

rjlsol.com/library/patterns-product-fit

[12] Richard White. Agent Onboarding: Treating an Agent Like a Data Product. RJL Solutions, Version 2 with product consumer supplement, 29 October 2026. 14 PDF pages. rjlsol.com/library/agent-onboarding

[13] Richard White. Data Product Certification Standard. RJL Solutions, Product construction supplement, 29 October 2026. 9 PDF pages. rjlsol.com/library/data-product-certification

[14] Richard White. AI Governance Evidence. RJL Solutions, Architecture update, 29 October 2026. 19 PDF pages. rjlsol.com/library/ai-governance-evidence

[15] Richard White. The 10-Question AI Production-Readiness Diagnostic. RJL Solutions, 2026. 4 PDF pages. rjlsol.com/library/production-readiness-diagnostic

[16] Richard White. AI Token Economics. RJL Solutions, Version 2 with product economics supplement, 29 October 2026. 22 PDF pages. rjlsol.com/library/token-economics

[17] Richard White. Data Is Not Just Code. RJL Solutions, Version 3 with product construction supplement, 29 October 2026. 19 PDF pages. rjlsol.com/library/data-is-not-just-code

[18] Richard White. Rewiring Data Management with AI. RJL Solutions, Version 1 with architecture update, 29 October 2026. 23 PDF pages. rjlsol.com/library/rewiring-data-management

[19] NIST. AI Risk Management Framework 1.0. January 2023; current overview checked October 2026. Official source. Voluntary status, published version and revision status; Chapter 13 and Appendix E.

[20] NIST. Artificial Intelligence Risk Management Framework Generative Artificial Intelligence Profile. NIST AI 600-1, July 2024. Official source. Profile identity and risk reference; recommendations remain use-specific.

[21] OWASP Cheat Sheet Series. RAG Security Cheat Sheet. Living resource, accessed 19 October 2026. Official source. Retrieval security reference for Chapter 14; pin the implementation reference at adoption.

[22] Malcolm Featonby, Amazon Builders' Library. Making retries safe with idempotent APIs. Living publication, accessed 19 October 2026. Official source. Idempotent request identity and uncertain outcomes; not an AWS deployment requirement.

[23] European Commission AI Act Service Desk. Article 14 Human oversight. Current published text, accessed 19 October 2026. Official source. High-risk oversight and automation bias; applicability and application date must be separately established.

[24] Federal Reserve; OCC; FDIC. Revised Guidance on Model Risk Management. 17 April 2026, SR 26-2 and attached guidance. Official source. Scope, nonbinding status and supersession. Generative/agentic exclusion checked in the official guidance: https://www.federalreserve.gov/frrs/guidance/supervisory-guidance-on-model-risk-management.htm

[25] Team Topologies. Key concepts. Living author resource, accessed 17 October 2026. Official source. Team types and interaction vocabulary. The book's organizational allocation is an RJL application.

[26] Snowflake. Cortex Search overview. Documentation accessed 19 October 2026. Official source.

Required privileges and owner-rights search; verify deployed configuration separately.

[27] Databricks. AI Search. AWS documentation, accessed 19 October 2026. Official source. Limitations section: row/column permissions and application ACL filters.

[28] Microsoft Learn. Document-level access control in Azure AI Search. Documentation accessed 19 October 2026. Official source. Security filters, native preview approaches and permission synchronization.

[29] NAIC. Artificial Intelligence. Official topic page, accessed 19 October 2026. Official source. December 2023 model bulletin and current program context; individual-state applicability not established by this book.

[30] New York State Department of Financial Services. Cybersecurity Risks Arising from Artificial Intelligence and Strategies to Combat Related Risks. Industry letter, 16 October 2024. Official source. Explains existing Part 500 duties; no new requirements imposed by the letter.

[31] CFPB. Regulation B section 1002.9 Notifications. Current regulation accessed 19 October 2026. Official source. Adverse-action notification and principal reasons; detailed scope and alternatives remain important.

[32] UK Information Commissioner's Office. The Data Use and Access Act 2025 What does it mean for organisations. Updated 19 June 2026. Official source. Current amendments and safeguards; do not reuse an unchanged pre-DUAA Article 22 summary.

[33] European Commission. AI Act overview and application timeline. Accessed 19 October 2026. Official source. Current reported Omnibus and dates, cross-checked with https://ai-act-service-desk.ec.europa.eu/en/ai-act/article-113 . The EUR-Lex record of the amending regulation was confirmed on 18 October 2026; see [46].

[34] Prudential Regulation Authority. SS1/23 Model risk management principles for banks. Current version published and effective 23 April 2026. Official source. Relevant firm population and current version; corrects reliance solely on the May 2023 edition.

[35] European Banking Authority. Preparations for reporting of DORA registers of information. Accessed 19 October 2026. Official source. 17 January 2025 application and ICT third-party registers.

Governing regulation: Regulation (EU) 2022/2554.

[36] Charles Conn and Robert McLean. Bulletproof Problem Solving The One Skill That Changes Everything. Wiley, March 2019. Official source. Publisher catalog and authorized contents/excerpt used for edition and general problem-definition reference; not a full-book review. ISBN paperback 9781119553021. Publisher contents: https://catalogimages.wiley.com/images/db/pdf/9781119553021.toc.pdf

[37] Richard White, RJL Solutions. Capability Ownership Worksheet. October 2026, 63 entries. Official source. Names and scope retained; descriptions technically clarified for this edition.

[38] CFPB. Withdrawn Guidance. Accessed 19 October 2026. Official source. Lists Circular 2022-03 as withdrawn 12 May 2025; distinct from the continuing regulation.

[39] Barbara Minto. The Minto Pyramid Principle Logic in Writing Thinking and Problem Solving. 1996 edition, author's edition description. Official source. General author-recommended communication reference; no detailed quotation or page claim. Confirm preferred edition for the final bibliography.

[40] Model Context Protocol. Security Best Practices. 25 November 2025 documentation version. Official source. Further technical reading on token audience and passthrough. Not a claim of implementation conformance.

[41] NIST. Zero Trust Architecture. SP 800-207, August 2020. Official source. Further technical reading on authentication and authorization boundaries.

[42] McKinsey & Company. The state of AI in 2026 On the road to ROI. 25 August 2026. Survey of 1,719 respondents in 97 nations, fielded 4 May to 8 June 2026 and weighted by national GDP. Enterprise scaling and self-reported EBIT impact; the high-performer definition also requires significant impact. Accessed 19 October 2026. Source report

[43] Stanford Digital Economy Lab. AI Economic Indicators June 2026 Update. Research Note No. 1, updated 10 August 2026. Takeoff Tracker summary as of May 2026; selected balanced ADP payroll sample, not the whole US labor market. Employment associations do not establish AI causation. Accessed 19 October 2026. Source report

[44] PwC. AI and the insurance workforce Enabling the human AI organization. 27 January 2026. Practitioner analysis of expertise concentration, entry-level work, managerial responsibility and career development. Accessed 19 October 2026. Source article

[45] Thomson Reuters Institute. Future of Professionals Report 2026 Legal. Fielded March and April 2026; 736 law-firm responses across 46 countries and 203 corporate legal-department responses. The preface statistics concern the stated respondent groups, not all professional-services firms. Accessed 19 October 2026. Source report

[46] European Union. Regulation EU 2026/1744 of 8 July 2026, Digital Omnibus on AI. Official Journal L, 24 July 2026, 41 pages. Article 1 point 40 amends AI Act Article 113; Article 4 provides entry into force on the third day after publication. Governing text checked 18 October 2026; amended Article 113 cross-checked against the Commission's consolidated text on 19 October 2026. Official text

[47] Constellation Research. Meet the 150 Artificial Intelligence Executives Driving an AI Powered Future. 2 July 2024. Inaugural 2024 to 2025 AI150 list includes Richard White, then listed as Chief Data Officer at New York Life. Accessed 18 October 2026. Source announcement

[48] NIST. Guide to Attribute Based Access Control ABAC Definition and Considerations. SP 800-162, January 2014, updated 2 August 2019. Accessed 19 October 2026. Subject, object, operation and environment attributes inform policy evaluation; the book's context-record schema is an author-proposed pattern. Official source

[49] W3C. OWL 2 Web Ontology Language Primer, Second Edition. Recommendation, 11 December 2012. Open-world semantics, identity assumptions and the distinction from syntax validation. Accessed 19 October 2026. Official source

[50] W3C. Shapes Constraint Language (SHACL). Recommendation, 20 July 2017. Validation of RDF data graphs against shapes and entailment considerations. Accessed 19 October 2026. Official source

[51] NIST/SEMATECH. Exact Binomial Confidence Limits. Statistical reference, accessed 19 October 2026. Chapter 20 uses the one-sided zero-event bound derived from P(no failures) = (1 − p)^n; the sampling assumptions remain essential. Official source

[52] NIST. Secure Software Development Framework Version 1.1. SP 800-218, February 2022. Secure-development reference for generated code, dependencies and release paths; not an AI accuracy certification. Accessed 19 October 2026. Official source

[53] Cyera. DSPM Platform for Complete Data Visibility. Product description accessed 19 October 2026. Vendor-described discovery, classification, context and remediation; validate selected modules and source coverage. Official source

[54] Cyera. Collibra integration. Published available integration, accessed 19 October 2026. Catalog enrichment, asset matching and optional missing-inventory creation; not a book deployment test. Official source

[55] Cyera. Cyera and Snowflake Govern every agent secure every dataset. Announcement, 2 June 2026; accessed 19 October 2026. Separates GA masking/risk-analysis claims from the then-described agent-discovery private preview. Official source

[56] Cyera. Cyera and Microsoft Purview Demo. Public demonstration description, accessed 19 October 2026. Labeling-gap and protection workflow example; connector coverage and operational fit require verification. Official source

[57] Snowflake. Introduction to sensitive data classification. Documentation accessed 19 October 2026. Native classification, tags and protection integration. Official source

[58] Snowflake. Tag-based masking policies. Documentation accessed 19 October 2026. Enterprise Edition prerequisite, policy signature and conditional protection. Official source

[59] Microsoft Learn. Data sources that connect to Data Map. Capability matrix accessed 19 October 2026. Separates scanning, Data Map labeling and policy support by source, including Snowflake. Official source

[60] Microsoft Learn. Learn about sensitivity labels. Documentation accessed 19 October 2026. Label configuration and supported protection surfaces; not universal control of every copy. Official source

[61] Cursor. Rules. Documentation accessed 19 October 2026. Project instructions and model-context behavior; instructions are distinct from authorization. Official source

[62] Cursor. Model Context Protocol MCP. Documentation accessed 19 October 2026. Connections to external tools; the book proposes its own protected metadata interface. Official source

[63] Cursor. Security and Privacy Hardening. Documentation accessed 19 October 2026. Controls to evaluate for the selected enterprise plan and configuration. Official source

[64] Marc Alvidrez. Embracing Risk. Google Site Reliability Engineering, edited by Kavita Guliani. Resource and opportunity costs of reliability and alignment with business needs. Accessed 19 October 2026. Official source

[65] BCG. How Leaders Build an AI-First Cost Advantage. 2026 practitioner analysis, accessed 19 October 2026. High-impact workflow reinvention and staged delivery; not proof that complexity causes ROI. Official source

[66] BCG. Scaling AI Requires New Processes Not Just New Tools. 2026 practitioner analysis, accessed 19 October 2026. Operating redesign, integration tradeoffs and implementation economics. Official source

[67] Buçinca, Z.; Malaya, M. B.; Gajos, K. Z. To Trust or to Think: Cognitive Forcing Functions Can Reduce Overreliance on AI in AI-assisted Decision-making. Proceedings of the ACM on Human-Computer Interaction 5, CSCW1, Article 188, 2021. DOI 10.1145/3449287. Author publication page; experiment-specific evidence, not a universal review design. Official source

[68] PostgreSQL. Range Types. Current official documentation, accessed 19 October 2026. Inclusive, exclusive and unbounded interval semantics. Official source

[69] Microsoft Learn. Temporal tables overview. SQL Server documentation, accessed 19 October 2026. System-versioned history and transaction-begin UTC period semantics. Official source

[70] PostgreSQL. Transaction Isolation. Current official documentation, accessed 19 October 2026. Isolation levels, concurrent updates and serialization retries. Official source

[71] PostgreSQL. Explicit Locking. Current official documentation, accessed 19 October 2026. Row locks, conflicts, deadlocks and scope. Official source

[72] AWS Prescriptive Guidance. Transactional outbox pattern. Accessed 19 October 2026. Local dual-write consistency, duplicate delivery and consumer idempotency. Official source

[73] Snowflake. Understanding row access policies. Accessed 19 October 2026. Enterprise Edition or higher, policy evaluation and limits. Official source

[74] Snowflake. IS_ROLE_IN_SESSION. Accessed 19 October 2026. Active roles, secondary roles and hierarchy; distinguish CURRENT_ROLE. Official source

[75] Microsoft Learn. Security filters for trimming results in Azure AI Search. Accessed 19 October 2026. Application-supplied security identifiers and filter semantics. Official source

[76] Microsoft Learn. Query-time ACL and RBAC enforcement in Azure AI Search. Accessed 19 October 2026. Preview prerequisites and elevated-read considerations. Official source

[77] Microsoft Learn. Query-time sensitivity label enforcement in Azure AI Search. Accessed 19 October 2026. Preview integration, identity prerequisites and supported rights. Official source

[78] Rau, V. Eliminating Toil. Google Site Reliability Engineering. Accessed 19 October 2026. Repetitive operating work and engineering that reduces it; the stewardship applica-

tion is the author's proposal. Official source

[79] Google. Data Processing Pipelines. The Site Reliability Workbook. Accessed 19 October 2026. Consumer-oriented pipeline freshness, correctness, monitoring and recovery. Official source

[80] W3C. SKOS Simple Knowledge Organization System Reference. Recommendation, 18 August 2009, section 10. Accessed 19 October 2026. Mapping relations and their distinct semantics. Official source

[81] W3C. OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax, Second Edition. Recommendation, 11 December 2012, section 3. Accessed 19 October 2026. Ontology and version identifiers, imports and versioning. Official source

[82] Bitol. Open Data Contract Standard. Official specification, accessed 19 October 2026. Producer-consumer contract structure; implementation and enforcement remain separate responsibilities. Official source

[83] OpenLineage. Object Model. Official specification, accessed 19 October 2026. Datasets, jobs, runs and metadata; observed coverage depends on instrumentation. Official source

[84] DORA. State of AI-assisted Software Development 2025. DORA and Google Cloud, 2025. AI as an amplifier of organizational strengths and weaknesses; returns depend on the underlying delivery system. Official source

[85] METR. We are Changing our Developer Productivity Experiment Design. 24 February 2026. Explains the early-2025 slowdown result and why later productivity estimates are affected by selection and task composition. Official source

[86] NIST National Cybersecurity Center of Excellence. Notional Reference Model for DevSecOps for Demonstration of NIST SSDF. Accessed 20 October 2026. Logical lifecycle phases, feedback, control gates and the current human-directed AI implementation. Official source

[87] Richard White. 25 Years of Lessons Learned as a Practitioner. RJL Solutions, 2026. Forty lessons in eight themes, with sources and practical checks. rjlsol.com/lessons-learned

Reading the legal references

Regulatory references identify selected sources for applicability work. The governing text and the relevant decision holders determine the actual duty. The amending regulation is cited at [46]; current dates were cross-checked with official Commission materials.

Figure provenance

The figures are teaching diagrams, not measured client results. This edition adds a governed stewardship loop and a model for shared and domain-specific ontologies. The DSPM, IDE, context and protection diagrams are proposed designs. The ROI charts use author-created assumptions rather than published industry benchmarks. Each numerical example retains its stated scope and limitations.

[88] Richard White. Choosing an Enterprise Platform for AI Agents. RJL Solutions, Version 2, 29 October 2026. 32 PDF pages. Assistant, model, model-access and agent-platform decisions; market statements checked 29 October 2026 against vendor documentation. rjlsol.com/library/choosing-agent-platform

About the author

Richard White is a data and AI executive whose work connects business strategy, enterprise architecture, engineering and governed operation. He was Global Firm Chief Data Officer at McKinsey & Company and, before that, Chief Data Officer at New York Life, Vice President and Head of Data in the Technology Group at the Federal Reserve Bank of New York, and Global Head of Data Governance in Citigroup's Chief Data Office. His earlier roles include AIG and AXA Equitable, and his career began at Prudential, Siebel and Bank of America.

He leads RJL Solutions, which he founded in 2024. His practice focuses on the ownership, information foundations, controls and operating capabilities needed to put AI into everyday work. He holds a bachelor's degree in Computer Information Systems and Business Management from York College of Pennsylvania and has contributed to industry work on financial-services ontologies and data-management frameworks.

He was named to Constellation Research's inaugural AI150 list of enterprise AI executives for 2024 to 2025.