AGENTIVE WORLD TRILOGY · IV

AgencyDomains

Architecture of the Agentive World

César Obach-Renner

GEGOLABS · FIRST EDITION · 2026

This page intentionally left blank.

AgencyDomains

Architecture of the Agentive World

César Obach-Renner

First edition v1.2.1 · October 2026

This page intentionally left blank.

Contents

AgencyDomains **1**
About this book . . . 1
Structure . . . 1
Who should read this book? . . . 3
How to cite . . . 3
The Trilogy . . . 4
License . . . 5
Contact information . . . 5

Preface **7**
Who should read this book? . . . 7
How is this book organized? . . . 8
A note on terminology . . . 8

Chapter 1 · The Nadella Line **9**
The question that divides the industry . . . 10
What the line separates — Agentic and Agentive in detail . . . 11
Agentive World and agentic world — a deliberate typographic distinction . . . 12
Who bets on each side, and why? . . . 12
Why does the question matter now and not later? . . . 14
How is the line crossed? — the transition is not an event . . . 15
Why the line might not be crossed — and what changes if it is not . . . 15
Visual summary . . . 17

Chapter 2 · The Agentive World **19**
The collapse of the application as interface . . . 19
The new economics of information . . . 21
From the classic cycle to the continuous intelligence cycle . . . 22
Three axes of deep change . . . 24
The nature of the transition . . . 28
The state of the field . . . 30
The architectural obligation . . . 30
Visual summary . . . 31

Chapter 3 · Bounded Concerns Architecture **33**
The Agentic era demands explicit architectural treatment . . . 33
The thinness of the domain as the answer . . . 34

The three layers . . . 36
The seven structural separations . . . 37
Intellectual genealogy . . . 38
Comparative map of proposals . . . 40
Scoring model for the qualitative axis . . . 43
Application case · Activating a new customer's broadband service . . . 44
Architectural implications . . . 46
Limitations of the model . . . 46
Mapping to the Agentive World . . . 47
Closing · The frontier of BCA . . . 50

Chapter 4 · Agentive Architecture **51**
The four layers, seen together . . . 52
The parallel topology . . . 53
The agent's three times . . . 54
Layer 1 — Interaction . . . 56
Layer 2 — Cognition . . . 61
Layer 3 — Autonomy . . . 62
Layer 4 — Access . . . 64
Trust Infrastructure — the cross-cutting axis . . . 65
The governing principle — Agent First . . . 66
The evolution of agents . . . 66
The computational scope — AgencyDomains . . . 68
Reference implementations . . . 69
Evolution frontier . . . 69
Visual summary . . . 70

Chapter 5 · Primitives **71**
AgencyDomains . . . 71
Botlets . . . 86
Capabilities . . . 108
Trust Infrastructure . . . 123
Assistant vs Autonomous Agent . . . 133
Facets . . . 138
Agentlets . . . 144

Chapter 6 · Market **153**
The AI value chain . . . 153
Deep-dive · Observability (link 8) . . . 166
Carbon World · link 11 . . . 171

Chapter 7 · Real-time knowledge **177**
The historical problem . . . 178
The agentive solution . . . 178
The underlying architecture — Varnished Kimball . . . 180
Mapping to the hyperscalers . . . 182
The industry consensus . . . 183
How is the canonical case implemented? . . . 183

Why does this case qualify as a canonical application? . . . 184

Chapter 8 · Trust Infrastructure operationalized — policies and CRUDLEX 185
Policy catalog . . . 185
Complete CRUDLEX model . . . 189
Append-only log format . . . 191
Human approval protocol . . . 194
Hallucination-detection rules . . . 195
Tokenization policy . . . 196
Minimum viable catalog . . . 197
The tripartite deployment pattern — Cloud + Client + Local . . . 197
The economics of operationalized Trust Infrastructure . . . 199
Operational continuity — operationalizing the second mechanism . . . 199
What comes next . . . 202

Chapter 9 · Vergis — the reference implementation 203
A note on the scope of the canon . . . 203
What is it and why does it exist? . . . 204
Where does it live? . . . 204
How are the pieces named? — Vergis · Botler · Mira · Daftar . . . 204
What does it include? . . . 205
Why is it production grade? . . . 206
How is it adopted? — the model . . . 207
How does the catalog grow? — common catalog and network effects . . . 207
How does one contribute? . . . 207
Do the reference and the proprietary codices coexist? . . . 208
Conformance . . . 208
Evolution frontier . . . 209

Epilogue · The Evolution Frontier 211
What the book established . . . 211
The four live frontiers . . . 212
Crystallization — why the advanced agent generates less . . . 214
What the technical community must build . . . 216
What is NOT in this book — and why? . . . 217
Why does this book aspire to be a standard? . . . 217
How does this book evolve? . . . 218
Closing . . . 219

Appendix A · Canonical glossary 221
A . . . 221
B . . . 224
C . . . 227
D . . . 231
E . . . 232
F . . . 233
G . . . 234
H . . . 235

I 235
J 236
K 236
L 236
M 237
O 238
P 239
R 241
S 242
T 243
U 245
V 245
W 246

Appendix B · References **247**
Market analysis and consulting 247
Regulatory and governance frameworks 247
AI platforms — Models and agents 248
Conversational BI and data architecture 248
AI observability 249
Vertical specialists 249
Tool platforms and vector databases 249
AI firewall and security 249
Enterprise integrations 250
Industrial and Carbon World 250
Press, blogs, and field articles 250
Academic research and reference specifications 250
Market studies and projections 251

Colophon · On how this book was written **253**

AgencyDomains

Architecture of the Agentive World

Author: César Obach-Renner **Publisher:** GegoLabs **Edition:** First edition · October 2026 · v1.2.1 **License:** GNU Free Documentation License v1.3

> **A note on the version.** This is the **first edition (v1.0; this revision is v1.2.1)**: the first stable public release. From here the *append-only* rule governs figure numbering — every new figure takes the next available number, with no renumbering of existing ones. Comments and errata are welcome in the repository. The version history lives in `CHANGELOG.md`.

About this book

The AI industry builds agents with the same attitude it built web applications with in 2005: each vendor with its own stack, no common contracts, no clear separation of concerns. Gartner projects that more than **forty percent of agentic AI projects will be cancelled before the end of 2027** because of costs, unclear business value, or inadequate risk controls. **Only twenty-one percent of organizations have mature governance over their agents.** The root cause is structural: the category lacks a shared formal architecture — and has no concrete name for the minimal unit of deployment.

This book proposes that name and that architecture. The minimal unit of deployment is the **AgencyDomain** — the computational scope where agents dwell, Botlets execute, and Trust Infrastructure is exercised. Around the AgencyDomain, the book defines a paradigm (**the Agentive World** — *post-applications*), a formal architecture (**four layers with distinct concerns**, organized in a parallel topology), a set of canonical technical primitives (**Botlets**, **Capabilities**, **Trust Infrastructure**), and a two-dimensional model for placing any actor in the AI market (**the AI value chain** — eleven links × four depths).

The book is **product-agnostic**. The formal constructs described here admit multiple implementations. The **public reference implementation** —Vergis— is developed in Chapter 9.

Structure

Chapter	Content
Preface	Origin of the book · who should read it · how it is organized · on terminology
1 · The Nadella Line	The canonical question · Agentic vs Agentive · the CEOs · why it matters strategically

Chapter	Content
2 · The Agentive World	The consequences of the crossing · organizational transformation · the new economics of information · field data
3 · Bounded Concerns Architecture	The pre-agentive state · three layers and seven separations · the seventh Procedural/Agentic separation · mapping to the Agentive World
4 · Agentive Architecture	The four layers in parallel topology · **the agent's three times** (Preparation · Attention · Engineering) · Layer 2 ↔ Layer 3 interface via `MCP` · cross-cutting Trust Infrastructure · the Agent First principle · GUI generated on-the-fly · **composition of Layer 1** (shell · view · operation · **multi-view PI and drill-through**) · **Facet vs Botlet**
5 · Primitives	AgencyDomains (with distributed Layer 3 + portability) · Botlets (with maturity + seed/emergent + **proto-Botlet · manifestation · temporality** + generic Botler) · Capabilities (**strict to Layer 2** + locality + regulatory certification + **portability · feature · Connector · Template**) · Trust Infrastructure (with operational continuity + **declarative quality contract**) · Assistant vs Autonomous Agent · **Facets (sixth primitive — Layer 1) · declared bounded interaction** · **Agentlets (eighth primitive — packaged routine judgment)**
6 · Market	The AI value chain · depths and archetypes · per-link deep-dives · the Carbon World
7 · Canonical applications	Real-time knowledge · Varnished Kimball · conversational BI
8 · Operation	Trust Infrastructure operationalized · CRUDLEX · policies · operational business continuity
9 · Vergis	The reference implementation · note on the scope of the canon · the Vergis · Botler · Mira · Daftar scheme · what it includes · production grade · adoption model · common catalog and network effects
Epilogue	Evolution frontier · four frontiers (non-LLM cognition · federation · the Carbon World · the institutional horizon) · essay on the Botlet generations
Appendices	Glossary · references · colophon

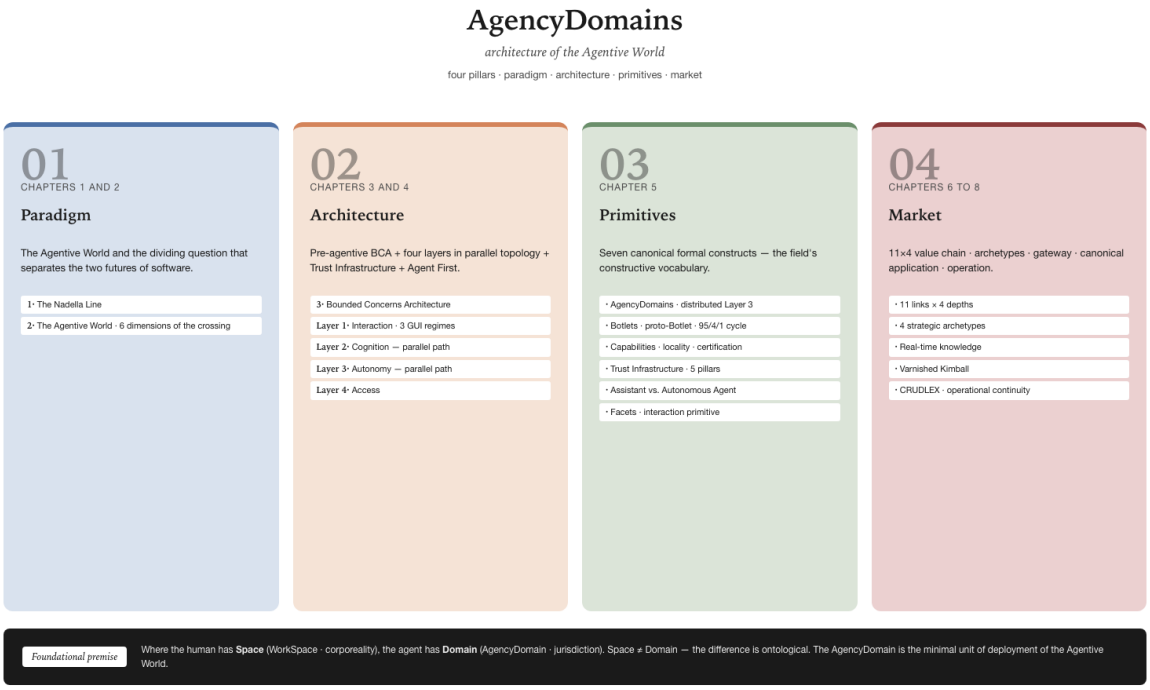


Figure 1: Index plate · the four pillars of the proposal

Who should read this book?

This book is addressed to:

- **Systems architects** who design or evaluate agentive platforms and need a common frame for reasoning about separation of concerns, governance, and resilience.
- **CTOs and technical leaders** who must make agentive-stack decisions on a three-to-five-year horizon.
- **Strategists and consultants** who advise companies on their transit into the Agentive World.
- **Researchers and academics** interested in the formalization of agentive architecture as a category of study.
- **Product builders** who want to situate their contribution in a value chain broader than their particular product.

The book is introductory in the sense that it assumes no prior knowledge of specific implementations, but it does assume basic familiarity with distributed systems, language models, and operating systems in production.

How to cite

> Obach-Renner, César. *AgencyDomains: Architecture of the Agentive World*. First edition v1.2.1. GegoLabs, 2026.

The Trilogy

This book is volume **IV** of the **Agentive World Trilogy**, in five parts (so far). The transition that occupies the trilogy is a single one: from the world where people open applications to work, to the world where AI agents are the interface of work. The four volumes answer, in order, the four questions of that transition — and each one reads complete on its own:

I · The Death of Software as We Know It Why now? — **the starting point.** For those who decide how to automate their business: why bought software stops being the answer, what is lost when every organization builds its own system, and which piece survives the change — the system of truth.

II · The Real-Time Enterprise — The Agentive World Where are we going? — **the destination.** For those who want to see the world on the other side of the transition: how people work, decide, and compete when the enterprise operates in real time — told in its four faces.

III · AURA — The Agentive Path Which way? — **the route.** For those who have to make the crossing: leaders and advisors of the transformation. Its five specifications are standalone instruments for the agentive era — each one can be used separately, without the rest of the book:

Spec	What is it?	Used alone, what does it give you?
IRIS	A maturity model of organizational intelligence	Locates your organization on the road from data to action
MOTOR	A maturity model of organizational automation	Measures how advanced your organization is at process automation
Data Canon	The evolution of the decentralized data-management model (Data Mesh)	Defines how the organization's data is governed
Wingmap	The evolution of business process management (BPM / process mining)	Reconstructs processes and information flows without interviews
Use Cases	A portfolio framework, illustrated with 100 use cases and 30 value solutions	Prioritizes where to start, according to your maturity

IV · AgencyDomains — Agentive Architecture *(this book)* How do we get there? — **the vehicle.** For architects, CTOs, and builders of agentive platforms: the formal specification of the Agentive World.

A · Vergis — Agentive Technology *(the reference implementation: not a book, it's code)* What do I start with today? — **the keys.** For anyone who wants to start making all of this real: the reference implementation of the architecture, open source, at https://github.com/gegolabs/vergis.

The four books were written in the reverse of their reading order, like a stack: *AgencyDomains* was the first and ended up at the bottom; *The Death of Software as We Know It*, the last, ended up on top. That is why a book may cite another that is read later: it cites it because it was already written.

Start with **The Death of Software as We Know It** if you need to understand why the change is happening now; with **The Real-Time Enterprise** if you need to convince yourself — or someone else — that the transition is for real; with **AURA** if you have to lead it; with **AgencyDomains** if you have to build it; with **Vergis** if you want to see it running. The full trilogy lives at https://agencydomains.org.

License

This book is published under the **GNU Free Documentation License v1.3**. The reader may copy, distribute, and modify the work under the terms of the license. The invariant section is the Preface.

The reason for the open licensing is deliberate: the Agentive Architecture aims to establish a category that serves the whole ecosystem, not a particular vendor. By adopting and developing it, other companies, researchers, and builders enrich it and consolidate it as a de facto standard. This serves the field and simultaneously reinforces the authority of those who coined it.

Contact information

- **Website:** https://agencydomains.org
- **Repository:** https://github.com/gegolabs/agencydomains.org
- **Errata and comments:** https://github.com/gegolabs/agencydomains.org/issues

This page intentionally left blank.

Preface

By César Obach-Renner · July 2026

After eighteen months watching the industry build AI agents — some as a demo's whim, others as enterprise pilots that died on the way to production — I synthesized a formal architecture for the problem. I call it the **Agentive Architecture**. This book is its specification.

I came to the problem by the least elegant route: I needed it. In 2025 I began building, inside my company, a portfolio of products in which humans and agents had to coexist as first-class digital citizens. I soon discovered that the category did not exist with the maturity required to sustain that design. There was a great deal of loose language — *"agentic AI"*, *"AI copilots"*, *"AI agents"*, *"autonomous workflows"* —, much marketing and little architecture. The industry named the horizon; but there was no common specification of the technical substrate, nor a shared vocabulary to discuss it.

Faced with that, I had two paths: improvise over the scattered pieces, or build the frame my company needed. I chose the second. I built it first as internal notes; then as the project's canonical documents; then as the backbone of the entire portfolio. At some point it became evident that the result had value beyond my company — that any organization building toward this horizon faces the same problem and needs the same frame.

This book is the formalization of that frame. It defines the paradigm (**the Agentive World**), a formal four-layer architecture, the technical primitives that inhabit it (**Botlets**, **Capabilities**, **Agency-Domains**), a cross-cutting trust infrastructure (**Trust Infrastructure**), and a two-dimensional market model (**the AI value chain**). The purpose is not to convince the reader to adopt my particular implementation. The purpose is to offer the whole ecosystem a common language for reasoning about productive agentive systems.

I am clear about where I come from. Almost twenty years ago I wrote a similar book for another category that was then beginning to take shape — *SOAr: El concepto* (2008), a formalization of Service-Oriented Architecture for enterprise integration. That work, too, was born of a real project where the methodology did not exist, so I created it and documented it. What I learned then, and apply here, is that technical categories are established not when someone invents them, but when someone **writes them down with enough discipline that others adopt them**. This book is that attempt.

Who should read this book?

This book is addressed to those who are building, evaluating, or governing productive agentive systems.

If you are a **systems architect** or a **CTO**, you will find a common frame for reasoning about separation

of concerns, governance, technical primitives, and stack positioning on a three-to-five-year horizon. If you are a **product strategist** or a **consultant**, you will find a two-dimensional market map for situating any actor — your own or another's — in the AI value chain. If you are a **researcher** or an **academic**, you will find a reasonably rigorous formalization of a category still in consolidation.

The book is introductory in the sense that it assumes no prior knowledge of specific implementations. But it assumes basic familiarity with distributed systems, language models, and operating systems in production. It is not an implementation manual — it is the specification a manual ought to respect.

How is this book organized?

The book advances from the paradigm to the implementation, in a sequence where each part rests on the previous one:

- **The paradigm** — The Nadella Line (the question) · The Agentive World (the consequences).
- **The Agentive Architecture** — four distinct layers · cross-cutting Trust Infrastructure · Agent First (the governing principle).
- **The primitives (seven)** — AgencyDomain · Botlet · proto-Botlet (+ manifestation + temporality) · Capability (cognitive, Layer 2) · Trust Infrastructure · Assistant vs Autonomous Agent (axis primitive) · Facet (Layer 1).
- **The market positioning** — The AI value chain (11×4) · Observability · the Carbon World.
- **The applications** — Real-time knowledge (the canonical case: Varnished Kimball · conversational BI).
- **The operation** — Trust Infrastructure operationalized (policies · CRUDLEX · log).
- **The reference implementation** — Vergis (Chapter 9).
- **The epilogue** — the evolution frontier (non-LLM cognition · federation · the Carbon World · AgentNation).

The path chains the chapters like this: **The Nadella Line** (the question) → **The Agentive World** (the consequences) → **the Agentive Architecture** (the four layers) → **the primitives** that populate them → **the market positioning** → **the canonical application** → **the operation** → **Vergis**, the reference implementation → **the epilogue**. The glossary at the end fixes the canonical vocabulary.

A note on terminology

Much of this category's technical vocabulary is native to English and lacks Spanish translations that are at once precise and recognized: *agent*, *agentic*, *agentive*, *runtime*, *guardrails*, *tool*, *prompt injection*, among others. This book was first written in Spanish, where it adopted the English originals whenever a translation would be forced or the term already circulated with authority in the field. In this English edition that tension largely dissolves; what remains is the discipline of a fixed vocabulary.

The canonical concepts coined in this book — **Agentive Architecture**, **Nadella Line**, **AgencyDomains**, **Botlet**, **Capability**, **Trust Infrastructure**, **Agentive World** — are kept in their original form throughout the work, capitalized, as proper names. Their precise definitions live in the glossary at the end of the book.

As the category matures across languages and communities, consensus terminology may emerge. If it does, future editions will gather it. For now, the priority is not to break the traceability of a concept across the communities that use it.

Chapter 1 · The Nadella Line

This whole book turns on a single question — and on the architectural answer that question demands. The question was dropped, almost in passing, by Microsoft's CEO; this chapter turns it into the frontier that today splits the industry in two: the **Nadella Line**. The answer — the formal construct that gives the book its title — will come in due time, once the reader has the full paradigm in hand.

In December 2024, Satya Nadella sat across from Brad Gerstner and Bill Gurley on the BG2 podcast and let drop a line his interviewers did not expect. The conversation had been about the future of productivity, the role of copilots, how Microsoft was thinking about the next decade of enterprise software. Nadella, with no prophetic tone, almost as if stating the obvious, offered the line that would give this book its name:

> *The notion that business applications exist — that's probably where it all collapses, in the era of agents.*

The statement went almost unnoticed. Dozens of podcast clips were published, and far more was discussed about OpenAI and reasoning models, about the economics of compute, about NVIDIA's role. But that line contained a prediction which, if it proves correct, changes the entire order of modern enterprise software. The prediction was not about AI — it was about **applications**: specifically, about the claim that they will cease to exist as a category.

Nadella was not talking about evolution. He did not say "applications will get smarter." He did not say "applications will have embedded copilots." He said *collapses*. And he said *where it all collapses* — singular, defining. If Microsoft, owner of the most widespread office suite on the planet, predicts that business applications cease to exist, one has to understand why.

The full quote deserves a careful read, because it contains more than the forecast — it contains the **mechanics** of the collapse:

> *"…they are essentially CRUD databases with a bunch of business logic. The business logic is all going to these agents, and these agents are going to be multi-repo CRUD: they're not going to discriminate between what the backend is. They're going to update multiple databases, and all the logic will be in the AI tier, so to speak. And once the AI tier becomes the place where all the logic is, then people will start replacing the backends."*

Read it slowly: Nadella describes **two phases**. First the interface collapses — the logic migrates to the agent tier, which operates over the repositories without discriminating the backend. Then, with the logic already in the AI tier, the backends themselves start being replaced. This book formalizes exactly that mechanics: the agent tier that concentrates the logic is the architecture of Chapter 4; the governed "multi-repo CRUD" is what Chapter 8 operationalizes as CRUDLEX; and the backends that persist invisible, consumed via Connectors, are Layer 4. The short quote gives the headline; the full

quote gives the blueprint.

This chapter is the answer. What we will call **the Nadella Line** is the conceptual boundary between two irreconcilable futures of software — one where applications persist, another where they collapse. The entire AI industry, consciously or not, is betting today on one side or the other. Every stack decision an organization makes today is, implicitly, an answer to a single question.

The question that divides the industry

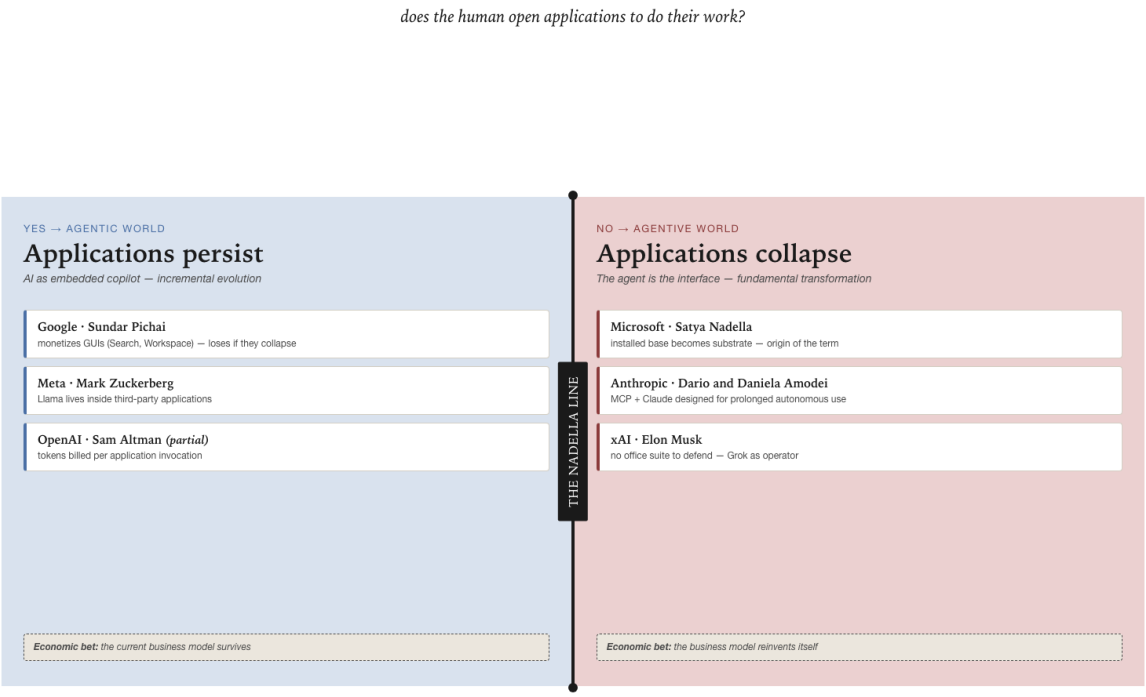


Figure 2: The Nadella Line · the dividing question that splits the industry

The question is deceptively simple:

> *Does the human open applications to do their work?*

If the answer is **yes**, the organization lives on what we call the **Agentic** side of the line. The traditional interface — Excel, Salesforce, Power BI, Outlook, ServiceNow, Confluence — persists. AI arrives as an **embedded copilot** in each of those applications: the "Suggest formula" button in Excel, "Compose email" in Outlook, "Ask your data" in Power BI. The application is the substrate; AI is sophisticated ornament on top.

If the answer is **no**, the organization has crossed to the **Agentive** side of the line. Applications, **as interface**, have disappeared. The human does not open Salesforce to ask about their pipeline — they ask an agent. The human does not open Power BI to review quarterly margins — they ask an agent. Applications may go on existing *underneath*, as invisible backend infrastructure the agent queries, but the human no longer sees them. Their primary interface with the digital world is **conversation with an agent that has access to everything**.

The difference between the two sides is not one of degree — it is **categorical**. A very advanced Agentic World, with perfect copilots in every application, remains Agentic as long as the human keeps the habit of opening applications. A nascent Agentive World, where conversation with an agent is still clumsy, has already crossed the line if the human has stopped opening applications to do their work. The boundary is set by the human's behavior, not by the technical sophistication of the system.

The question matters because it admits no honest middle ground. A human who opens twenty applications a day and a human who converses with an agent all day are two distinct operating models of professional work, and the technical architectures that support them are **incompatible**. An organization betting that applications persist builds one thing; an organization betting that they collapse builds another. You cannot build both at once with coherence.

What the line separates — Agentic and Agentive in detail

The **Agentic** paradigm describes the horizon where AI agents are complementary tools within a universe of applications that survives. The traditional interface persists. The employee remains the operator of the software — opens the applications, navigates their menus, presses their buttons — only now each application has **copilots** that accelerate what the human did by hand. Microsoft 365 Copilot inside Word, Gemini for Workspace inside Google Docs, Salesforce Einstein inside Salesforce, GitHub Copilot inside the IDE.

The agentic paradigm is **incremental evolution**. The way of working does not change: the speed does. The financial analyst still builds their model in Excel — only now they ask the copilot to suggest the right formula instead of looking it up in the documentation. The consultant still drafts their proposal in Word — only now the copilot helps them structure the first draft. The executive still reviews dashboards in Power BI — only now they can ask the copilot to explain an anomaly. The required skill is a natural extension of the current skill: the employee who knew how to operate Excel now learns to *operate Excel with a copilot*. Same application, same mental structure of the work, better speed.

The **Agentive** paradigm describes a qualitatively different horizon. In the Agentive World, agents **are** the interface. Applications, as a category perceptible to the human, **collapse**. The financial analyst stops opening Excel: they ask an agent directly why margins fell in Q3. The agent, underneath, queries systems that may or may not include Excel databases — but the human never sees the cell. The consultant stops opening Word: they dictate the structure of the proposal to an agent and review the result. The executive stops opening Power BI: the agent proactively sends them the anomalies it detected, in conversational form, with no dashboards in between.

The agentive paradigm is **fundamental transformation**. The way of working changes. The required skill is redefined: it is no longer about knowing how to operate applications but about **directing agents** — framing requests well, validating responses, governing what agents decide and execute. The analyst who in the Agentic World prided themselves on knowing Excel in depth, in the Agentive World prides themselves on knowing how to **pose analytical questions** the agent can answer. The specific application — Excel, Power BI, Salesforce — becomes an implementation detail the human never touches.

The Agentive World **does not imply that applications disappear entirely**. It implies that they stop being the human's **interface**. Salesforce, as a system that stores customer information, can keep operating perfectly well in the Agentive World — only the agent queries it via API, not the human via UI. The application becomes **invisible backend infrastructure**. It survives as a repository of data and business logic, but loses its role as work surface. This distinguishes the agentive scenario from an

apocalyptic one where "all software dies" — the software survives; what dies is the **GUI** as the primary interface of cognitive work.

Agentive World and agentic world — a deliberate typographic distinction

The English-speaking industry took *agentic AI* as a general term for any AI that acts with some degree of initiative. The term circulated fast and filled with heterogeneous content: embedded copilots, virtual assistants, autonomous systems. Today *agentic AI* means whatever a vendor wants to sell under that umbrella. The distinction Nadella made on BG2 — the distinction between the two sides of the line — was buried under the noise of marketing.

This book preserves the distinction that the consolidated industry lost. The distinction is qualitative, not one of degree. **Agentic** describes the evolutionary mode: agents that complement applications. **Agentive** describes the transformational mode: agents that replace applications as interface. Agentic says: *the tools remain, but now they are smarter*. Agentive says: *the tools we know disappear as interface; what remains is conversation with agents*.

In Spanish, where *agentivo* and *agéntico* differ by a single accent, the distinction between the two categories collapses into a change no reader retains — which is why the original edition adopts a deliberate typographic convention. In English the two words are already distinct, but the book keeps the same capitalization rule because it carries a second load: when the terms refer to the **paradigm** as a named noun — the vision, the category, the side of the line — they are capitalized: the **Agentive World**, the **Agentic World**. When the same terms are used as a **descriptive adjective** — qualifying a system, an organization, an era — they are lowercase: *an agentive system*, *an agentic era*, *agentive products*.

The uppercase / lowercase distinction is what carries the categorical difference. When the reader encounters *"the Agentive World is the destination,"* they read the noun of the paradigm — the referenceable entity the book defends. When they encounter *"building a serious agentive system,"* they read the descriptive adjective — a quality of a particular object. It is the same convention serious technical treatises use: *Bounded Context* in Eric Evans, *Aggregate* in Domain-Driven Design, *Replication* in Kleppmann. The canonical concept is capitalized; the descriptive use is lowercase.

The cost of this convention is a microsecond of the reader's attention when they see the two forms. The benefit is preserving the category. When someone reads *"the Agentive World"* and it reads differently from a plain *"agentive world,"* they are grasping exactly the difference the book sets out to defend.

Who bets on each side, and why?

As of early 2026, the industry's principal players have taken identifiable positions on the Nadella Line question. The divide is not ideological — it is one of **business model**. Each player predicts the world that protects its competitive position, and although the predictions are published as technical vision, they are ultimately predictions about where their company's cash flow will survive.

Microsoft (Satya Nadella) bets openly on the Agentive World. The bet is consistent with a strategic repositioning Microsoft has been pursuing for years. The company that built its fortune on Office is reinventing itself as an agent platform: Copilot Studio to build enterprise agents, AutoGen as a multi-agent orchestration framework, Microsoft 365 Copilot integrating specialized agents that replace, not extend, the traditional interfaces. Microsoft can afford to predict the collapse of applications because its

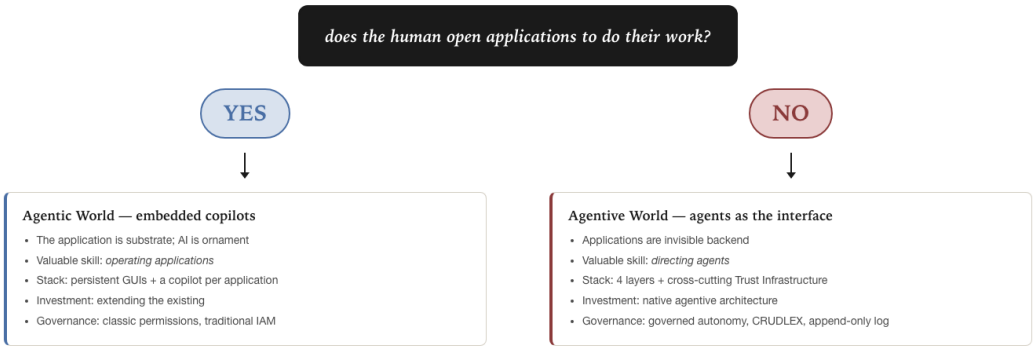


Figure 3: The dividing question · the flow of architectural consequences

installed base becomes the substrate — the data that lives in SharePoint, Outlook, Teams, OneDrive is already on its infrastructure. If GUIs collapse, the data is still theirs. For Microsoft, the Agentive side is where its cash flow survives.

xAI (Elon Musk) also bets on the Agentive World, though from a different position. Musk has no office suite to defend; what he has is Grok integrated into X and the planned integration with Tesla vehicles. His bet is to build an agent that operates autonomously over extended tasks — not as an application assistant, but as a process operator. For xAI, the Agentive World is where its product is relevant: in an Agentic World with copilots embedded in existing applications, Grok competes with consolidated products (Office Copilot, Google Gemini); in an Agentive World, it competes for a new category with no established incumbents. And it is worth recording that Musk's position is more radical than this book's: in his formulation, not only does the interface collapse — applications do not survive as backend either (*"apps will disappear, like Blockbuster's VHS tapes"*; phones become "edge nodes for AI inference"). The real spectrum has three positions — Agentic, Agentive (the interface collapses; the backend persists, invisible and governed), and extreme Agentive (no backends either) — and this book's thesis is deliberately the middle one: the one that can be sustained with architecture.

OpenAI (Sam Altman) occupies a more ambiguous position. Publicly, Altman has favored the agentic narrative — "GPT as a complementary tool for existing applications" — consistent with OpenAI's API-centric business model: the more GPT is invoked from existing applications, the more tokens are billed. But internally, OpenAI has developed growing agentive capabilities (GPTs as in-application agents, Operator as an execution agent, a declared "AGI" vision). The ambiguity reflects the operational reality: OpenAI makes money today in the Agentic World, but its valuation assumes it will make exponentially more in the Agentive World.

Google (Sundar Pichai) bets on a dominant Agentic World. The reason is structural: Google's business model rests on search and on advertising tied to interfaces — a Google Search page with ads, a Google Workspace dashboard with a monthly subscription. If GUIs collapse, Google loses the surface where it monetizes. Pichai speaks of Gemini as a "layer that supercharges" Search, Workspace, and Android — a deliberate verb: *supercharges*, not *replaces*. The prediction is consistent with his model: Google's applications survive, enhanced with AI, but do not collapse.

NVIDIA (Jensen Huang) occupies a technical position with no need to take sides. NVIDIA sells compute. Either world consumes GPUs massively: the Agentic World requires copilots that invoke models on every user action; the Agentive World requires agents that operate continuously in the background. Huang speaks of agents as "a new class of workloads" — strategic neutrality, because NVIDIA wins in both scenarios.

Anthropic (Dario and Daniela Amodei) bets predominantly on the Agentive World, with an emphasis on prolonged autonomy and tools. The introduction of the Model Context Protocol (MCP) in November 2024 is a clear signal: MCP is an open standard for agents to connect to external tools — exactly the architectural piece the agentive category needs. Claude is designed, from its training, for prolonged autonomous use rather than short turn-by-turn responses. Anthropic's agentive bet is consistent with its thesis on AGI: the path to the most capable model runs through models that act, not models that answer.

Meta (Mark Zuckerberg) bets on the Agentic World. Llama, its foundation model, is distributed open-source to be used inside third-party applications. The agents in WhatsApp and Instagram are features within applications that exist. Meta benefits from a world where social platforms survive and agents are ornament within them.

The resulting divide is revealing. The players who predict the Agentive World are those whose business model survives — or improves — in that scenario. The players who predict the Agentic World are those whose business model depends on applications persisting. Each one's technical prediction is, beneath the neutral language, a prediction of economic survival.

Why does the question matter now and not later?

The reasonable objection of a prudent executive reading this chapter is: *even if the Nadella Line is real, its crossing seems distant. Why make decisions today based on a transition that will take a decade?* Three operational reasons make the question **inevitable** within the current decision horizon, not deferrable.

The first reason is the speed of the transition. The field data no longer allows treating the Agentive World as a distant horizon: the global market for agentic AI multiplies its size twentyfold in a decade, at a compound rate above forty percent a year. The transition is not linear: it accelerates. **Chapter 2 develops** the figures and the sources that underpin this speed. Whoever plans for the 2027–2028 horizon is planning for a world where a material fraction of operational decisions are made by agents.

The second reason is the horizon of the stack decisions made now. A company that adopts an enterprise suite today — Microsoft 365, Google Workspace, Salesforce Sales Cloud, ServiceNow — is committing capital and implementation time that typically amortizes over three to five years. That same window is exactly the frontier where the Nadella Line becomes verifiable or falsifiable. The question is not postponed: every stack decision the organization makes today is implicitly an answer. Renewing Office 365 with a five-year horizon is implicitly betting that the Agentic World will last five years. Migrating to a native agentive architecture is implicitly betting that the crossing will occur within that

horizon. Not making the decision is also a decision: it defaults to inertia, which almost always coincides with betting on the agentic side out of mere institutional habit.

The third reason is the asymmetry of the cost of being wrong. An organization that bets on the agentic side and invests in application-dependent architectures — workflows tied to UIs, integrations by screen scraping, business logic embedded in how the system looks — builds **mounting architectural debt** if the line is crossed. Migrating to the Agentive World from an agentic base is not extending what was built: it is dismantling it. Conversely, an organization that bets on the Agentive side early and builds with the right architecture can sustain its system under both paradigms throughout the transition period, with no rewrite. The four layers this book develops in Chapter 4 — Interaction, Cognition, Autonomy, Access — are valid whether Layer 1 remains a traditional GUI or is a conversation with an agent. The asymmetry is structural: building Agentive from the start serves both worlds. Building pure agentic and migrating later does not.

This asymmetry is what makes the question undeferrable. A company can be wrong by betting on the Agentive World too early — losing a few years of efficiency while the transition has not finished happening — but it is not left without options. A company that bets on the Agentic World assuming the crossing will not happen, and then discovers that it did, faces a costly structural migration or, worse, is left with unviable systems operating in a market that evolved. The error in one direction is an inconvenience; the error in the other is architectural debt that is hard to reverse.

How is the line crossed? — the transition is not an event

There is a risk of reading the Nadella Line as if it were a single point-in-time event: the day applications disappear, the moment of transition. That reading is mistaken. **The crossing of the line is a progressive transition that accumulates** layer by layer, function by function, team by team. An organization does not become agentive on a Monday. It becomes **progressively** agentive: first a process, then a team, then a function, then most of its day-to-day operation.

The transition has three characteristic dynamics — **evolutionary coexistence** (the current paradigm and the emerging one coexist for years), **asymmetry across functions** (not all cross at the same pace), and **temporal asymmetry** (the pace of the crossing is not uniform in time either) — and one underlying consequence: the **reorganization of human work**, which shifts from executing tasks to governing the agents that execute them. **Chapter 2 develops it** in detail, with the field data that underpins it.

The three dynamics taken together explain why the transition is not traumatic for the organization that plans it. An organization that recognizes evolutionary coexistence is not forced into a big bang; one that accepts asymmetry across functions does not force uniform paces; one that redesigns human roles toward governance does not discard its human capital but relocates it. The transition is progressive, asymmetric, and reorganizing — and that is what makes it operable. The crossing also admits an operational indicator — the **agentive percentage**: what fraction of their tasks an employee can delegate entirely without opening an application. *AURA*, volume II, develops it as a diagnostic instrument alongside its maturity models.

Why the line might not be crossed — and what changes if it is not

A thesis that does not face its strongest objection is not validated — it is untested. This chapter has argued that the question is unavoidable and that the cost of being wrong is asymmetric; it is now time

to let the other side speak. Five serious arguments hold that the line might not be crossed — or crossed only partially.

The control argument. Humans have a control bias: we prefer to see and validate before executing, even when delegating would be more efficient. A CFO may not want to ask "are our finances all right?" and trust the answer — they will want to **see** the numbers, the formulas, the sources. The graphical interface provides a feeling of control that conversation does not replicate, and might keep the high-risk, high-accountability use cases for that reason.

The irreducible-complexity argument. Enterprise systems are not just "CRUD databases with business logic": they accumulate decades of rules no agent can infer without documentation that frequently does not exist, integrations with behaviors that emerged from years of use, and edge cases codified after real incidents. An agent that orchestrates them without understanding them produces catastrophic errors; training it system by system replicates the complexity it promises to eliminate.

The regulatory argument. Entire sectors — finance, health, government — operate under frameworks that demand deterministic traceability and assignable accountability, and agents are fundamentally probabilistic: identical inputs can produce different outputs. Until that changes, there are industries that will remain on the agentic side by legal mandate, not by technical lag.

The economic argument. The transition demands retraining entire workforces, rewriting or abandoning trillions in existing software, and building a governance infrastructure that largely does not yet exist. A rational company may stay on the agentic side simply because crossing costs more than it yields — for years.

The historical argument. Every technology generation produced the same prophecy of total extinction — "the PC will kill the mainframe," "the web will kill installed software," "the cloud will kill on-premise" — and every time the reality was stratified coexistence, on timelines three to five times longer than predicted.

These arguments do not topple the Nadella Line. They **refine** it, and the refinement is worth incorporating into the thesis rather than dodging: the line may be **asymptotic** — indefinite approach with no complete crossing —; each industry has **its own line**, and some may never cross; the crossing can **reverse** after incidents; and **coexistence is the stable state** — agentic for high risk, agentive for the rest, with the proportion moving in a single direction.

And there is a second reason to face the objections rather than dodge them: **this book exists, in good part, because they are valid.** The control argument is answered by Trust Infrastructure's Transparency and auditable record (Chapter 5 §4): the CFO who wants to see the numbers sees them — with lineage. The regulatory one is answered by certification residing in the certified component rather than in generated code (Chapter 5 §3), and by Chapter 8's append-only log. The complexity one is answered by a Layer 3 that consolidates operational knowledge into mature Botlets instead of improvising every time (Chapter 5 §2). And the economic one is exactly the asymmetry we already argued: building agentive from the start serves both worlds. The objections are not enemies of the architecture — they are its list of requirements.

The question is not rhetorical. It is the question on which the decisions of stack, talent, architecture, and investment of every serious organization depend, over the horizon of the next five years. It is the question on which the rest of the book rests: understanding where the crossing is headed allows one to operate with confidence during the transition period; ignoring it exposes the organization to decisions

that age badly. The question is: *which side of the Nadella Line is your organization building on?*

Visual summary

For support in later reading, the two sides of the line synthesized in a comparison table.

	Agentic World (pre-line)	**Agentive World** (post-line)
State of applications	Persist as primary interface	Collapse as interface; survive as backend
Operational question	The human opens applications to work	The human converses with agents that have access to systems
Role of AI	Copilots embedded in each application	Agents that replace the traditional interface
Valuable human skill	Knowing how to operate applications	Knowing how to direct agents
Transformation	Incremental evolution, same way of working	Fundamental transformation, new way of working
Predicted by	Altman (OpenAI)*, Pichai (Google), Zuckerberg (Meta)	Nadella (Microsoft), Musk (xAI), Amodei (Anthropic)
Economic bet	The current business model survives	The business model reinvents itself

* a nuanced position; see the body of the chapter.

What follows — the Agentive World (Chapter 2), the pre-agentive cartography (Chapter 3), the formal four-layer architecture (Chapter 4), and the primitives (Chapter 5) — gains coherence under a single central construct that the book proposes as the minimal unit of deployment: the **AgencyDomain**. The book names it explicitly in the title not by rhetorical choice, but because the entire spec turns around it. The Nadella Line is the question of the paradigm; the AgencyDomain is the architectural answer.

This page intentionally left blank.

Chapter 2 · The Agentive World

Chapter 1 established the dividing question: if the human opens applications to do their work, the organization lives on the agentic side; if not, it has crossed to the Agentive side. This chapter assumes the answer is **no** — that the organization has crossed the Nadella Line — and develops, with the calm and detail the question deserves, what that crossing means in practice.

The question is not trivial. Any executive, architect, or consultant who has faced a large technological transition knows that the consequences of the crossing are never the deck of slides with optimistic arrows that the first evangelist promises. The real consequences are rugged, asymmetric, partly predictable and partly surprising. But there is a core of changes that can indeed be anticipated with discipline, and it is those changes that this chapter describes.

Crossing the Nadella Line simultaneously changes six dimensions of any organization's operation: the way the human interacts with the system, the nature of the data the system consumes, the roles of human work, the economics of information, governance, and the operating model. None of the six changes in isolation. Advancing on one without the others produces **successful pilots but no real transformation** — an observation the consulting firms that have documented the field repeat with uncomfortable frequency. The crossing is systemic or it is not.

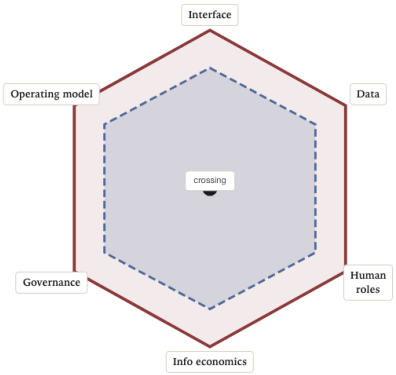
We will begin with the most visible consequence: the collapse of the application as the primary interface of cognitive work.

The collapse of the application as interface

For forty years enterprise software was built around an implicit postulate that was rarely made explicit: **the minimal unit of interaction is the application**. The human opens Excel to model numbers. Opens Salesforce to manage opportunities. Opens Power BI to review dashboards. Opens ServiceNow to open tickets. Opens Confluence to document. Each application has its screen, its menu, its mental model, its learning curve. The valuable skill of the modern knowledge worker consisted, in large part, of knowing how to operate a reasonable collection of applications well — knowing where each thing is, how to get there, which button to press.

That postulate ceases to operate in the Agentive World. The application, seen as the primary interface of work, **collapses**. But one must be precise about what exactly collapses, because the claim read without nuance can sound more radical than it is.

Not everything disappears. Distinguishing is important so as not to lose credibility before an executive who has to decide a budget. **GUIs as the entry point to work** die fast: the human stops opening Salesforce to review the pipeline and instead asks an agent which opportunities require attention. **Applications as backend systems** survive, but invisibly: the agent that answered about the pipeline queried



Dimension	Before the crossing	After the crossing
Interface	applications (GUIs, menus)	conversation with agents
Data	DW · quality = cleanliness	semantic layer · quality = actionability
Human roles	information consumers	designers and supervisors of agents
Economics of info.	each question = a project (weeks)	each question = a conversation (seconds)
Governance	who can see what?	what can an agent do?
Operating model	competency centers	agent factories

*Advancing in a single dimension without the others produces **successful pilots but no real transformation**. The crossing is systemic or it is not.*

Figure 4: The six dimensions of the crossing

Salesforce via API. Salesforce, as a system, is still there. As a human interface, it is not. **Traditional office suites** — Office, Google Workspace — reposition themselves: Word, Excel, and PowerPoint cease to be the first choice of the user who needs to write, calculate, or present — the agent does it. They survive in cases of fine editing, specific formatting, or creative work where conversation is inefficient relative to direct work. And **specialized tools with a complex surface** — CAD, Figma, advanced IDEs, music DAWs — survive longer: their interface encodes professional knowledge that conversation takes time to replace.

The emerging pattern is sharp. Applications that exist to **navigate and filter information** die fast under the pressure of the Agentive World. They were visual intermediaries between the human and the data, and an agent with direct access to the data makes the intermediary unnecessary. Those that exist to **produce specialized artifacts** survive longer, though eventually with a copilot or agent as mediator. The transition is not uniform across categories of application, and the stack leaders who plan it well accept that unevenness.

A useful image for intuitively grasping what happens: think of the Agentive World as the one in which enterprise applications live **beneath** a conversational layer, not on top of it. The human never sees them, but the agents query them continuously. Salesforce does not disappear: it becomes the database of customer relationships that a sales agent consumes without the human ever seeing its UI. Power BI does not disappear: it becomes an analytical-query endpoint that a financial agent invokes when asked about quarterly performance. ServiceNow does not disappear: it remains the ticket record that an operations agent consults and updates without opening the portal. Applications become **invisible backend infrastructure**. Their value survives as a structured store of data and business logic. They lose their value as interface.

Applications do not disappear. What disappears is the obligation for the human to open them.

The analytics industry — the oldest and most consolidated in enterprise software — was the first to openly accept that the "human opens application" model had hit its ceiling. Tellius frames it with the candor of an actor that has seen the cycle: *"Dashboards still tell you what happened, but rarely why — and never what to do next."* Superwise extends the observation: *"BI was built for a slower business environment — that assumption no longer holds true."* These are not marketing provocations — they are acknowledgments of a persistent operational problem that the BI industry has tried to solve for fifteen years with successive cosmetic redesigns, until it understood that the problem was not cosmetic but structural. The interface itself — the dashboard as a visual artifact the human must open, look at, and interpret — was the bottleneck.

The new economics of information

If I had to single out one change that crossing the Nadella Line produces on an organization's daily operation, it would be this: **the collapse of the marginal cost of an analytical question**. It is the least visible of the six changes and, at the same time, the most transformative. It is the **enabling condition** of everything else this chapter describes.

In the traditional model, every new business question implies a project. The sequence is familiar and painful: the executive poses the question to their BI area; the BI area coordinates with the executive to pin down the scope; the analysts gather the relevant data; the developers build the report or dashboard; the validators confirm the result is correct; the executive receives the answer. The whole process typically takes between four and twelve weeks. The real bottleneck, as is often said in mature organizations, **is not the technology — it is the transfer of knowledge between people**. There are humans in the middle, and each human introduces latency and the possibility of an interpretation error.

The cost of putting the traditional model into operation is no trivial matter either: building the Data Lake → Synapse → Power BI chain, or any modern equivalent, demands a six-figure initial setup, sustained monthly operation, and months until the first useful dashboard (**Chapter 7 develops** the detailed BI cost figures). And all that investment delivers the capacity to answer only those questions someone foresaw when designing the system. The unanticipated questions — the ones the executive really wants to ask when they arrive on Monday with a new intuition — are not on the menu. They wait in the queue, or they are not asked.

When that cost collapses from weeks to seconds, **the very nature** of the relationship between the organization and its information changes. Three immediate effects transform daily operation.

The first is that **analytical capacity becomes elastic**. It adapts in real time to the current need, not to what someone pre-defined months ago. There is no fixed menu: there is unlimited responsiveness within the limits of the available data. The executive who has an intuition on Monday explores it on Monday — they do not wait until Thursday for the BI team to have the dashboard ready. The speed of analytical curiosity ceases to be limited by the infrastructure.

The second is that **iteration replaces specification**. In the traditional model, the executive had to specify in advance what they wanted to see, wait for the result, and from there formulate the next question. The latency of each cycle was weeks, so the questions had to be very well posed — the cost of a framing error was high. In the agentive model, the executive poses a first approximate question, receives the answer in seconds, refines, deepens, discards hypotheses, pursues others. Knowledge emerges from **dialogue**, not from the project. The very form of doing analysis changes: from sequential project to

continuous conversation.

The third is perhaps the deepest: **the questions that were never asked are now asked**. When asking is free, the organization discovers insights it did not even know it needed. Analytical curiosity ceases to be limited by the BI budget. An executive who in the traditional model reserved their queries for the most obvious, highest-return questions — because each one cost the BI area a project — begins to also ask the marginal questions, the exotic hypotheses, the details that in the old model did not justify the cost. And they discover, frequently, that the marginal questions contained the most valuable insights.

This transformation is not merely a quantitative improvement. It is the **enabling condition** of everything else in this chapter. The continuous intelligence cycle we will describe in the next section cannot exist if each iteration takes weeks. The autonomy governance we will describe later makes no sense if the agents do not operate in real time. The transformation of human roles we will describe at the end does not occur if access to knowledge continues to depend on human intermediaries. The collapse of the cost of the question is not a feature of the agents — it is the **precondition** of everything else.

From the classic cycle to the continuous intelligence cycle

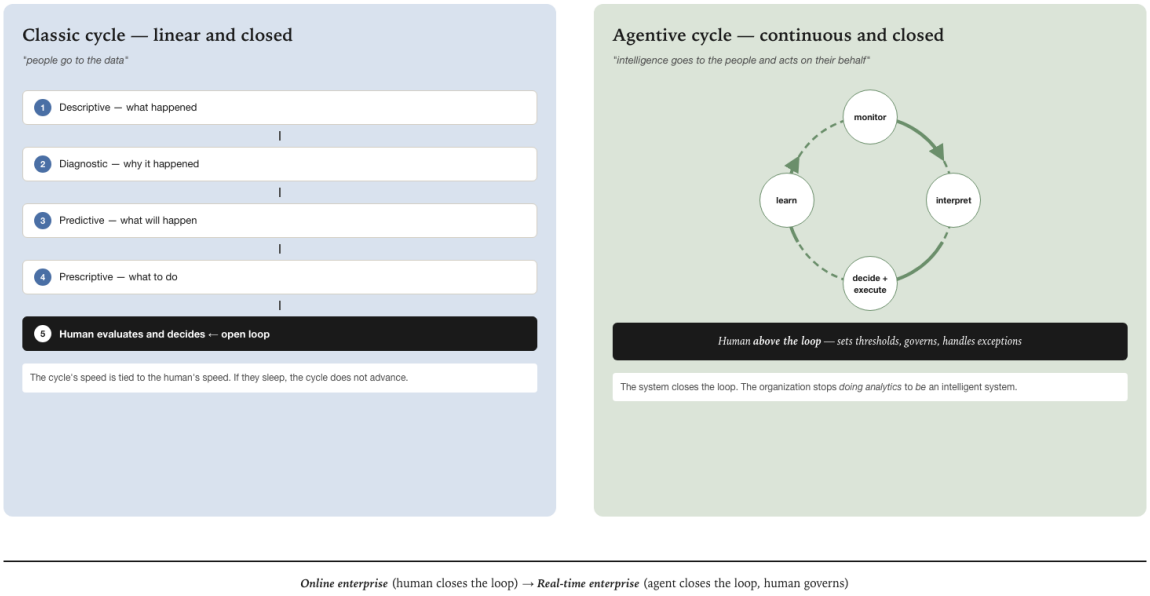


Figure 5: From the classic cycle to the continuous intelligence cycle

For thirty years the information-management paradigm rested on the principle **"people go to the data"**. The phrase sounds anodyne but it encodes the entire operating model of classic BI: you build a data warehouse, you set up dashboards, you train users, and you hope someone looks at the right report at the right moment and makes the right decision. The whole model rests on **human attention** as the scarce resource, the bottleneck around which the system is designed.

The classic cycle is linear: descriptive → diagnostic → predictive → prescriptive → human decides. It starts with data describing what happened, continues by diagnosing why it happened, predicts what will happen, prescribes what to do, and ends with a human evaluating the prescription and deciding. The human who decides is the end of the cycle — the last step, the close. And the speed of the cycle is tied to the speed of that human. If the human is busy, the cycle does not advance. If the human is on vacation, the cycle does not advance. If the human is asleep, the cycle does not advance.

Agentive AI inverts that flow. The canonical phrase of the new paradigm — and you will find it repeated in this book because it is central — is: **"intelligence goes to the people, and acts on their behalf."** A system of agents monitors continuously, interprets what it detects, decides within the limits the organization has defined, executes the decision, and escalates to the human only when warranted — when something falls outside the expected range, when the impact exceeds thresholds, when judgment is required that the agent does not have. The cycle ceases to be linear and becomes **continuous, self-regulating, agent-executed, human-governed**. Its canonical formulation — the same across the trilogy — is **Perceive → Interpret → Decide → Act → Learn**; the short formula "detects, interprets, decides, and acts" that this book uses in passing is its abbreviation, not another cycle.

The critical change is structural. The step from *prescriptive* to action is no longer a recommendation a human evaluates. It is a decision an agent executes, monitors the result of, and adjusts. The organization stops **doing analytics** and starts **being** an intelligent system. This is the formulation the field has begun to use — *"continuous intelligence"* in Gartner's language, *"agentic analytics"* in Tableau's and Tellius's, *"agentic BI"* in Databricks's. They all name the same shift: from the cycle where the human closes to the cycle where the agent closes and the human governs.

The transition between the two cycles gives rise to an organizational distinction worth coining carefully, because it will be a recurring reference throughout the rest of the book. An **online enterprise** has its data up to date, its dashboards current, its information accessible. But it **depends on a human looking, interpreting, and deciding**. It lives with the classic cycle, optimized to the maximum. A **real-time enterprise**, by contrast, does not merely access information: it **detects, interprets, decides, and acts** continuously and autonomously, within governed frames. It lives with the agentive cycle. The boundary between the two is exactly what the collapse of the cost of the analytical question enables — it is what we call **the Quantum Leap**, the event from which the very nature of the operation changes. Before the Leap, an organization can have the best infrastructure in the world and still be an online enterprise. After the Leap, the same infrastructure becomes the substrate of a real-time enterprise.

The distinction matters because it captures something that traditional BI maturity metrics do not. An organization with perfect dashboards, data updated to the second, and the whole BI team running like clockwork, is still an online enterprise if the humans are still the ones who look and decide. The real-time enterprise is not the faster version of the online enterprise — it is **something else**. The difference is not one of speed, it is one of operating model.

In the vocabulary of the architecture the rest of the book develops, the online enterprise and the real-time enterprise are **points on the temporality continuum** of the components that sustain the operation: "real time" is not enabled by choosing a channel that pushes data more often, but by giving **continuous temporality** to the components that operate on the organization's behalf. The spec of manifestation and temporality (`discrete/continuous`) lives in **Chapter 5 §2**.

Cube puts it without rhetoric: *"The modern data stack is beginning to show its age."* BCG takes it to the operational plane, describing how agentive AI orchestrates actions across the whole value chain, *"closing the loop between insight and execution."* The phrase is exact: the classic cycle **left the loop open** — it ended with a recommendation that the human closed with their decision. The agentive cycle **closes**

the loop — the system itself decides and executes, within the frames the human defined. It is a distinct paradigm, not an increment over the current one.

The crossing is not a switch: an organization goes through it gradually, and within it the proportion between assisted work (the agent as a reactive **Assistant**) and autonomous work (the **Autonomous Agent** that closes the loop) shifts as it matures. By way of illustration:

Stage	Assistant	Autonomous Agent
1 · Initial	90 %	10 %
2 · Adoption	70 %	30 %
3 · Maturity	50 %	50 %
4 · Advanced	30 %	70 %

The figures are indicative, not measured: they mark the **direction** of the shift — from a world where the human governs every step to one where the agent sustains the operation and the human governs the frames. The Assistant vs Autonomous Agent distinction is developed in Chapter 5 §5.

Three axes of deep change

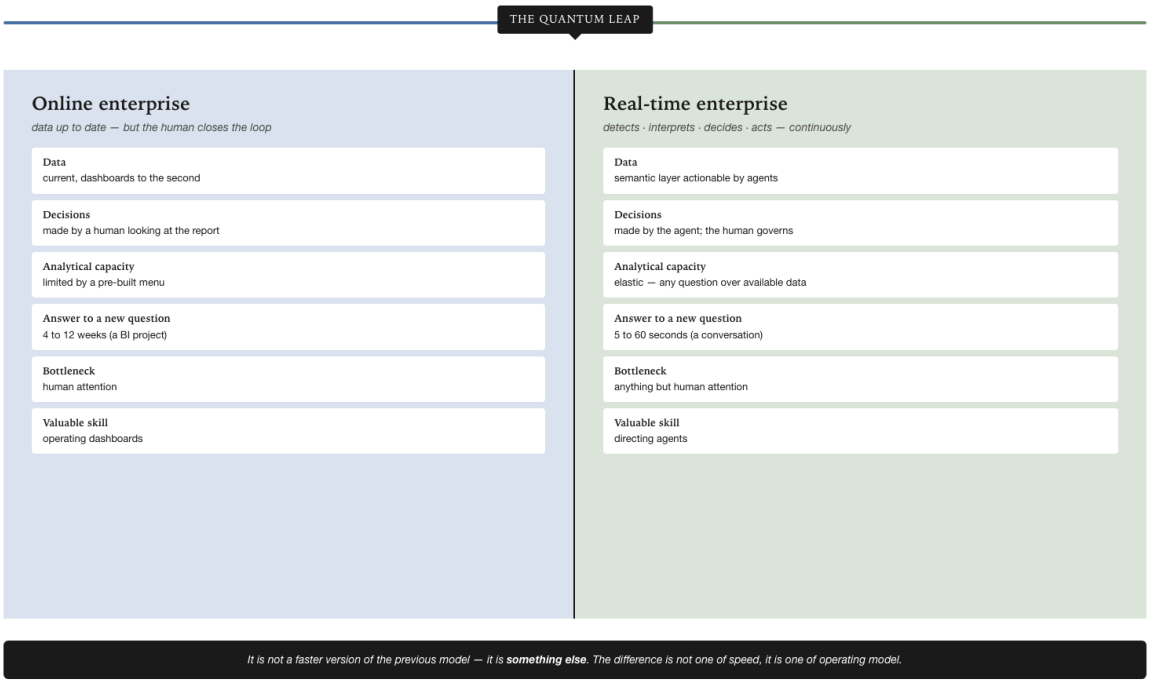


Figure 6: Online enterprise → real-time enterprise · the Quantum Leap

An organization that crosses the Nadella Line experiences three simultaneous axes of transformation — the three axes group the six dimensions of the crossing: the human-information relationship and the roles of human work condense into the first; the data and the operating model, into the second;

governance and the economics of information, into the third. Taken in isolation, each axis sounds like a reasonable improvement. Taken together, they constitute a change of operating model. The warning, recurrent among the firms that have documented the field: advancing on one axis alone without the others produces **successful pilots but no real transformation**. The three are interdependent.

From consuming information to governing agents

The first axis changes the human's relationship with information. In the current paradigm, the knowledge worker is a **consumer of information**: someone builds reports, someone builds dashboards, and the worker reads, interprets, and decides on them. The valuable skill is **data literacy** — knowing how to read tables, understand visualizations, formulate hypotheses from numbers. The value is in *understanding* the information.

In the emerging paradigm, the knowledge worker moves toward a different role: **designer and governor of agents**. People stop looking at reports and move to designing the rules, the thresholds, the protocols under which agents monitor, interpret, and act. The valuable skill is **the design and supervision of autonomous systems** — knowing how to formulate the right rules, knowing how to evaluate the agent's aggregate behavior, knowing how to detect when the agent operates out of range. The value is in *governing intelligent action*, not in consuming information about it.

The change is radical but not sudden. The CFO who in the current paradigm reviews a cashflow dashboard every morning, in the emerging paradigm defines the thresholds and protocols that a financial agent executes autonomously. The agent monitors continuously, executes liquidity-management actions within the limits the CFO defined, and escalates to the CFO only when it approaches the thresholds or when it detects anomalous conditions. The CFO no longer looks at the dashboard — they look at the agent's behavior, adjust the thresholds when they learn something new, intervene when the agent notifies them of an anomaly. The CFO is still the CFO, but their daily work changed in nature.

McKinsey describes this transition precisely in its report on the "Agentive Organization": employees move from executing tasks to orchestrating outcomes, supervising agents, setting objectives, and managing trade-offs. The recurring phrase among analysts — *"humans above the loop"* — captures the shift. The human is not outside the decision loop, nor inside it: they are **above** the loop, defining its parameters and supervising its aggregate behavior. McKinsey estimates that seventy-five percent of current roles will require redesign, upskilling, or reassignment by 2030.

BCG documents a concrete organizational consequence of this shift: forty-five percent of AI leaders expect to need fewer layers of middle management. The reason is structural. Middle management exists in large part to coordinate execution across levels — passing instructions from the executive level to the operational level, monitoring that they are executed, reporting back. When the agent executes autonomously, that coordinative role loses its necessity. What survives of middle management is the part that contributes professional judgment: defining the right rules, handling complex exceptions, mediating between objectives in tension. The pure coordinative part disappears.

New roles appear, symmetrically. An analysis by CIO.com enumerates them in detail: **AI Agent Orchestrator** (the person responsible for the fleet of agents operating in a function or area), **Human-Agent Interaction Designer** (the person who designs how humans interact productively with the agents they govern), **AI Ethics & Governance Specialist** (the person who ensures the agents operate within ethical and regulatory limits), **AgentOps Specialist** (the equivalent of the DevOps Engineer but for fleets of agents). It is a new org chart. It does not replace the traditional org chart immediately — it coexists with it for years — but it reflects the shift of human work toward governing agents that execute.

From architecture for humans to architecture for agents

The second axis changes the data and the underlying architecture. This is the dimension least visible to the executive and most critical to the technical architect. The reason: the data architecture of the current paradigm is optimized for humans to query — and that is structurally incompatible with agents querying correctly.

The current paradigm assumes that the final consumer of the data is a human operating an application. Data warehouses are optimized for SQL queries written by analysts. Data quality means **cleanliness**: rows without nulls, consistent formats, dates in their place. Data models are designed to feed visualizations — Power BI, Tableau, Looker. The integration between systems operates by batches or on demand, at frequencies the human can tolerate.

The emerging paradigm changes each of these assumptions. Data is consumed by agents, which do not read rows — they interpret **meaning**. Warehouses on their own are insufficient: they need an explicit semantic layer on top, where the agent reads not only the tables but the **intent of the tables**: what this indicator means, how it relates to those others, what transformations are legitimate, what special cases apply. Data quality ceases to mean cleanliness and comes to mean **actionability** — why an impeccably clean datum can be worth less to an agent than an imperfect but context-rich one is something the canonical application of Chapter 7 develops.

AtScale measured this difference quantitatively. According to AtScale, agents that query data without a semantic layer fail on more than **eighty percent** of queries — they generate incorrect SQL, misinterpret metrics, hallucinate relationships that do not exist; the same agents with an explicit semantic layer reach, in the same AtScale study, very high accuracy. The conclusion AtScale draws admits no ambiguity: *"For AI agents, the semantic layer is not a nice-to-have — it is the foundation that makes AI truly useful."* **Chapter 7** applies this result to the architecture of the canonical case (Varnished Kimball).

ThoughtSpot coined the term **Agentic Semantic Layer** to describe the semantic layer designed natively for agents — dynamic, context-aware, connected to the agent's flows. Salesforce, in its agentive-enterprise architecture, proposes an **Enterprise Knowledge Graph** as the central layer — a knowledge graph instead of a dimensional model, because the graph captures relationships the two-dimensional table cannot. Databricks talks of unifying infrastructure, data, and semantics to enable **Agentic BI**. Each of these vendors is attacking, from its own angle, the same problem — the one the canonical application of Chapter 7 states as a principle: data alone is not enough for the agent.

The industry is still debating the details — knowledge graph versus semantic layer versus ontology versus extended dimensional model — but the recognition that something new is necessary is already consensus. Informatica frames it candidly: *"Because agents act without human approval loops, the data they use must be fully trusted, verified, and monitored."* And it proposes explicit data-quality SLAs: less than five minutes of freshness for transactional agents, less than one hour for analytical agents. The old SLAs — refreshing the data warehouse every night — do not serve systems that act in real time.

Deloitte found that forty-eight percent of organizations cite **data discoverability** as the principal barrier to their agentive strategy, and forty-seven percent cite **reuse**. The data exists, but the agents cannot find it or cannot interpret it. It is an architectural problem, not one of quantity. The transition from the current paradigm to the emerging one demands a paradigm shift in data architecture: from ETL pipelines designed to feed dashboards to semantic fabrics designed for autonomous reasoning.

From access governance to autonomy governance

The third axis changes how the organization exercises control over what the system does. It is the axis where most agentive projects fail, according to the field data, and for that reason it deserves careful attention.

Traditional IT governance asks: **who can see what data?**. The control model is **access**: authenticated identities, assigned roles, permissions granted over specific resources. The question is static (permissions rarely change) and discrete (a user has or does not have access to a resource). The traditional tools — IAM (Identity and Access Management), SSO, RBAC — are optimized for this model. They work well because the subject of access (the human) has a stable identity and the object of access (the resource) has clear granularity.

Agentive governance asks something different: **what can an agent do, under what conditions?**. The subject of control is not a human — it is an agent acting on behalf of a human or organization. The object of control is not an isolated resource — it is a sequence of actions the agent can execute autonomously. The question is dynamic (the conditions change with context), continuous (the agent acts all the time, not only when someone points the cursor), and multi-dimensional (what action, on what data, with what impact, under what threshold).

Traditional IAM tools are insufficient in this model. They are designed for human subjects with stable identity and discrete permissions, not for agents that act continuously with varying degrees of autonomy. The organization that tries to govern an agent with classic permissions quickly discovers that the model does not capture the questions it needs to answer: can the agent execute bank transfers? Yes, but up to what amount without human approval? during what hours? with what level of prior validation? with what subsequent audit? Each question demands a new mechanism that the classic access model does not have.

Regulators have recognized this shift faster than the industry usually accepts. Singapore's IMDA published in January 2026 the first state framework of governance for agentive AI: the **Model AI Governance Framework for Generative AI** (MGF), which — despite carrying generative AI in its name — devotes its body to agents acting with autonomy. The framework establishes, with a clarity uncommon in early regulation, that although the agents act autonomously, *"human accountability continues to apply"*. Organizations must make human accountability meaningful and human-in-the-loop effective over time. The World Economic Forum proposes progressive governance: logging and traceability for all agents, identity tagging per action, real-time monitoring. Its key distinction: *"autonomy entails decision-making flexibility; automation emphasizes execution reliability"* — they are design choices, not inherent properties of the system. The organization chooses how much autonomy to give the agent; it does not inherit it as a technical property of the system.

BCG reports that fifty-eight percent of heavy AI adopters expect a fundamental change in their governance structure in the next three years, and a third believe AI will have more decision authority in the same period. NACD — the national association of corporate directors in the United States — warns that agentive AI impacts board oversight, regulatory compliance, and risk exposure. KPMG's phrase sums the period up well: *"The winners won't be the ones with the most pilots but the ones investing now in scalable data architectures, agent governance models, and workforce readiness."*

The figure that worries most, however, is the operational one. Gartner predicts that more than forty percent of agentive AI projects will be canceled before the end of 2027 — due to costs, unclear business value, or inadequate risk controls. Governance is not optional; it is what separates pilots from production. The scale of the problem is significant: eighty percent of organizations report risky behaviors by

their agents — unauthorized data access, unexpected interactions, out-of-range decisions — and only twenty-one percent have mature governance models. ISACA stresses that agentive AI presents a growing challenge for audit functions because its decision processes lack clear traceability.

The message of the data is sharp. The organizations that survive the crossing will be those that have invested in autonomy governance with the same seriousness with which they invested in access governance over the last twenty years. Those that treat the problem as a late application of the old tools — one more permission in the IAM, one more role in the SSO — will end up on the wrong side of that projection. And autonomy governance is not built in the last month before launch — it is designed from the start, in the very architecture of the system. It is what Chapter 5 develops under the name **Trust Infrastructure**.

The nature of the transition

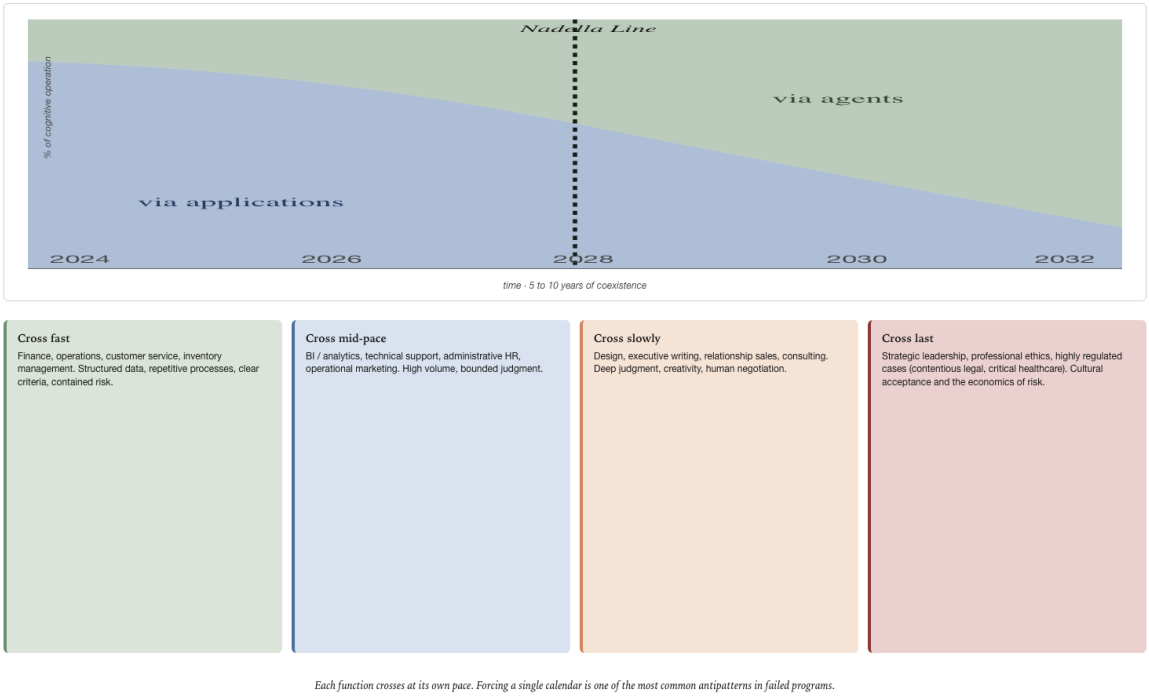


Figure 7: The transition is not an event · evolutionary coexistence

There is a risk of reading the consequences of the crossing as if they were a single event — the day the organization passed to the Agentive side. That reading is mistaken and, in practical terms, dangerous: it leads to planning the transition as a big bang that almost always fails. The reality is **evolutionary coexistence**: the current paradigm and the emerging one coexist for years, with the proportion changing gradually from one toward the other.

The existing infrastructure — data warehouses, ETL pipelines, traditional BI tools, ERPs, CRMs — does not disappear when an organization crosses the Nadella Line. Its role changes. It ceases to be a surface with which the human interacts and becomes a **data source consumed by agents**. The traditional architecture remains valid for large-scale historical data warehousing, for highly complex analytical

models that require pre-computation, for pipelines with very specific business logic, for regulatory requirements of retention and formal lineage. What changes is not whether those systems exist — it is **who consumes them**. In the early stages, ninety percent of consumption is by humans via dashboards and reports; the agent is an occasional assistant at the margin. In the advanced stages, the agent orchestrates most of the analytical queries and the traditional infrastructure operates invisibly underneath, feeding it.

Each stage of the transition does not invalidate the previous one — it subsumes it.

An organization advanced in the transition did not eliminate its data warehouse. It integrated it into a semantic fabric that agents consume autonomously. The online enterprise does not disappear when the real-time enterprise emerges — it becomes its foundation. This is a critical reading so as not to fall into the opposite error of the "big bang": that of agentive fundamentalism, which discards all the existing infrastructure and tries to rebuild from scratch. The serious organization recognizes that twenty years of investment in traditional BI built an asset — the data is clean, the models are agreed upon, the pipelines are reliable — and that this asset is exactly what the agent needs underneath. To throw it away and start over is to renounce the asset. To keep it and add an agentive layer on top is to capitalize on it.

Evolutionary coexistence is not a hope — it is the **documented pattern** of every interface transition the industry has already crossed. When the PC challenged the mainframe, the prophecy was extinction; the inflection point came in 1984 — desktop computer sales surpassed mainframe sales for the first time — and yet mainframes today process between 2.5 and 3 billion transactions a day, sustaining the world's financial markets. When the web challenged installed software, and when the cloud challenged on-premise, the pattern repeated, always with four constants: **the initial prediction is extreme** (total death of what came before); **the reality is coexistence** (the "doomed" technology finds niches of irreplaceable value); **the real timeline multiplies the predicted one by three to five**; and **the stable state is hybrid** — layers stratify, they do not replace one another. The Nadella Line will inherit the pattern: the crossing will be asymmetric, slower than its headlines, and applications will long survive as substrate — which is exactly what evolutionary coexistence plans for instead of denying.

The transition has, in addition, two asymmetries worth recognizing. The first, **asymmetry across functions**: not all functions cross at the same pace. Repetitive functions with structured data — finance, operations, customer service, inventory management — cross fast, because the economics are evident and the risk of error is contained. Creative functions or those of deep judgment — design, strategy, negotiation — cross more slowly, not for technical incapacity of the agents but for cultural acceptance and the economics of risk. Forcing a uniform crossing across all functions is one of the most common antipatterns in failed agentive transformation programs. Serious organizations let each function cross at its own pace, with specific plans per area, rather than imposing a single calendar.

The second asymmetry is **temporal**: organizations that invest early in agentive architecture pay the cost of exploring before having mature reference points, but they also capture the learning that comes from operating the new model for longer. Those that wait for the field to mature pay less exploration cost but reach the market with less operational experience. The choice between the two positions is not obvious — it depends on risk appetite, on competitive position, on the decision-maker's time horizon. But the choice must be conscious. Remaining in the middle — investing enough to have a pilot but not enough to reach production — is the worst possible position. It is exactly the position that swells Gartner's cancellation projection.

The state of the field

Treating the Agentive World as a distant horizon is a diagnostic error. The field data at the start of 2026 documents a transition already under way, with a speed significantly greater than what traditional marketing suggested eighteen months ago. Three figures give dimension to the phenomenon and frame the reading of the rest of the book.

The first: by the end of 2026, **forty percent of enterprise applications will include AI agents**, compared to less than five percent in 2025. The projection is Gartner's, and the order of magnitude — an eightfold growth in twelve months — is what makes it remarkable. It is not the absolute figure that matters; it is the slope. A slope like that implies the transition is accelerating, not stabilizing.

The second: by 2028, **at least fifteen percent of daily operational decisions will be made autonomously by agents**, with no human in the decision loop. The figure is a Gartner projection. Fifteen percent seems little, until one calculates how many operational decisions a mid-sized company makes per day — on the order of tens of thousands across all its areas — and understands that fifteen percent is a significant volume of daily autonomous operation, with regulatory, organizational, and technical implications that companies have not yet fully mapped.

The third: the global agentive AI market goes from **7.3 billion dollars in 2025 to a projected 139.2 billion by 2034** — a compound rate above forty percent per year sustained over a decade. This figure is the collective bet of capital on the agentive horizon. When capital bets like that, it does not guarantee that the transition will occur as predicted — but it does guarantee that the organizations betting against it will have to justify the bet against massive investment in the opposite direction.

There are three possible readings of these data, all valid, and the serious organization must hold all three simultaneously without their contradicting one another. The first reading: **the crossing is happening**. It is not a future scenario over which to debate whether it will arrive — it is a measurable present. The second: **but most are poorly prepared** — the governance data already cited (widespread risk, scarce maturity, projects on the way to failing) confirms it. The third: **the difference between success and failure is structural**. The organizations that survive the crossing will be those that have invested in formal agentive architecture, not in isolated pilots without discipline.

The three readings are, ultimately, the same. There is a real transformation occurring. There is a real cost to doing it badly. And the difference between those who will do it well and those who will do it badly is not subtle — it is architectural.

The architectural obligation

The consequences of the crossing that this chapter has documented are not aspirational — they are **technical requirements**. Each consequence has a direct architectural implication, and the list of implications, read as a whole, is almost exactly the list of the four layers and the cross-cutting infrastructure that Chapter 4 will propose as the formal architecture.

The collapse of the application as interface demands a channel-agnostic **Interaction** layer. If the human is going to converse with agents instead of opening applications, the system must be able to manifest itself coherently in chat, voice, corporate channels, GUI on-the-fly, and eventually in any new channel that appears. A rigid Interaction layer, tied to a particular surface, fails in the Agentive World.

The new economics of information demands a **Cognition** layer that can reason over data in real time, multi-LLM, capable of applying specialized knowledge and of delegating repetitive tasks to another layer

that executes them without invoking it each time. A monolithic cognition, tied to a single provider, fails when the economics of operation demand otherwise.

The transformation of human roles toward governing agents demands that the agent have **persistent life** — that it not be merely a reactive assistant that responds and forgets. An **Autonomy** layer where the agent can maintain state, execute continuously, monitor, and act on its own initiative is what makes the human *above the loop* instead of *in the loop*. Without that layer, the human remains a bottleneck even while believing they are governing.

Autonomy governance demands a control point where policy is exercised before execution. An **Access** layer where every action of the agent on the real world passes through policy validation, auditable logging, and configurable controls is what separates pilots from production. Without that layer, the system is indefensible both regulatorily and operationally.

And the transition from pilots to production demands a **cross-cutting trust infrastructure** — one that does not live in a specific layer but cuts across all four. It is what Chapter 5 develops under the name Trust Infrastructure: five pillars — Governance, Auditing, Validation, Resilience, Transparency — exercised at different points depending on the case but which together sustain the organization's trust in what the system does.

Each requirement is a layer. Each layer is the answer to a consequence of the crossing. The agentive architecture is not an abstract proposal designed on a whiteboard — it is the list of things that must be resolved so as not to swell those cancellations. That architecture, its layers, its primitives, its formal interfaces, is what the rest of the book develops.

Whoever has followed this chapter now understands what changes upon crossing the Nadella Line, why the crossing is systemic, and what is demanded of an architecture that aims to sustain it. The following chapters deliver that architecture. The reader who is going to make stack decisions on a five-year horizon will find in them the discipline that prevents it.

Visual summary

Dimension	Before the crossing	After the crossing
Interface	Applications (GUIs, menus, screens)	Conversation with agents
Data	Data warehouses optimized for SQL · quality = cleanliness	Semantic layers where agents reason · quality = actionability
Human roles	Consumers of information · executors of tasks	Designers and supervisors of autonomous systems
Economics of information	Each new question = a project (weeks)	Each new question = a conversation (seconds)
Governance	Who can see what data · static permissions	What an agent can do, under what conditions
Operating model	Centers of competence · specialized functions	Agent factories · ecosystem orchestration

This page intentionally left blank.

Chapter 3 · Bounded Concerns Architecture

Chapters 1 and 2 established the frontier and described the destination. The Nadella Line separates two futures of software, and the Agentive World is the side organizations will cross to over the coming decade. But before describing the architecture of the destination — the task of Chapter 4 — we must look carefully at the place from which one departs. The vast majority of organizations that will cross the line will not do so by building from scratch: they will do so by transforming a patrimony of enterprise systems that have operated for decades, sustain the heart of the business, and cannot be switched off during the crossing.

This chapter describes that point of departure with the same rigor with which the rest of the book describes the destination. The instrument we will use is called **Bounded Concerns Architecture** — BCA for short. It is not an architecture of the Agentive World: it is the formal cartography of the state *prior* to the crossing, designed so that an architect can look at an existing enterprise system, identify where each responsibility lives, and reason in a disciplined way about which components will migrate to the Agentive World, which components will disappear, and which components will remain as invisible backend infrastructure that agents consume without the human ever seeing it.

The choice of name is not ornamental. *Bounded* captures the operational principle that sustains the entire architecture: each responsibility occupies its place and does not overflow into the territory of the neighboring responsibility. That discipline, which in the pre-agentive era was a recommended best practice, becomes a **condition of viability** the moment the system begins to incorporate agentic components. Without that discipline, the statistical volatility that agents bring with them contaminates the stable core of the business, and what had worked for twenty years ceases to be reliable. With that discipline, the agentic incursion stays confined to the layer that is designed to tolerate volatility, and the business core stays protected throughout the crossing.

> *BCA is the architecture of the transition, not the architecture of the destination. It describes the state from which one crosses the Nadella Line, not the state into which one crosses.*

The Agentic era demands explicit architectural treatment

The incorporation of agents into enterprise systems is not the addition of yet another new technology. It is the introduction of a class of behavior whose nature is structurally distinct from traditional code, and which therefore demands distinct architectural treatment.

Traditional code — workflows, rule engines, service classes, orchestration scripts — has a common property: its behavior is **explicitly specified** by humans. Someone wrote, step by step, what happens when,

in what order, under what conditions. To understand what it does, it is enough to read the code. Its evolution is deliberate: when it changes, someone decided to change it and modified the corresponding code.

Agentic code — agents based on language models, recommendation systems, machine-learning optimizers, expert systems — has the opposite property: its behavior is **contextually derived** from a model. No one programmed step by step what it does; someone described an objective or a pattern, and the entity derives the behavior from its weights, training data, prompts, or heuristic rules. To understand what it does, it is not enough to read the code — one must understand the data or the model from which the behavior emerges. Its evolution is statistical: when it changes, it is not because someone modified a line but because the data, the weights, or the prompts that synthesize the reasoning changed.

This difference has four concrete architectural consequences that separate agentic components from traditional ones. Procedural components are tested with deterministic unit tests; agentic ones are tested with evaluation datasets and aggregate metrics. Procedural ones produce predictable logs that reflect code execution; agentic ones produce logs that require interpretation because the behavior varies with context. Procedural ones are auditable line by line; agentic ones are auditable by their inputs, outputs, and metrics but not by their internal logic. Procedural ones evolve through code refactoring; agentic ones evolve through retraining, prompt tuning, or model swaps. These four properties are distinct enough to justify the architecture treating them as separate citizens.

The structural threat this difference introduces into the enterprise system is the following: **agents are volatile, but the invariants of the business cannot be**.

An agent evolves rapidly. It changes with every update to the underlying model, with every prompt adjustment, with every incorporation of new tools or new data. Its behavior can vary between two identical invocations. Its correctness is not binary but statistical: it operates correctly most of the time, with a tolerable error threshold that depends on the use case.

The invariants of the business are the opposite. A customer has a unique identifier. An invoice has a total that equals the sum of its lines. An activated service has an activation date. These rules admit no statistical errors: either they hold always or the system ceases to be reliable. Their correctness is binary, their rate of change is structural, and their violation has serious regulatory, financial, or operational consequences.

When an agent and an invariant coexist in the same architectural layer with no explicit frontier between them, the volatility of the first contaminates the reliability of the second. Every update to the agent risks breaking invariants that had worked correctly for years. Every change of prompt may inadvertently alter compliance with a regulatory rule. Auditing becomes impossible because there is no longer any separation between what was deliberately programmed and what the agent decided contextually.

The thinness of the domain as the answer

The operational answer to this threat is what we will call **the thinness of the domain**. If the authoritative core of the system — where the invariants and the persistence of the business truth live — is kept strictly thin, admitting only what is structural to the domain and expelling all volatile logic outward, then the agentic incursion stays confined to the layer that is designed to tolerate volatility. Agents operate in the orchestration logic, alongside the traditional procedural workflows, and both invoke the core as a service with a stable contract. The heart of the business stays protected.

This is the operational position that sustains the entire architecture. Keeping the domain thin is not

an aesthetic preference but a necessary condition for procedural and agentic capabilities to coexist in a single system without the latter corroding the reliability of the former.

The distinction between Systems of Record and Systems of Engagement, articulated by Geoffrey Moore[1], operates convergently with this thesis but at a different level. Moore distinguishes between two categories of systems according to their purpose and their rate of change: Systems of Record store the authoritative state of the business and derive their value from their reliability, which demands stability; Systems of Engagement mediate interaction and derive their value from their capacity for adaptation, which demands speed. Forrester[2] would later extend the taxonomy by adding Systems of Insight, oriented toward knowledge generation. Moore's dichotomy operates at the level of a systems inventory — which systems an organization has; the thinness of the domain operates at the level of code organization within a component. Both converge on the same fundamental intuition, which Buschmann and colleagues[3] articulate in terms of the Layers pattern as a decomposition criterion: each layer should have a distinct and specific responsibility with respect to abstraction, granularity, or **rate of change**. The Agentic era intensifies this logic: agentic components change at an even faster rate than traditional commercial policies — model rate, not business rate — and must therefore be kept architecturally even further from the stable core.

The thesis holds a specific position on the hierarchy of architectural decisions. Other frequently debated decisions — the choice between a monolithic or microservices architecture, the direction of dependencies, the degree of event-sourcing adoption, the command/query separation, the deployment modality of agents — are important but subordinate. They operate within a space whose shape is determined beforehand by the decision about where the domain ends. A microservices architecture with obese domains where agents are introduced reproduces the problems of a poorly structured monolith, now distributed and with additional agentic volatility. A hexagonal architecture with a core that mixes invariants and workflows uses the correct direction of dependencies to protect a poorly designed core, but the incorporation of agents into that core invalidates the protection. A thin domain, by contrast, produces a component whose core is stable, whose rules are auditable, and whose integrations — procedural or agentic — are interchangeable.

Keeping the domain thin requires a concrete architectural operation: drawing an explicit frontier between the domain and the orchestration logic, and defending that frontier against the incremental drift that tends to fatten the core over time. The *Bounded* in the model's name takes up the principle of Separation of Concerns articulated by Dijkstra[4]: the discipline of studying one aspect of a problem in depth, for its own consistency, knowing all the while that it is only one among several. Dijkstra's separation is originally cognitive. Its transformation into an architectural principle over the following decades — visible in patterns such as Layers[5], Hexagonal[6], Clean Architecture[7], and DDD[8] — produces the theoretical lineage to which BCA belongs. The difference between BCA and its predecessors lies in which specific frontier is chosen as the most consequential: BCA chooses the frontier of the domain, and upholds it as

[1] Moore G. "Systems of Engagement and the Future of Enterprise IT — A sea change in enterprise IT". AIIM White Paper, 2011.

[2] Hopkins B. "Systems of insight will power digital business". Forrester Research, July 2014.

[3] Buschmann F., Meunier R., Rohnert H., Sommerlad P., Stal M. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons, 1996.

[4] Dijkstra E. W. "On the role of scientific thought" (EWD447), 1974. Reprinted in *Selected Writings on Computing: A Personal Perspective*, Springer-Verlag, 1982.

[5] Buschmann F., Meunier R., Rohnert H., Sommerlad P., Stal M. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons, 1996.

[6] Cockburn A. "Hexagonal architecture", April 2005. `https://alistair.cockburn.us/hexagonal-architecture/`.

[7] Martin R. C. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall, 2017.

[8] Evans E. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2003.

a condition of viability for the Agentic era.

The three layers

BCA decomposes an enterprise component into three layers. Layer 1, **Presentation**, contains the component's two external frontiers: UI toward humans and API toward systems. Layer 2, **Business Logic**, contains the orchestration logic, organized in two parallel boxes: *Procedural* for explicitly programmed behavior, *Agentic* for contextually derived behavior. Layer 3, **Domain**, contains the stable core of the component, organized along two orthogonal dimensions: own domain versus foreign domain on the vertical axis, logic versus persistence on the horizontal axis.

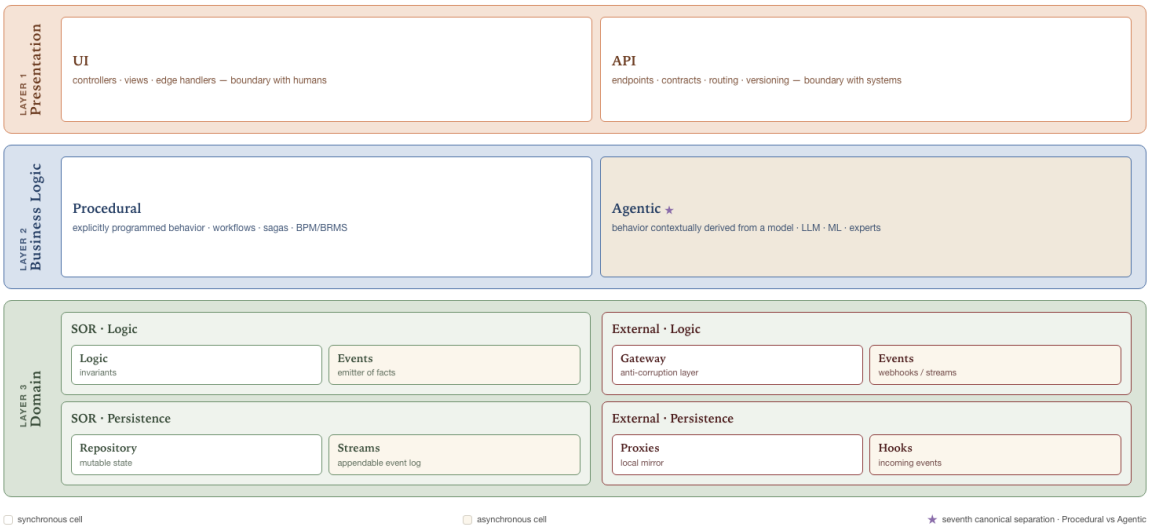


Figure 8: Bounded Concerns Architecture in three layers with the seventh separation Procedural / Agentic in Business Logic.

Layer 1 treats UI and API as parallel citizens, not as variants of the same concern. UI is the controllers, views, and handlers that mediate interaction with humans; it evolves at the rate of UX needs. API is the endpoints, contracts, routing mechanisms, and versioning policies that mediate interaction with other systems; it evolves at the rate of the component's public contracts, under specific constraints of backward compatibility and precise semantics[9]. In the pre-agentive era this separation was a microservices best practice; in the Agentic era it takes on additional relevance, because agents can operate both on the UI (assisting the human user) and on the API (consuming and producing machine-to-machine contracts), and recognizing the distinction makes it possible to manage their respective modes of invocation appropriately.

[9]Newman S. *Building Microservices: Designing Fine-Grained Systems*, 2nd ed. O'Reilly, 2021.

Layer 2 contains the orchestration logic: the logic that knows which steps make up a use case, in what order they occur, what happens if one fails. Here lives the **seventh canonical separation** of the model, which we will return to in detail further on. *Procedural* comprises all logic whose behavior is explicitly programmed by humans: process engines, rule engines, sagas, choreography frameworks, workflow orchestrators, service classes, coordination scripts. The operative distinction is not one of technology but of nature: the behavior is coded step by step by a human and can be understood by reading the code line by line. *Agentic* comprises the components whose behavior is contextually derived from a model, not from a coded sequence. The two boxes share a layer because both are orchestration logic — both operate on the domain without forming part of it — but their distinct operational nature justifies treating them as parallel citizens.

The frontier between Layer 2 and Layer 3 — between orchestration and domain — is the operational frontier of the thesis. Layer 3 contains exclusively logic that the business considers structural; everything else lives in Layer 2, whether Procedural or Agentic. The operative criterion is temporal in nature: if a rule can change as a consequence of a commercial decision without altering the fundamental model of the business — or if the rule is statistically derived rather than explicitly coded — it belongs in Layer 2; if the rule is structural to the domain and changing it would imply a change in the very nature of the business, it belongs in Layer 3.

Layer 3 is organized along two orthogonal dimensions that produce four cells. The vertical dimension separates the own domain from the foreign one: the **SOR** (System of Record) column contains the components that model data whose authority belongs to the system — the authoritative core, what is validated, what is persisted as truth, what emits events when it changes; the **External** column contains the components that know, translate, and locally represent data whose authority lives outside, in systems with which the component collaborates — CRMs, external billing, provisioning OSS, B2B partners, third-party APIs. The horizontal dimension separates logic from persistence: the **Logic** band contains the invariants in SOR and the translations from the foreign model in External; the **Persistence** band contains the physical storage mechanisms.

Each of the four resulting boxes is internally subdivided into a synchronous path and an asynchronous path. The lexical asymmetry between **Streams** (SOR side) and **Hooks** (External side) is deliberate. On the SOR side the system emits its own facts and persists them as an appendable log; authorship is its own and the direction is outgoing. On the External side, events arrive because external systems push them via webhooks or equivalents; authorship is foreign and the direction is incoming. Calling both *Streams* would lose that important directional information.

The seven structural separations

Taken together, the decisions of the three layers produce seven explicit structural separations. Each one responds to the classic observation that differentiated concerns have distinct rates of change, distinct audiences, and distinct semantics, and therefore deserve distinct architectural treatment.

#	Separation	Materialized by
1	Human presentation vs external contract	UI and API boxes in Layer 1
2	Orchestration vs domain rules	Frontier between Layer 2 and Layer 3
3	Logic vs persistence	Horizontal bands in Layer 3

#	Separation	Materialized by
4	Own domain vs foreign	SOR and External columns in Layer 3
5	Synchronous vs asynchronous communication	Parallel paths within each cell
6	Mutable state vs event log	Repository/Streams and Proxies/Hooks pairs
7	Procedural vs agentic behavior	Procedural and Agentic boxes in Layer 2

Separation 2 is the operational frontier of the thesis. Separations 3 through 6 structure the internal content of Layer 3. Separation 1 structures the internal content of Layer 1. And separation 7 — the seventh canonical separation — structures the internal content of Layer 2 by explicitly reflecting the dual nature of orchestration logic in the Agentic era.

Each cell of the architecture has its own rate of change. SOR · Logic and SOR · Persistence change at the rate of the evolution of the fundamental business model — years or decades. External · Logic and External · Persistence change at the rate of the external systems they serve. Business Logic Procedural changes at the rate of commercial policies — months. Business Logic Agentic changes at the rate of the underlying models — weeks or days. API changes at the rate of public contracts. UI changes at the rate of UX needs. This differentiation justifies distinct versioning, deployment, and testing policies for each cell, and it is the structural reason why the seventh separation is necessary: the Agentic rate is fast enough to require its own regime.

Intellectual genealogy

BCA does not invent primitives. It synthesizes patterns published, debated, and refined by the software architecture community over the past two decades, reorganizes them around an explicit operational principle — the thinness of the domain — and repositions them in relation to an emerging phenomenon: the incorporation of agentic capabilities into enterprise systems. Every decision of the model is anchored in prior literature.

The frontier between Layers 2 and 3 — between orchestration and domain — comes directly from Fowler[10]. In *Patterns of Enterprise Application Architecture*, Stafford — author of the *Service Layer* pattern in that work — articulates the distinction between *domain logic*, which has purely to do with the problem domain, and *application logic*, which has to do with application responsibilities such as notifications, integrations, and workflows. Fowler reinforces the distinction in *Domain Logic and SQL*[11] and acknowledges its practical limits in *Organizing Presentation Logic*[12]. The separation between logic and persistence within Layer 3 follows from the *Data Mapper* and *Repository*[13] patterns, whose explicit purpose is to keep the domain model clean of storage details.

[10] Fowler M., Rice D., Foemmel M., Hieatt E., Mee R., Stafford R. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.

[11] Fowler M. "Domain logic and SQL". *martinfowler.com*, February 2003. `https://martinfowler.com/articles/dblogic.html`.

[12] Fowler M. "Organizing presentation logic". `https://martinfowler.com/eaaDev/OrganizingPresentations.html`.

[13] Fowler M., Rice D., Foemmel M., Hieatt E., Mee R., Stafford R. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.

The Domain Layer as a container of invariants comes from Evans[14]. In *Domain-Driven Design* he proposes a four-layer architecture (UI / Application / Domain / Infrastructure) that coincides structurally with BCA, with two deliberate differences: BCA bifurcates the domain into SOR and External, explicitly incorporating integration with foreign systems as a full citizen of the domain; and BCA separates persistence from the domain within the same layer, whereas in Evans persistence lives in Infrastructure.

BCA deliberately departs from canonical DDD on one specific point: the placement of Domain Services. Under canonical DDD, Domain Services — the logic that coordinates multiple aggregates — belong to the Domain Layer. BCA places them in Business Logic. The reason is the temporal asymmetry articulated earlier: Domain Services frequently contain commercial policies that change with the business (eligibility policies, discount rules, approval procedures), whereas the structural invariants of the domain change at a fundamentally slower rate. In the Agentic era this asymmetry intensifies: part of the coordination becomes the responsibility of agents, which adds statistical volatility to the temporal volatility already present. Vernon[15] discusses the tension between Domain Services and Application Services without resolving it in a single direction, recognizing that the choice depends on context. BCA takes a position.

The philosophy of the asynchronous leg is anchored in Helland[16]. *Data on the Outside vs. Data on the Inside* articulates that authoritative data lives inside a service in a transactional world with serializable changes, while data that circulates between services is immutable. A decade later, in *Immutability Changes Everything*[17], Helland goes deeper: domain facts are intrinsically immutable — events that occur cannot be erased; corrections are new events. Kleppmann[18] synthesizes this tradition by articulating that mutable state and event log can coexist as complementary representations of the same domain, a position that is the conceptual basis for the coexistence of Repository and Streams in BCA's SOR column.

Young[19] articulates CQRS by arguing that the command side should be thin and focused on protecting invariants; process logic lives in sagas and process managers that coordinate the SoRs, not within them. BCA adopts the spirit of CQRS without going all the way to strict CQRS: the synchronous cells of Layer 3 are thin, the asynchronous cells exist as complementary citizens, but mutable state and event log coexist rather than one deriving from the other. Fowler[20] synthesizes Young's position in his bliki entry on CQRS.

The External column of Layer 3 generalizes the Anti-Corruption Layer pattern originally proposed by Evans[21] and refined by Vernon[22] with specific focus on how bounded contexts integrate with one another. Vernon explicitly articulates that integration with legacy systems is one of the paradigmatic cases where the pattern applies.

The separation of the API as a distinct architectural concern does not appear in the classic texts of

[14] Evans E. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2003.

[15] Vernon V. *Implementing Domain-Driven Design*. Addison-Wesley, 2013.

[16] Helland P. "Data on the outside versus data on the inside". *2nd Biennial Conference on Innovative Data Systems Research (CIDR)*, 2005. Republished in *ACM Queue* 18(3), 2020.

[17] Helland P. "Immutability changes everything". *7th Biennial Conference on Innovative Data Systems Research (CIDR)*, 2015. Republished in *Communications of the ACM* 59(1), 2016.

[18] Kleppmann M. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly, 2017.

[19] Young G. "CQRS Documents", 2010. Available at `https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf`.

[20] Fowler M. "CQRS". *martinfowler.com*, July 2011. `https://martinfowler.com/bliki/CQRS.html`.

[21] Evans E. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2003.

[22] Vernon V. *Implementing Domain-Driven Design*. Addison-Wesley, 2013.

Fowler[23], Evans[24], or Moore[25], but it is established practice in modern microservices architecture. Newman[26] devotes entire chapters to API design decisions, explicitly separating them from presentation to the user. The asynchronous leg rests on the formal catalog of messaging patterns by Hohpe and Woolf[27] — their 65 patterns constitute the vocabulary of the Events, Streams, and Hooks cells. In the telco context specifically, the TM Forum[28] classifies the functional blocks of the Open Digital Architecture according to Moore's dichotomy: Party Management, Core Commerce Management, and Production as SoR; Engagement Management as SoE; Intelligence Management as SoI. The adoption of the SOR column in BCA is consistent with this classification: if TMF already defines certain functional blocks as SoR, it makes sense for the logic that resides within them to be consistent with the purpose of a SoR according to Moore.

The seventh canonical separation — Procedural / Agentic — and the repositioning of the model as the architecture of the pre-agentive state are anchored in *The Agentive Future*[29] and in this book. There, the Nadella Line is articulated as the conceptual frontier between two possible worlds for enterprise software. BCA takes a specific position: it is the architecture of the Agentic transition, and it yields the floor to the Agentive Architecture framework of the Agentive World (Chapter 4) when the organization crosses the line.

Comparative map of proposals

The proposals mentioned in the genealogy coexist in the architectural conversation as an archipelago where each author draws lines in different places. The figure that follows situates them on a two-dimensional map that makes it possible to visualize how they relate to one another and why BCA's position is a choice, not the only reasonable option.

The map is built on two dimensions whose methodological nature is deliberately distinct. The X axis is **measurable** along a discrete scale of five levels. The Y axis is **qualitative** and is built through an interpretive inference that the following section formalizes.

The **X axis — Domain thickness** measures how much logic the component that handles authoritative data accumulates. The levels are cumulative: each level includes what is in the lower levels. *Level 0 — Pure DAO*: only CRUD over storage, with no invariants or auditing; it does not qualify as a domain in the strict sense. *Level 1 — Strict domain*: adds the aggregate's intrinsic invariants, stable identity, and change auditing. It is the thickness the thesis defends as optimal for the Agentic era. *Level 2 — Multi-entity*: adds operations that coordinate several aggregates of the same domain (Domain Services). *Level 3 — +policies*: adds changeable commercial rules (campaigns, discounts, eligibility restrictions). *Level 4 — +workflows*: adds orchestration of complete processes with multiple steps and external integrations.

The **Y axis — Theoretical rigor vs real-world flexibility** measures how dogmatic the proposal is about requiring its separation. At the purist pole are the proposals that hold *"if you don't separate strictly,*

[23] Fowler M., Rice D., Foemmel M., Hieatt E., Mee R., Stafford R. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.

[24] Evans E. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2003.

[25] Moore G. "Systems of Engagement and the Future of Enterprise IT — A sea change in enterprise IT". AIIM White Paper, 2011.

[26] Newman S. *Building Microservices: Designing Fine-Grained Systems*, 2nd ed. O'Reilly, 2021.

[27] Hohpe G., Woolf B. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2003.

[28] TM Forum. "ODA functional architecture", IG1167, Version 5.1.0, March 2021.

[29] Obach C. *The Agentive Future*. Technical document, ultraBASE — Grupo Ultra, 2025. Material absorbed into Chapters 1 and 2 of this book.

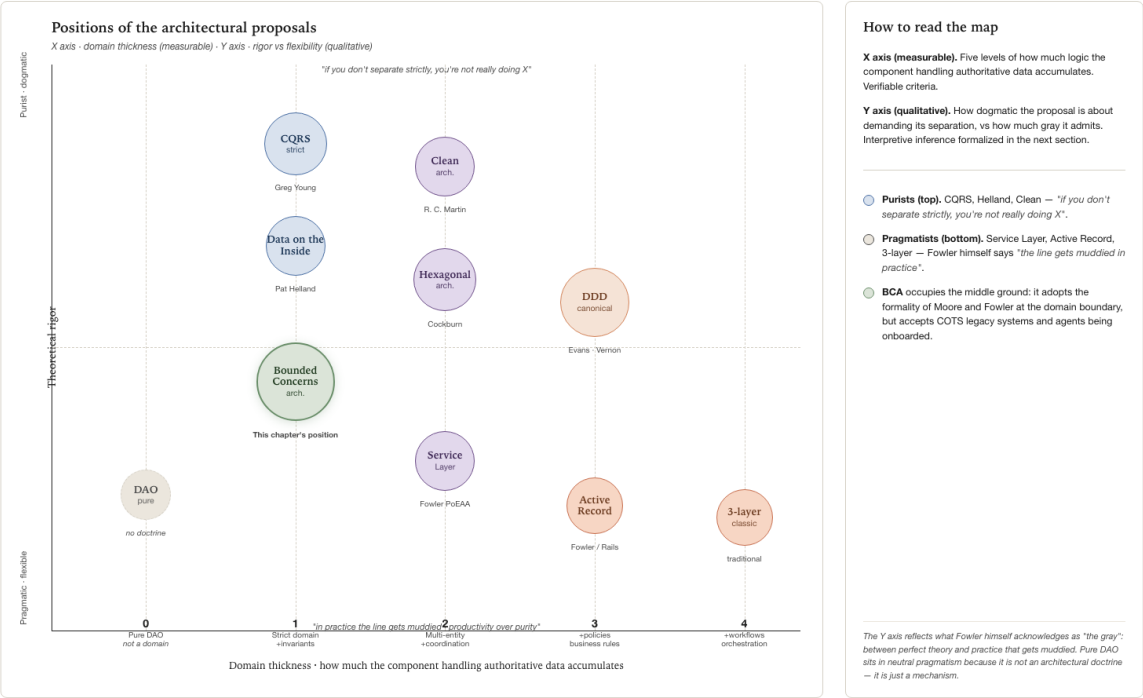


Figure 9: Comparative map of architectural proposals in the space “domain thickness” × “theoretical rigor vs real-world flexibility”.

you're not really doing X". At the pragmatic pole are the proposals that explicitly acknowledge that the conceptual line gets messy in practice. The pragmatic pole reflects exactly the caveat that Fowler[30] acknowledges about layered architectures.

The positions reflect the stance of the original author of each proposal, not the interpretations that the various communities have made of them. In the upper-left quadrant — purists with a minimal core — live strict CQRS[31] and Data on the Inside[32]. Both proposals demand a thin domain and are dogmatic about it. In the upper-middle quadrant — purists with architectural frameworks — live Clean Architecture[33] and Hexagonal[34]. Both define strict rules about the direction of dependencies and the separation between domain and infrastructure, but allow a certain multi-entity richness within the core. In the upper-right quadrant — purist with a thick core — lives canonical DDD[35], refined by Vernon[36]. It accumulates more logic within the Domain Layer through Domain Services but with strict discipline over Bounded Contexts, Anti-Corruption Layers, and well-delimited aggregates.

In the intermediate zone sits **Bounded Concerns Architecture**. On the X axis it coincides with CQRS and Helland (level 1, strict domain). On the Y axis it lands in the intermediate zone: it adopts the formality of Fowler and Moore at the *frontier* of the domain (what enters and what does not), but is pragmatic about the *internal implementation* of the component, recognizing that in enterprise systems with COTS products one does not have control over the internal organization of purchased software, and that in the Agentic era the Agentic components introduce an additional layer of variability that requires operational flexibility.

In the lower-middle quadrant — declared pragmatic — lives Service Layer[37]. It is the most explicitly pragmatic of the mature proposals: Fowler[38] himself writes that the distinction between application logic and domain logic gets messy in practice. In the lower-right quadrant — pragmatics with a thick core — live Active Record[39] and the classic 3-layer[40]. Active Record was designed for productivity — Rails's motto *"convention over configuration"* is its ethos. The classic 3-layer, without additional architectural discipline, tends to accumulate entire workflows within the Business Logic layer mixed with persistence.

The map is an orientation tool, not an official classification. Three reading precautions. First, the positions reflect the stance of the original author; different communities have interpreted the same proposals with greater or lesser rigor (there is DDD-lite and there is orthodox DDD). Second, the Y axis is qualitative and depends on an interpretive inference that the following section formalizes. Third, this map covers proposals for organizing the interior of a component; it does not speak to integration patterns

[30] Fowler M. "Organizing presentation logic". `https://martinfowler.com/eaaDev/OrganizingPresentations.html`.

[31] Young G. "CQRS Documents", 2010. Available at `https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf`.

[32] Helland P. "Data on the outside versus data on the inside". *2nd Biennial Conference on Innovative Data Systems Research (CIDR)*, 2005. Republished in *ACM Queue* 18(3), 2020.

[33] Martin R. C. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall, 2017.

[34] Cockburn A. "Hexagonal architecture", April 2005. `https://alistair.cockburn.us/hexagonal-architecture/`.

[35] Evans E. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2003.

[36] Vernon V. *Implementing Domain-Driven Design*. Addison-Wesley, 2013.

[37] Fowler M., Rice D., Foemmel M., Hieatt E., Mee R., Stafford R. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.

[38] Fowler M. "Organizing presentation logic". `https://martinfowler.com/eaaDev/OrganizingPresentations.html`.

[39] Fowler M., Rice D., Foemmel M., Hieatt E., Mee R., Stafford R. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.

[40] Buschmann F., Meunier R., Rohnert H., Sommerlad P., Stal M. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons, 1996.

between components, which are orthogonal and are cataloged in other works[41][42].

Scoring model for the qualitative axis

The Y axis of the map situates the proposals along a qualitative dimension of "theoretical rigor vs real-world flexibility". This position does not come from the sources — Fowler never wrote that CQRS is more purist than Service Layer in formal terms — it is an interpretive inference constructed from reading the sources.

For that inference to be verifiable and debatable, "pragmatism" is decomposed into five observable dimensions, each scorable from 0 to 3 according to explicit textual criteria. The resulting sum (0 to 15) situates each proposal on the Y axis, where 0 represents maximum purist rigor and 15 represents maximum pragmatic flexibility. This model does not pretend to produce a definitive score. It aims to make visible and disputable the reasoning that led to each placement. Any reader with access to the primary sources can recalibrate the cells and obtain a different position — that is desirable, not a defect.

D1 — Normative language. Does the proposal use phrases of the type *"you must / never / always / strict / must"* about the separation, or does it use *"consider / often / it depends / frequently"*? **D2 — Acknowledgment of the practical gray.** Does the proposal explicitly admit cases where applying the rule causes harm, or does it defend purity in every scenario? **D3 — Permissiveness toward lite variants.** Are there flexible variants tolerated or promoted by the original author? **D4 — Focus.** Does the proposal aim to reorganize the entire system, or is it offered as one pattern among many in a toolbox? **D5 — Compatibility with legacy systems.** Does the proposal offer paths to integrate with existing systems that do not follow it, or does it demand greenfield?

Applying this to the evaluated proposals produces the scores in the table that follows.

Proposal	D1	D2	D3	D4	D5	Total
Strict CQRS[43]	0	0	0	2	2	**4**
Clean Architecture[44]	0	1	1	0	2	**4**
Data on the Inside[45]	1	1	1	1	1	**5**
Hexagonal[46]	1	1	2	1	2	**7**
Canonical DDD[47]	1	2	2	1	2	**8**

[41] Hohpe G., Woolf B. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2003.

[42] Newman S. *Building Microservices: Designing Fine-Grained Systems*, 2nd ed. O'Reilly, 2021.

[43] Young G. "CQRS Documents", 2010. Available at `https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf`.

[44] Martin R. C. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall, 2017.

[45] Helland P. "Data on the outside versus data on the inside". *2nd Biennial Conference on Innovative Data Systems Research (CIDR)*, 2005. Republished in *ACM Queue* 18(3), 2020.

[46] Cockburn A. "Hexagonal architecture", April 2005. `https://alistair.cockburn.us/hexagonal-architecture/`.

[47] Evans E. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2003.

Proposal	D1	D2	D3	D4	D5	Total
Bounded Concerns Architecture	**2**	**2**	**2**	**2**	**1**	**9**
Service Layer[48]	3	3	3	2	2	**13**
Active Record[49]	3	3	2	3	2	**13**

The scores produce a correlation that supports the chapter's thesis. The proposals with the lowest score on the Y axis (strict CQRS: 4/15; Clean Architecture: 4/15; Data on the Inside: 5/15) are also the ones that keep the thinnest domains when applied with discipline. The proposals with the highest score (Service Layer and Active Record: 13/15 both) tend to produce variable domains because their flexibility permits it.

BCA occupies an intermediate position (9/15) that the thesis defends as pragmatically optimal in enterprise contexts with legacy systems and agentic capabilities under incorporation: firm enough at the frontier of the domain to preserve its thinness, flexible enough in the internal implementation to coexist with purchased software, legacy process engines, and agents of a statistical nature over which one has no traditional architectural control.

Application case · Activating a new customer's broadband service

To make the three layers and the distribution of responsibilities tangible — including the seventh separation Procedural / Agentic — consider a case from the telecommunications domain: activating a new customer's broadband service. It is a workflow representative of everyday operations under the TM Forum ODA framework[50] that touches both the own domain (customer, service, subscription) and foreign domains (network provisioning OSS, external billing, email provider), and that is enriched in the current era by the incorporation of a conversational assistant that supports the call-center agent and an automated content generator for the communication to the customer.

The distribution of the actions across BCA's cells is as follows:

Action	Layer	Specific cell
Receive HTTP request from the call-center agent	1 · Presentation	UI
Validate the API contract (fields, types, authentication)	1 · Presentation	API

[48] Fowler M., Rice D., Foemmel M., Hieatt E., Mee R., Stafford R. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.

[49] Fowler M., Rice D., Foemmel M., Hieatt E., Mee R., Stafford R. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.

[50] TM Forum. "ODA functional architecture", IG1167, Version 5.1.0, March 2021.

Action	Layer	Specific cell
The conversational assistant suggests the optimal plan based on customer context	**2 · Business Logic**	**Agentic**
Verify plan eligibility according to geographic coverage	2 · Business Logic	Procedural
Query the OSS for an available port	3 · Domain	External · Logic → Gateway
Cache the OSS response locally	3 · Domain	External · Persistence → Proxies
Apply current campaigns and promotional discounts	2 · Business Logic	Procedural
Validate that the customer has no other active service of the same type	3 · Domain	SOR · Logic
Atomically change the state to `active` with timestamp	3 · Domain	SOR · Persistence → Repository
Emit `ServiceActivated` event	3 · Domain	SOR · Logic → Events
Persist the event in the authoritative log	3 · Domain	SOR · Persistence → Streams
Receive webhook from the OSS confirming provisioning	3 · Domain	External · Logic → Events
Persist the webhook in the external event log	3 · Domain	External · Persistence → Hooks
Notify billing of the new service	2 · Business Logic	Procedural
Generate a welcome email personalized to the customer's profile	**2 · Business Logic**	**Agentic**

Five observations follow from this distribution. **First**, most of the behavior lives in Business Logic, distributed between Procedural and Agentic. Only four of the fifteen actions touch the own domain, and two touch the foreign domain; the remaining nine live in Layer 2 or in Presentation. The Agentic era adds nature to the behavior of Layer 2 without touching the core of the domain. **Second**, Procedural and Agentic coexist naturally in the same layer: the two agentic actions are not isolated in a special layer nor treated as exceptional cases — they are full citizens of Business Logic, interleaved with the procedural actions according to the logic of the workflow. **Third**, the agentic actions do not touch the domain: they operate exclusively in Business Logic, take domain information as input and produce output that returns to Layer 2 or Layer 1, but never directly modify the authoritative state of the SOR. It is the operational materialization of the thesis: the invariants of the domain stay protected from agentic volatility because the frontier between Layers 2 and 3 is respected strictly. **Fourth**, the two communication paths coexist naturally: the synchronous actions use the synchronous cells of each box, the asynchronous actions use the asynchronous cells, and no workflow is obliged to use one path or the other exclusively[51]. **Fifth**, the core of the domain stays intact and thin: the four actions marked with SOR are precisely what the thesis identifies as the authoritative core — invariants and persistence, with their asynchronous counterparts. All the volatile logic — whether procedural or agentic — lives outside the domain.

[51] Hohpe G., Woolf B. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2003.

Architectural implications

Adopting BCA has specific implications for the design of enterprise components in the Agentic era. **Components that play the role of SOR should expose thin APIs centered on operations that preserve invariants**, consistent with Young's[52] vision of the command side in CQRS. The procedural and agentic orchestration components should be explicit and separate; they should not be disguised as extensions of the domain. Persistence should be interchangeable without touching the domain logic. Integration components (Gateway, Proxies, Events, Hooks) should be treated as modelers of a foreign domain[53][54]. Asynchronous components (Events, Streams, Hooks) should be treated as full citizens of the design with their own versioning logic, their own delivery semantics, and their own SLAs[55].

The placement of AI and autonomous-agent logic deserves specific attention. Intelligence and agent components — language-model-based orchestrators, recommenders, optimizers, expert systems — are by nature coordinating process logic. Their correct place in BCA is the Agentic box within Business Logic, not Domain. This preserves the stability of the SOR while allowing the agentic logic to evolve rapidly. The asynchronous leg of the domain (the Events, Streams, Hooks cells) is particularly useful for integrating agents: they can subscribe to domain events to react autonomously without coupling to the core of the SOR. The seventh separation within Business Logic also makes it possible to apply specific governance regimes to each type of component: deterministic testing versus dataset evaluation, line-by-line observability versus aggregate metrics, code-based auditing versus input/output auditing, evolution by refactoring versus evolution by retraining. These differentiated regimes are critical for maintaining traceability and regulatory compliance in systems that combine both types of behavior.

The coexistence of mutable state and event log is an explicit architectural decision of BCA. Mutable state and the event log coexist as complementary representations of the domain, not as substitutes. This places the proposal in an intermediate position between traditional systems — where only mutable state exists and events, if any, are emergent and not authoritative — and pure Event Sourcing — where the state is always reconstructed from the event log and the state is not authoritative, only derived. In BCA, the mutable state is the primary source of truth and the event log is an authoritative complement. The Change Data Capture technique discussed by Kleppmann[56] provides the concrete mechanism for keeping both representations synchronized.

Limitations of the model

BCA is one architectural position among several legitimate ones, not the only one. Canonical DDD[57][58] is a respectable architectural position with genuine merits, especially in domains where the business logic is very rich and benefits from being concentrated in domain objects. The choice between canonical DDD and BCA should be made case by case, considering the context. The chapter's thesis holds that in the Agentic era BCA produces better results, but this claim does not extend universally to all contexts.

[52] Young G. "CQRS Documents", 2010. Available at `https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf`.

[53] Evans E. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2003.

[54] Vernon V. *Implementing Domain-Driven Design*. Addison-Wesley, 2013.

[55] Hohpe G., Woolf B. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2003.

[56] Kleppmann M. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly, 2017.

[57] Evans E. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2003.

[58] Vernon V. *Implementing Domain-Driven Design*. Addison-Wesley, 2013.

The frontier of the domain is not always crisp in practice. As Fowler[59] acknowledges, the distinction between *domain logic* and *application logic* gets messy in real systems. There will be ambiguous cases where it is not clear whether a validation is intrinsic to the domain or is business policy. The recommended operative criterion is temporal in nature — if the rule can change through a commercial decision without altering the fundamental business model, it goes in Business Logic; if the rule is structural to the domain, it goes in SOR · Logic — but this criterion does not eliminate the ambiguity: it makes it discussable and resolvable case by case.

The frontier between Procedural and Agentic can blur. A specific limitation of the current era is that the separation between Procedural and Agentic components is not always crisp. A classic BRMS can incorporate heuristic rules with statistical properties; a modern agent can operate within a rigid procedural workflow. The seventh separation captures a fundamental distinction — explicit origin versus contextual derivation of behavior — but its practical application requires judgment in borderline cases, particularly in hybrid systems that combine symbolic reasoning with statistical inference.

BCA does not replace other frameworks. This decomposition is complementary, not substitutive, of other frameworks: TM Forum ODA[60] for functional classification at the level of a systems inventory, cloud-native principles for deployment, integration patterns such as those cataloged by Hohpe and Woolf[61] for communication between components, and the Agentive Architecture framework of the Agentive World (Chapter 4) for systems that have crossed the Nadella Line. BCA applies to the *interior* of a component or set of components that operate in the pre-agentive state.

And the scoring model of the previous section has three acknowledged limitations. First, the scores reflect the author's interpretations of each proposal's texts; an evaluator with access to the same sources may arrive at different scores, especially in dimensions such as D2 and D3 that require judgment about the spirit of the texts. Second, the simple sum of the five dimensions implicitly assumes that they carry equal weight, which is a simplification; in some contexts D5 (compatibility with legacy) may weigh more than D4 (focus), and vice versa. Third, the model evaluates the stance of the original author, not the effective practice of the communities that adopt each proposal.

Mapping to the Agentive World

So far we have described the cartography of the pre-agentive state. The operative question this book poses — and which justifies BCA's presence in it — is the question of the bridge: when an organization crosses the Nadella Line, what happens to each of BCA's cells? Which migrate to the Agentive World and transform into something different? Which remain as invisible backend infrastructure? Which disappear?

The answer is not uniform across cells. The crossing affects each one differently, at different moments, with different consequences. What follows is the cell-by-cell mapping — the piece that turns BCA into an instrument of migration, not a mere description of the past.

(A reading note: the destination column uses the vocabulary Chapters 4 and 5 define — Botlet, Capability, Connector, Trust Infrastructure. On a first read, the trajectory column is enough — what is preserved, what transforms, what disappears —; the destination column earns its full meaning when you come back here after Chapter 5.)

[59] Fowler M. "Organizing presentation logic". `https://martinfowler.com/eaaDev/OrganizingPresentations.html`.

[60] TM Forum. "ODA functional architecture", IG1167, Version 5.1.0, March 2021.

[61] Hohpe G., Woolf B. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2003.

BCA cell	Trajectory on crossing the Nadella Line	Destination in the Agentive World
UI (Layer 1)	Empties progressively. The application as the primary interface of cognitive work collapses, as we described in Chapter 2.	Replaced by **Layer 1 — Interaction** of the Agentive World (conversational modalities, GUI on-the-fly, passive signage, corporate channels). The traditional UI survives only in specialized tools with a complex surface.
API (Layer 1)	Persists and intensifies. The agent that resolved the human's question queries the component's API without the human seeing it.	Becomes a **Connector** within **Layer 4 — Access** (not a Capability). What was a machine-to-machine contract between systems becomes a machine-to-agent contract, discoverable and governed by Trust Infrastructure.
Business Logic — Procedural (Layer 2)	Contracts. The workflows that encode rigid sequences of steps are reabsorbed into agents that compose the steps contextually from the catalog of Capabilities.	Replaced by **Botlets** that dynamically orchestrate the use case. Only the workflows with strict regulatory requirements where line-by-line traceability is non-negotiable survive as Procedural.
Business Logic — Agentic (Layer 2)	Expands until it becomes the dominant behavior of the layer. What in BCA was a parallel citizen of the Procedural becomes the system's center of gravity.	It is the **operational seed of the Botlet** of Chapter 4. BCA's seventh separation is, in retrospect, the first visible crack of the Agentive World within the Agentic World.
SOR · Logic (Layer 3)	Is preserved. The structural invariants of the domain do not change on crossing the line — a customer still has a unique identifier, an invoice still squares with its lines.	Becomes invariant logic within the corresponding **AgencyDomain**. The operative rule is the same: the domain stays thin, now also in the face of Botlets instead of procedural workflows.
SOR · Persistence (Layer 3)	Is preserved. The authoritative state of the business still lives in transactional databases with their ACID guarantees.	Persists as the storage layer of the **AgencyDomain**. What changes is who queries it: no longer human users via applications, but Botlets via Capabilities.

BCA cell	Trajectory on crossing the Nadella Line	Destination in the Agentive World
External · Logic (Layer 3)	Is preserved but repositioned. The logic that translates foreign models into the own model is still necessary, but it now operates within a federated network of AgencyDomains that trust one another.	Reabsorbed into federation patterns between AgencyDomains managed by **Trust Infrastructure** (Chapter 4 describes the mechanics).
External · Persistence (Layer 3)	Is preserved. The local proxies and the received event logs are still the physical mechanism for integrating foreign domains.	Remain as the storage layer of the federation. The agentive novelty is not structural but of governance: Trust Infrastructure records which AgencyDomain read what from whom and under what policy.
Events (synchronous and asynchronous in SOR)	Are preserved and rise in importance. The asynchronous leg of the domain is particularly useful for integrating agents: a Botlet can subscribe to SOR events to react autonomously without coupling to the core.	Become the substrate of coordination between Botlets and AgencyDomains. The immutability of the authoritative log lets Botlets reason about the history of the domain without contaminating the current state.

Four general patterns emerge from the mapping. **First**, **Layer 3 survives almost intact**. The structural invariants and authoritative persistence are what the enterprise system has to preserve during the crossing, and the discipline of the thinness of the domain is what ensures they survive. BCA's Layer 3 is, to a large extent, what becomes the substrate of the AgencyDomains of Chapter 4. **Second**, **Layer 2 transforms profoundly**. The Procedural box contracts; the Agentic box expands until it dominates; both eventually cease to be parallel boxes and become Botlets that compose behavior from the catalog of Capabilities. BCA's seventh separation is the initial crack through which the Agentive World enters the enterprise system; over time, that crack widens until it consumes the entire layer. **Third**, **Layer 1 bifurcates**. The UI empties and eventually disappears as the primary interface; the API intensifies and transforms into a Connector (Layer 4), not a Capability. The transformation is asymmetric: one of the two boxes of Layer 1 dies; the other thrives under a different name. **Fourth**, **Trust Infrastructure appears as a new cross-cutting concern** that was not materialized in BCA. Governance, auditability, and identity and access policies, which in BCA were responsibilities dispersed across the cells, are elevated into an explicit cross-cutting layer in the Agentive World. This is one of the architectural contributions of Chapter 4: making visible what in the pre-agentive era remained implicit in each component.

The mapping also makes it possible to answer operative questions that an architect in transition asks daily. *Do I have to rewrite my SOR to enter the Agentive World?* No: Layer 3 survives. What you have to do is expose it with discipline via Capabilities. *Do my current workflows die?* Not all of them: those with regulatory traceability requirements survive; the rest are reabsorbed into Botlets. *Are my current APIs useful?* Yes, but their contract has to be renegotiated as a Connector (Layer 4), discoverable and governed by Trust Infrastructure. *Do my applications with UI die?* The ones that existed to navigate

and filter information, yes. The ones that produce specialized artifacts survive longer, eventually with a copilot or agent as mediator, as we already described in Chapter 2.

Closing · The frontier of BCA

BCA is the architecture of the pre-agentive state. Its value in this book is twofold. First, it offers a formal cartography of what most of the organizations that will cross the Nadella Line have today: three layers, seven separations, a thin core, a dual orchestration between Procedural and Agentic. Second, and more important, it offers the mapping instrument that makes it possible to reason in a disciplined way about what migrates, what disappears, what remains — and why.

But BCA has an explicit frontier. When an organization crosses the Nadella Line and enters the Agentive World fully, the separations that BCA articulates tend to collapse. Layer 1 becomes a conversation with an agent. The distinction between Procedural and Agentic dissolves because the first box empties progressively. Eventually, even the frontier between Business Logic and Domain becomes mediated by an agentic entity that decides which steps to invoke. At that point, BCA has fulfilled its transitional purpose and yields the floor to a different architectural framework — a framework no longer organized around the separation between domain and orchestration, but around the four layers of the Agentive World: Interaction, Cognition, Autonomy, and Access.

That framework is the one Chapter 4 develops. It does so on the foundation that BCA has just established: we know what we had before, we know what migrates, we know what remains. Now we can describe, with the calm the change deserves, the architecture of the side we are heading toward.

Chapter 4 · Agentive Architecture

Chapters 1 and 2 established the paradigm; Chapter 3 mapped the pre-agentive state from which most organizations will cross the line, and identified which cells of that state migrate and how. This chapter delivers the architectural answer on the side one crosses to. What follows is the **Agentive Architecture** — the canonical technical design of a system built to live on the right side of the line.

The operative question the architecture answers is concrete: how does one build a system in which an agent can reason, persist, act upon the real world, and account for what it does — without those four functions blurring into an indistinguishable magma? The question is not rhetorical. The blur — fusing cognition, autonomy, action, and governance into a single undifferentiated surface — is exactly what produces the pilots that work in a demo and die on the way to enterprise production. It is the recurring pattern behind the forty percent of cancelled projects we documented in Chapter 2. The root cause of that failure is not technical in the sense of missing algorithms or compute capacity: it is **architectural**. The systems do not separate concerns, and without separation of concerns there is no way to reason in a disciplined way about what they do.

The answer this book proposes is **separation of concerns into four layers, organized in a parallel topology**, governed by a cross-cutting trust infrastructure and ordered by a governing principle we will call *Agent First*. The four layers are not an arbitrary division — each corresponds to a distinct architectural concern that can be reasoned about, specified, and implemented independently. The cross-cutting infrastructure is not an additional layer; it is a property that runs through all four. And the governing principle is not a slogan: it is an operative design rule that orders how any dilemma is resolved.

The precision around **parallel topology** matters from the first reading. The numbering of the layers (1 → 2 → 3 → 4) suggests a sequence, but the real operation of the system is not linear. Layers 2 (Cognition) and 3 (Autonomy) are **parallel paths** between Layer 1 (Interaction) and Layer 4 (Access), not stages in series. An operation that enters through Layer 1 may reach Layer 4 by way of Cognition — costly and decisive — or by way of Autonomy — cheap and repetitive, via Botlets. The two paths interact with each other, but neither dominates the other. The section "The parallel topology" develops the consequences.

The urgency of the work is not theoretical. Bain & Company identifies the absence of shared architectural foundations as the root cause of the stall between pilot projects and productive operation. Only twenty-one percent of organizations have mature governance over the agents they operate. The industry still lacks a shared formal architecture that would allow reasoning about these systems with the discipline applied to distributed architectures, operating systems, or networks. This chapter proposes that architecture, not as revealed truth but as a **reasoned point of departure** that any organization or product can adopt, criticize, extend, or replace.

> **A necessary clarification before entering the detail.** The four layers this chapter develops are an **X-ray of the individual agent** — the four behaviors every agent must exhibit,

whether they materialize in a single monolithic block or are distributed among cooperating components. **They are not links in an industrial value chain**, nor slots where one assigns a market product to each. The industrial value chain — who participates in the agentive economy and where each actor positions itself — is developed by Chapter 6. The two lenses are both true and cross cleanly when kept separate: the X-ray describes the agent; the chain describes the ecosystem in which the agent operates. Confusing them produces the reasoning errors typical of discussions about the in-market product portfolio — for example, assuming that each layer "belongs" to one particular product.

The four layers, seen together

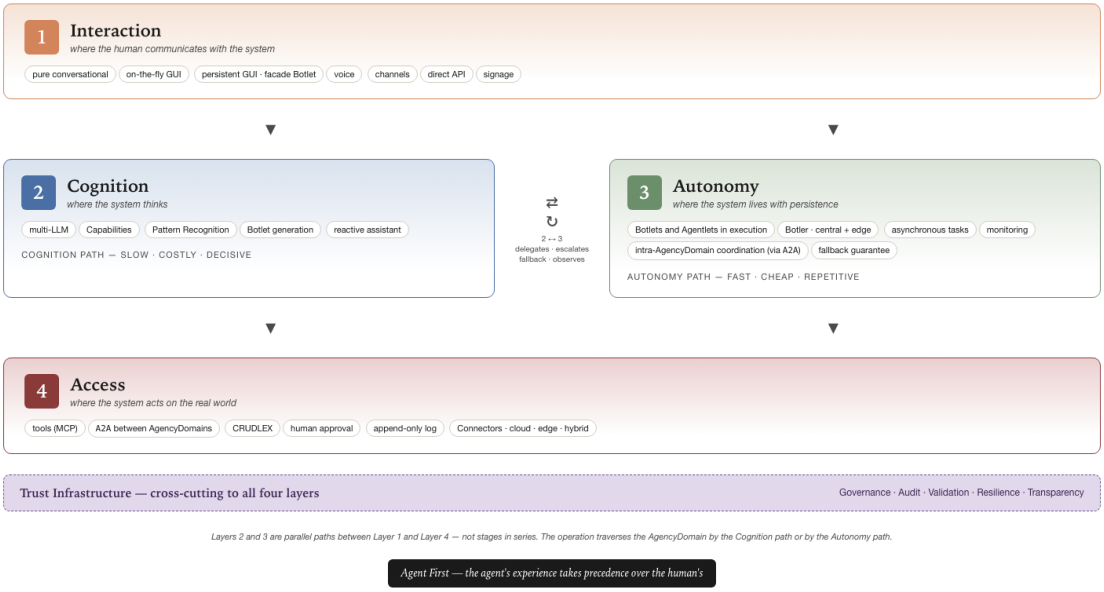


Figure 10: The four layers + cross-cutting Trust Infrastructure

The four layers of the Agentive Architecture are named **Interaction, Cognition, Autonomy, and Access**. The numbering has didactic value — Layer 1 is the surface the human encounters, Layer 4 is the point where the system touches the real world — but **it does not describe the order in which operations traverse the system**. Layer 1 is where the human (or the external agent) communicates with the system; Layer 2 is where the system thinks; Layer 3 is where the system lives and persists; Layer 4 is where the system acts. Layers 2 and 3 are parallel paths, not stages in series — the following section develops this.

The separation matters because each layer solves a distinct problem, demands distinct properties, fails for distinct reasons, and evolves at distinct rhythms. A well-architected system can improve its Layer 1 — adding new interaction channels — without touching the other three. It can change its Layer 2 provider — moving from one model to another — without rewriting its Layer 4. It can strengthen its Layer 3 —

adding persistence or continuous monitoring — without the Layer 1 human seeing any change. This independence between layers is not abstract elegance: it is what allows the system to evolve over years without a complete rewrite, which is exactly what a production system needs in order to survive beyond the first year.

Separation of concerns is a necessary condition for being able to operate agents with confidence.

Each layer is an **architectural concern**, not an attribute. The distinction matters: a system that mixes cognition and action into a single surface does not have a "fused layer" — it has an architectural violation that is paid for on the way to production. The difference between an architectural violation and a "simplifying decision" is practical: in a controlled pilot, the fusion works because the operating volume is low and the human supervises closely; in enterprise production, the fusion makes it impossible to diagnose failures, govern policies, scale volume, or change individual components. The system ceases to be explainable, and a system the team cannot explain is a system the organization cannot operate.

Next, before developing each layer, we formalize the topological model that relates them. After that we present each layer with the detail its role demands, the cross-cutting infrastructure — Trust Infrastructure — and the governing principle that orders the design. To close the chapter we describe the evolution frontier, those vectors where the architecture admits extension that has not yet set as normative spec.

The parallel topology

The mental diagram with which most readers enter the four-layer model is the linear stack: the human interacts with Layer 1, Layer 1 invokes Layer 2, Layer 2 produces a plan, Layer 3 executes it persistently, Layer 4 touches the world. That reading is wrong and produces concrete design errors. The real topology is **parallel**: Layers 2 and 3 are **alternate paths** between Layer 1 and Layer 4, not sequential stages. An operation traverses the AgencyDomain by one of the two paths — or by both across different stretches — but never by the two in mandatory series.

Each path has its own regime. The **Cognition Path** is slow, costly, and decisive — it works well for conversation, new decisions, unanticipated cases, situations where the human needs reasoned dialogue and the system needs to combine Capabilities into new patterns. The **Autonomy Path** is fast, cheap, and repetitive — it works well for executing Botlets over stable patterns, where cognition has already consolidated operative know-how into traditional code that runs without invoking the model. The operating economics of the AgencyDomain depend on the mix: the more operation flows through Path 3, the lower the unit cost; the more flows through Path 2, the greater the capacity to adapt to new cases.

The two paths do not operate in isolation. Three interaction patterns cross between them and we develop them in their corresponding chapter, but it is worth naming them here so the topological model is complete. First, Cognition **delegates to a Botlet** ($2 \rightarrow 3$): when Pattern Recognition detects a repetitive operation, cognition generates a Botlet that will execute the pattern thereafter without invoking it. Second, the Botlet **escalates fallback to Cognition** ($3 \rightarrow 2$): when the environment changes and the Botlet fails, cognition rescues the operation, regenerates the Botlet with the variant incorporated, and returns execution to Path 3. Third, Cognition **observes the Botlet log** ($2 \leftarrow 3$): the Botlet emits events and metrics that cognition consults when the human asks or when it needs to reason about the behavior of the system as a whole.

The parallel topology has five practical consequences worth retaining. The **first** is that the offline mode of an edge node — a physical site without a network — is trivial to explain under this model: the Cognition Path typically depends on cloud and goes inactive without a network; the local Autonomy Path

stays active because its Botlets run on the edge against a local DB and local Connectors. The operation traverses the AgencyDomain by the path that remains alive. What without a parallel topology would seem to require a separate system, under it emerges as a structural property.

The **second** is that cognitive economics becomes evident. The organization does not pay for "the AgencyDomain" — it pays for the mix of paths its operation triggers. Decisions about which patterns to consolidate into Botlets are explicit economic decisions, not an implementation detail. The fine-grained view of this economy distinguishes three rungs of cost, not two: within the Autonomy Path coexist units of zero marginal cost (Botlets) and units of bounded, budgeted inference (Agentlets, Chapter 5 §7) — the middle rung between muscle memory and full Cognition.

The **third** is that **Trust Infrastructure is exercised on both paths**, not only on the one that passes through Cognition. The linear model could suggest that cognition filters everything that reaches Layer 4. The parallel model makes clear that Path 3 also passes through Trust — the policies are applied before invoking Layer 4 regardless of which path the invocation comes from. A Botlet that invokes DTE-SII — the electronic invoicing of Chile's tax regulator — passes through the same Trust validations as the cognition that would do it.

The **fourth** is that the parallel topology distinguishes two types of Botlets that the linear model confused. **Operational-facade Botlets** are invocable from Layer 1 — a button on a POS, a command line, an endpoint — with a stable contract and human identity propagated toward Layer 4. **Cognition internal-tool Botlets** are invocable only from Layer 2 — cognition composes them into plans that it itself executes. Both live in Layer 3, but their invocation surface is distinct and so are their governance properties.

The **fifth** is that the path `Layer 1 → Layer 3 → Layer 4` ceases to be an exception and becomes a **canonical path**. A specialized surface — a floor POS, a kitchen screen, a cashier dashboard, an industrial operation panel — that invokes a senior Botlet and produces an action in Layer 4 traverses this path without touching Layer 2. Under the linear model, that looked like a bypass of cognition, a local decision with a caveat. Under the parallel topology, it is **one of the AgencyDomain's two structural paths**, perfectly legitimate, with its own Trust and observability properties.

The agent's three times

The parallel topology describes **where** each operation lives within the AgencyDomain. This section describes **when** the agent operates — the temporal dimension that the topology alone does not capture. Without this second reading, capacity plans confuse background activity with online activity, and the agent ends up mis-sized: either it is asked for continuous service with no windows to stay capable, or so much maintenance time is reserved that effective operation suffers.

The spec recognizes **three canonical times** of the agent. All three are real and simultaneous activities in a productive system; they differ in their regime, their urgency, and their cognitive economics.

Preparation

Preparation is the time in which the agent **creates and improves its capabilities outside the service window**. It refines its catalog, improves its cognitive capabilities, studies the environment, regenerates Botlets that detected drift, incorporates new variants, trains Pattern Recognition on observed traffic, tunes Capabilities from field feedback. It is the agent's *mise en place* — the work that sustains the quality of service without being visible to the user.

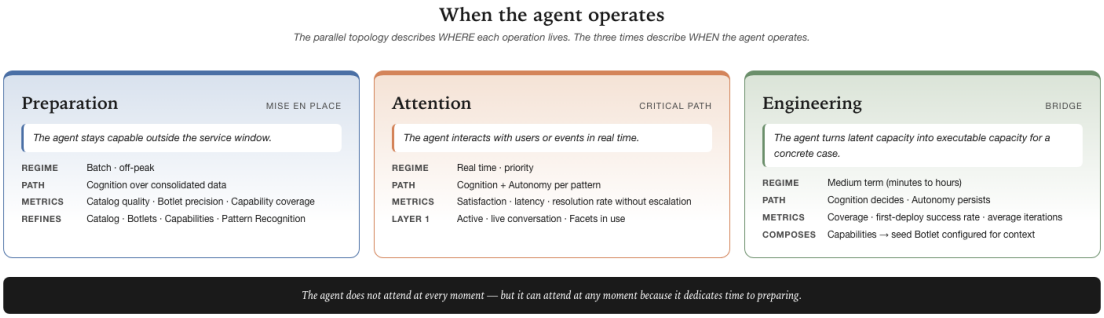


Figure 11: The agent's three times · Preparation · Attention · Engineering

Preparation operates predominantly on the Cognition Path (Layer 2) over consolidated data, not over the burst of the moment. It is typically **batch / off-peak**: it runs when effective operation does not demand all available cognition, or on separate cognitive infrastructure. Its metrics are about quality — how good the catalog is, how accurate the recent Botlets are, how complete the Capabilities are.

Attention

Attention is the time in which the agent **interacts with users or events in real time**. Layer 1 active, live conversation, execution of Botlets that sustain operation, escalations where appropriate. It is the critical path — where the organization feels the agent, where the SLA matters, where the cost of error materializes.

Attention operates over both paths (Cognition and Autonomy) according to the pattern, with priority, with bounded latency and high availability. Its metrics are operational — user satisfaction, response latency, resolution rate without escalation, mean time between escalations.

Engineering

Engineering is the bridge between Preparation and Attention: the time in which the agent **converts latent capacity into executable capacity for a concrete case**. It receives a request, identifies which Capabilities apply, configures a seed Botlet for the specific context, validates its execution over real data, deploys it to the corresponding environment, observes the result. It is configuration and orchestration work, not general reasoning nor pure service.

Engineering operates on a mix of paths: it uses cognition to decide composition but generates artifacts that persist in Autonomy. It is typically **medium term** — minutes to hours, not seconds — and has its own rhythm distinct from the rhythm of Attention. Its metrics are about coverage — what fraction of requests can be served with configured seed Botlets, what success rate they have on first deploy, how many iterations on average they require.

Implications for reasoning about the system

The distinction among the three times has three practical consequences for the operation of the AgencyDomain.

First, scheduling of cognitive capacity. Cognition is a costly and finite resource. The organization consciously chooses how much is allocated to each time: Attention demands bounded latency and high priority; Preparation tolerates batch and exploits demand valleys; Engineering occupies an intermediate band. Without this distinction, cognition is allocated by temporal proximity to the request and Preparation is relegated — the agent stops improving itself, its catalog ages.

Second, distinct metrics per time. A single "agent performance" dashboard lies: Preparation is measured by aggregate catalog quality and Attention is measured by operational satisfaction. Mixing them hides where the problem is when something goes wrong. The mature organization instruments the three times separately.

Third, availability model. A well-operated agent is **not 100% in Attention**. It needs Preparation windows. The promise "always-available agent" is better understood as "Attention always available" — Preparation operates behind it. This distinction is what allows offering service SLAs without cannibalizing the background work that sustains quality.

> *The agent does not attend at all times — but it can attend at any time because it devotes time to preparing itself.*

Required properties

Property	Level
Explicit recognition of the three times in operation	**MUST**
Separate metrics per time (Preparation, Attention, Engineering)	**MUST**
Reserved Preparation windows, not optional	**SHOULD**
Scheduling of cognitive capacity by time priority	**SHOULD**
Traceability in the log of which time executed which operation	**SHOULD**

Layer 1 — Interaction

Layer 1 is responsible for all communication between humans and the system. It is pure interface, with no business logic. The human who interacts with an agentive system never directly touches the other three layers — they only see Layer 1, and Layer 1 translates their intentions toward the layers that

execute. It sounds simple stated that way, but the design of Layer 1 contains almost all the decisions that determine whether the human will use the system frequently or abandon it after the first week.

The canonical modalities of Layer 1 are six, and a serious agentive system typically supports more than one. The **textual conversational** modality — direct chat with the agent — is the most visible and the one most contemporary commercial products implement first. It is an efficient modality for analytical or drafting tasks, where the human formulates their requests well. The **voice conversational** modality — virtual assistants, calls, audio-bots — is critical for use cases where the human has their hands busy or needs to interact while on the move. **Corporate channels** — Slack, Teams, WhatsApp, email — function as conversational surfaces when the human does not want to open a specific application to talk to the agent, but prefers the agent to appear where the human already is. The **programmatic API** enables external systems to invoke the agent without human intermediation — a critical pattern for cases where the agent is invoked by another system, not by a person.

The two least-discussed but structurally important modalities are the **generated GUI** and **passive signage**. Passive signage is surfaces that communicate information continuously without requiring human interaction — panels, operation dashboards, ambient displays. The human does not operate: they read. This modality is central for operational use cases where the agent must keep the human informed without waiting for the human to ask. The generated GUI deserves extended treatment because it is where the reading of the agentive paradigm is most easily confused. The section that follows develops it.

Three GUI regimes in the agentive Layer 1

A recurring — and wrong — reading of the agentive paradigm concludes that the Agentive World implies abandoning all graphical interface: if everything is conversation with the agent, GUIs disappear. That reading confuses two distinct things. What disappears **is not the GUI** — it is the GUI **pre-created by human teams in pre-agentive times**. The GUI continues to exist when operation requires it; what changes is **its mode of existence**: it goes from being a fixed template coded before use to being a **surface generated by cognition** according to the needs of each interaction.

The productive Layer 1 of the Agentive World distinguishes **three regimes of generation**:

1. Pure conversational. The agent responds in text or voice; sufficient when the information is sequential and the decision is flexible. A customer asking about their balance, a user asking to draft an email, an operator checking the status of a process — all cases where conversation is the correct modality. There is no generated graphical surface because none is needed.

2. GUI generated on-the-fly. The agent composes a graphical surface adapted to the immediate task: a view, a form, a panel, a dashboard. The GUI lives as long as the task lasts; the next time the human needs something similar, the agent can regenerate it differently according to context. It is the correct modality when the information is dense or multidimensional, when the decision demands visual comparison, or when the human must manipulate several elements simultaneously. It is what is usually understood as "dynamic GUI".

3. Persistent GUI generated as a Botlet. For repetitive operational roles — a cashier at peak hour, a kitchen panel, a cashier dashboard, an industrial monitoring screen — the agent generates a stable surface and consolidates it as a **facade Botlet** — a Botlet that lives in Layer 3 and exposes that surface with a stable contract in Layer 1. It is generated GUI that persists because the usage pattern is stable, the operational role is clear, and response speed is critical. It remains agentive: the agent can regenerate it when the environment changes (new products, new rules, new flow), exactly as any other Layer 3 Botlet

regenerates when its environment changes. The difference from the traditional GUI is that **no human UI/UX team designed it**: cognition generated it because the usage pattern justifies it.

The three modalities coexist in a mature agentive system. The distinction among them is not hierarchical — it is not that the persistent GUI is "better" than the conversational one. It is **fit to the usage pattern**: conversation when the case is new or flexible, on-the-fly GUI when the task is dense but occasional, persistent GUI when the role is operational and repetitive.

> *The GUI does not disappear in the Agentive World. What disappears is the pre-created GUI. Every GUI in an agentive Layer 1 is generated by cognition — some ephemeral, others stabilized as facade Botlets.*

The practical consequence of the distinction is operational. Without it, "agentive" is interpreted as "everything is chat" — operationally impractical for roles that need speed. A cashier at peak hour does not converse to ring up a sale; a cook does not chat with the ordering system; a plant operator does not ask the agent by voice to show the process status. With the distinction, those roles operate over **persistent GUIs generated as facade Botlets** — stable, fast, specialized surfaces that invoke Layer 3 Botlets directly (the `Layer 1 → Layer 3 → Layer 4` path that the parallel-topology section formalizes). The system remains entirely agentive; what changes is that cognition does not participate in every operational interaction — it does participate in the initial generation of the facade, it does when the facade needs to regenerate, it does when the operator escalates a new situation.

The **facade Botlets** connect naturally with the seed/emergent distinction of Chapter 5 §2: persistent GUIs typically correspond to **seed facade Botlets** — Layer 3 Botlets with their surface in Layer 1, generated by cognition at the design team's request as part of the initial product, just like the seed transactional Botlets of the same layer.

Composition of the surface · shell, view, operation

A non-trivial surface **is not a monolithic Botlet**. It is composition. Reading it this way makes explicit what is reused, what is specific, and where each piece lives within the architecture; treating it as a single block condemns the design of the productive Layer 1 to intuition and wastes reuse across surfaces.

The spec recognizes **three canonical roles** that compose a surface:

Surface Botlet (shell) — Layer 1. It is the **container**: layout, navigation among views, session lifecycle, shared state. The product-specific part. There is typically one shell per principal operational role — the floor POS shell, the cashier panel shell, the mobile executive dashboard shell. The shell is the least reusable: it encapsulates product identity.

View Botlet — Layer 1. A **screen or panel** within the surface. A surface has one or several views; those used in several shells are extracted as their own Botlets. The "shopping cart" view, the "order detail" view, the "shift summary" view. Views are **highly reusable** — the same "order detail" view can appear inside the POS shell and inside the cashier panel shell.

Operation Botlet — Layer 3. The **business execution** that the view invokes. It lives in Autonomy, not in Interaction. *"Charge a table", "print a kitchen ticket", "close a shift", "consolidate inventory"* — these are operations in Layer 3, not surfaces in Layer 1. An operation can be invoked from multiple views within multiple shells. Operations are **the most reusable asset** of the catalog.

The key distinction: **shell and view are surface (Layer 1); operation is execution (Layer 3)**. A surface is a composition of Layer 1 Botlets that orchestrate and invoke Layer 3 Botlets.

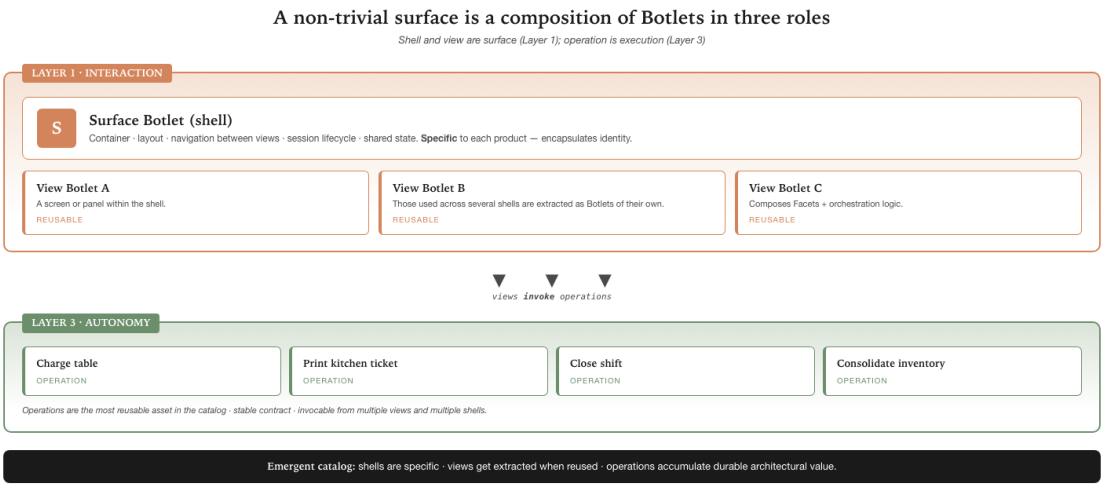


Figure 12: Composition of Layer 1 · shell · view · operation

Emergent catalog. This decomposition is a **prerequisite for reasoning about the catalog of reusable pieces**: operations accumulate in the catalog over time and form the most durable architectural asset; reusable views are extracted and catalogued; shells remain specific but their construction is accelerated because they assemble existing pieces. Without the explicit decomposition, everything is treated as an "application feature" and reuse is not exploited.

Multi-view Information Product · drill-through. An **Information Product (`PI`)** — the manifestation that an informational operation Botlet leaves on being consumed — is not necessarily a single piece: it can be composed of N named **views** connected by **drill-through**, the navigation-with-context that narrows — never widens — what the viewer can already see. The normed description of the `PI` — the multi-view composition, the data-anchored / no-bypass property, and its canonical example — lives in Chapter 5 §2, alongside the manifestation that begets it.

Facet · atomic primitive of Layer 1

So far Layer 1 has been described in terms of generation regimes (pure conversational, on-the-fly, persistent as a Botlet) and composition (shell, view, operation). What is missing is to name the **atomic unit** with which these surfaces are built: the **Facet** — an atomic reusable component of Layer 1: a catalog-picker, a calendar, a clickable map, a slider, a freehand drawing board. It is an **instrument**, not a process: the agent picks it up while thinking — in live conversation or inside a view Botlet — and drops it when it is done.

The Facet is not a Botlet: it has no fallback guarantee, no `95/4/1` cycle, no maturity phases — if it fails, the agent returns to conversation. Offering a Facet is, moreover, a cognitive act of the agent, not a

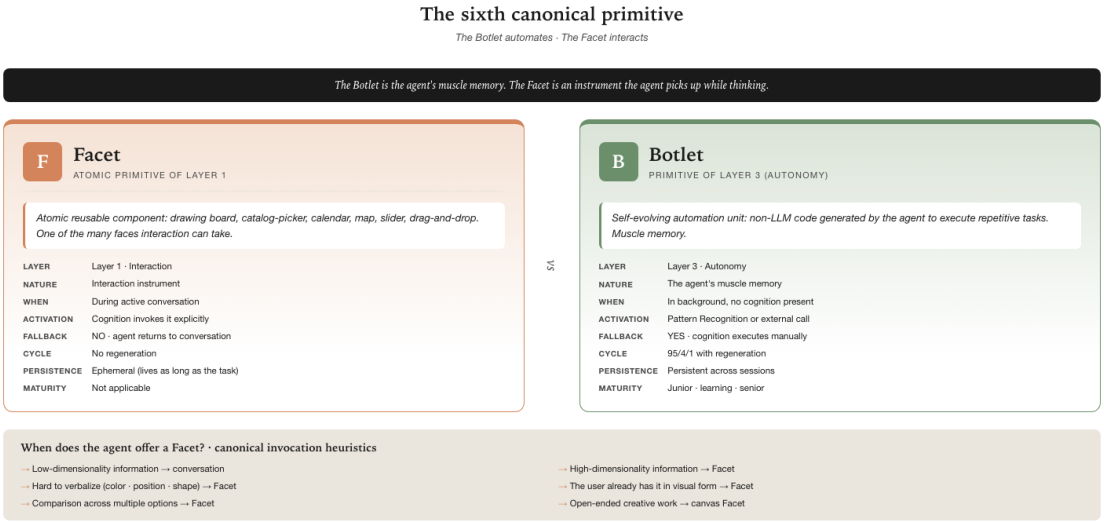


Figure 13: Facet vs Botlet · two primitives, two layers, two natures

pre-programmed feature: cognition estimates whether the information is obtained faster visually than verbally, and decides. The full canonical distinction, the two uses, and the heuristics of that decision live in Chapter 5 §6, where the Facet is formalized as a primitive.

> *If the human opens applications to do their work, we are not in the Layer 1 of the Agentive World.*

The statement gathers Satya Nadella's thesis from the BG2 podcast of December 2024, which we already cited in Chapter 1. It is an operative calibration exercise with an additional nuance under the three regimes we have just defined: the question is not whether there is a GUI or not, nor how pretty it is. The question is **who generated it**. If the GUI was pre-created by a UI/UX team in traditional application sprints, it is not agentive Layer 1. If the GUI was generated by cognition — ephemeral or persistent as a Botlet —, it is.

Three required properties distinguish a well-designed Layer 1 from a collection of ad hoc adapters. Being **channel-agnostic** means that the conversation logic does not depend on the medium: the same agent must manifest coherently in chat, voice, on-the-fly GUI, without the developer rewriting the logic for each channel. If the agent knows the customer's data and preferences, that information is the same regardless of whether the customer is speaking by voice from their car or by chat from their laptop. **Register adaptation** requires that the agent understand the channel's register — formal in corporate email, concise in chat, verbal in voice — without that adaptation living in conditional code. It is a property of cognition manifesting through Layer 1, not of Layer 1 itself. And **context persistence** guarantees that the conversation survives the change of channel: a human who begins by chat and continues by voice keeps the thread. Without this property, the system fragments the human experience

into channel silos, and the human perceives the "agent" as multiple disconnected agents — exactly the friction the agentive paradigm promises to eliminate.

Layer 2 — Cognition

Layer 2 is where the system thinks. It is the agent's brain — interpretation, reasoning, planning, application of specialized know-how, the decision to delegate. If Layer 1 is the agent's face, Layer 2 is what lies behind the face.

The canonical components of Layer 2 are five. The first is **multi-LLM**: cognition is not tied to a single model provider. Different providers, models, modalities — text, multimodal — and architectures — LLM, symbolic, hybrid — coexist under a common contract. The reason is operative before it is philosophical: the landscape of cognition providers evolves on the scale of months, and a system tied to a single provider accumulates debt every time that provider loses competitiveness against a new entrant. A well-designed multi-LLM system allows migrating between providers without rewriting the agent's logic.

The second component is **Capabilities** — units of modular, composable know-how, organized in a hierarchical tree. Cognition selects and applies Capabilities according to the task. Capabilities are codified professional know-how — accounting know-how for a financial agent, regulatory know-how for a legal agent, operative know-how for a support agent. We develop them in detail in Chapter 5. For now it suffices to retain that Layer 2 does not operate with monolithic knowledge — it operates by selecting modules of specialized know-how and combining them according to the case.

The third component is **Pattern Recognition** — detection of repetitive patterns in the agent's activity. The capacity is inspired by the neurobiological architecture of human memory; Chapter 5 §2 develops the parallel (Squire and Wixted) alongside the Botlet cycle. When the agent recognizes a repetitive pattern in the activity — the same task executing with variable frequency but stable structure —, it triggers the generation of a Botlet that automates that task without requiring additional cognition each time. Pattern Recognition is the entry to the Botlet cycle, which we develop in Chapter 5.

The fourth component is **Botlet generation** itself. Cognition decides when to delegate repetitive tasks to Layer 3 — where Botlets execute without invoking cognition. This decision is not trivial: a cognition that delegates too much loses flexibility when the environment changes; a cognition that delegates too little saturates its resources on tasks that traditional code executes better. The calibration of when to generate a Botlet is an emergent property of mature cognition.

The fifth component is the **reactive Assistant** — the agent operating in response-to-request mode. It waits for input from the human, responds, moves to the next turn. This mode is pure Layer 2 — cognition without autonomy, unlike the proactive mode that lives in Layer 3. The Assistant vs Autonomous Agent distinction is developed by Chapter 5 §5.

The specification further recognizes **two modes of access to cognition**: **Tokens** — the system centralizes credentials, billing, and policies; the natural mode for background Autonomous Agents — and **Subscription** — the user's assistant (Claude, ChatGPT, Copilot, Gemini) accesses under the user's own subscription, consuming no tokens from the system. The two coexist in the same system, the spec requires declaring which mode applies to which component, and confusing them is the most recurrent source of economic errors in an agentive deployment. The full formalization of both modes lives in Chapter 5 §1.

Under fixed Subscription plans, **Botlets are the architectural mechanism for extending autonomy**

without saturating the plan: an agent that executes its daily work via Botlets, reserving cognition for when the environment changes, can operate in continuous background without exhausting the quota. This makes the Botlet an economic lever, not just a technical optimization. Chapter 5 §2 develops this economics of the subscription.

A complementary property of Layer 2 is the **configurability of the cognition provider**. A conformant agentive architecture must allow the system to use a default provider — the one the AgencyDomain operator has chosen as its base economics — but admit its substitution by a provider brought by the end client. The industry uses the term **BYOModel** (*bring your own model*), analogous to the BYOK (*bring your own key*) or BYOIP (*bring your own IP*) pattern of the cloud field. The architectural consequence is that the agent's spec — its Capabilities, its tools, its Trust Infrastructure policies — must be **independent of the cognition runtime**. This enables multi-tenancy with heterogeneous cognition (different clients operating on the same substrate with different model providers) and respects the client's **cognitive sovereignty**: the organization decides who processes its prompts. The spec requires BYOModel as a **SHOULD** property: not every implementation supports it today, but architectures that aspire to operate in regulated markets will have to incorporate it within a foreseeable timeframe.

A final note on the evolution frontier of Layer 2: the specification admits **agnostic cognition** — symbolic, hybrid, multimodal. Contemporary implementation is predominantly LLM-centric, but the architecture does not require it. The formal extension of Layer 2 to other cognitive substrates — symbolic systems for formal problems, multimodal models that integrate sensor data, hybrid architectures that combine both — is a strategic horizon, not a short-term one. The importance for the architect is not to tie the design of the other layers to the assumption that Layer 2 will always be LLM. The architecture must survive the change.

A second note on the role of the **semantic layer** — a concept Chapter 2 already introduced with its figures. The quality of cognition depends critically on the quality of the information that feeds it. A serious agentive architecture contemplates the semantic layer as a necessary integration between Layer 2 and the Environment's data (Layer 4): without it, cognition operates over inconsistent representations of reality and produces answers that look coherent but fail at what matters.

Layer 3 — Autonomy

Layer 3 is where the agent lives. It is persistent life, continuous execution, action on its own initiative. Where the **Autonomous Agents** dwell — proactive, not reactive. Distinct from the Assistant that lives in Layer 2 and waits to be invoked.

The Autonomous Agent is distinguished from the Layer 2 Assistant by an operational difference: the Assistant responds when asked, with no persistent state nor Botlets of its own; the Autonomous Agent pursues objectives without continuous human input, maintains state, regenerates Botlets, and lives in the background. Chapter 5 §5 develops the distinction.

The canonical components of Layer 3 are six. **Proactive processing** is the heart of the layer: the agent does not wait for orders; it pursues objectives. **Asynchronous tasks** are operations that execute in the background without blocking any conversational thread. **Continuous monitoring** detects anomalies, events, thresholds that trigger action. The **Lets in execution** — the packaged units that inhabit the layer — are two sibling species: the **Botlets** — the agent's muscle memory operating, which the agent forms and owns (Chapter 5 §2), canonical cycle `95/4/1`: 95% normal execution, 4% change detected in the environment that makes the Botlet fail, 1% regeneration of the Botlet by cognition — and the

Agentlets — packaged routine judgment, units whose body invokes bounded inference for tasks recurrent in form but interpretive in every instance (Chapter 5 §7 formalizes them). The **Botler** is the framework runner that executes the Lets of both species — a piece invisible to the user and to the agent itself, a responsibility of the implementation. **Intra-AgencyDomain coordination** —via the `A2A` protocol— is communication among specialist components that live in the same computational space: coordination among specialist agents, dynamic delegation, exchange of results.

And the non-negotiable property that defines the layer: **fallback guarantee**. If a Botlet fails catastrophically, cognition executes the task manually. The process never stops. This guarantee distinguishes the agentive system conformant to this specification from any fragile "AI automation". An organization that delegates operation to agents must be able to trust that an isolated failure does not stop its business. Resilience is what makes that trust reasonable. Without a fallback guarantee, Botlets are fragile scripts disguised as innovation; with a fallback guarantee, they are operational pieces the organization can lean on.

Without Layer 3, the agent only reacts. With Layer 3, the agent can anticipate.

Layer 3 demands three strong properties that any implementation must satisfy. **Persistence between sessions** ensures that the agent's state survives disconnections, restarts, and migrations. An agent that loses its state when the server restarts is not an Autonomous Agent — it is an Assistant with a long-running process. **Execution isolation** ensures that the Lets — Botlets and Agentlets — run under sandboxing appropriate to the environment — containers, WASM, MicroVMs according to the security-versus-overhead trade-off. A Botlet generated by cognition is new code; it must operate under strict confinement before touching sensitive resources. **Structural resilience** ensures that no failure of an individual Botlet stops the agent's operation. One Botlet fails, another replaces it, cognition regenerates, and the system continues.

There is a recurring temptation, especially in teams coming from the traditional software world, to treat Layer 3 as "where the workflows run" — the agentive analog of a classic orchestrator like Airflow or Temporal. The analogy is misleading because traditional workflows are static: they are defined by code a human wrote, they execute the steps in the foreseen order, they fail when something departs from the flow. The agentive Layer 3 is **dynamic**: the Botlets it executes were generated by cognition, they regenerate when the environment changes, and the organization trusts that the whole operates coherently without any human having explicitly written the flow. A classic orchestrator is an executor of human instructions; Layer 3 is an executor of instructions that cognition itself generated — and that changes the entire governance model of the system.

The interface by which Cognition commands this layer is **internal** and lives **within the same AgencyDomain**: Layer 2 commands Layer 3 — Cognition operating its own muscle memory. The natural transport is `MCP` (Cognition is an LLM agent and `MCP` is the LLM↔tool protocol): the **Botler exposes `MCP` server(s)** and **Cognition is the client**. This **is not `A2A`** — `A2A` is the relation between AgencyDomains (agents), federation or external, not the internal operation of an agent over its own runtime. The development of this interface and of its operation API lives in the AgencyDomains specification (Chapter 5 §1) and in the Botlets chapter.

An additional structural property of Layer 3 — central for systems with multiple physical presence — is that it admits **geographic distribution within a single AgencyDomain**. A system that operates 7 restaurant locations, 50 bank branches, or 200 healthcare points of service does not need 7, 50, or 200 independent AgencyDomains — it needs a single AgencyDomain with **Layer 3 distributed** between a central Botler (orchestration, planning, reporting, global-decision Botlets) and N edge Botlers (local transactional Botlets with a local DB and an event queue toward the center), coordinated by intra-

AgencyDomain coordination —via the `A2A` protocol— between Botlers of the same AgencyDomain. This is not `A2A` federation between distinct AgencyDomains; it is **internal distribution of Layer 3** within a single AgencyDomain. The complete spec of the pattern lives in Chapter 5 §1.

Layer 4 — Access

Layer 4 is where cognition becomes real action upon the world. It is the agent's power of execution over systems, data, and external agents. The point where every decision of the agent must pass through governance before touching the world. If Layer 3 is where the agent lives, Layer 4 is where the agent acts.

The canonical components of Layer 4 are eight and we lay them out carefully. **Tool servers** are tools the agent can invoke to touch external systems — email, calendars, repositories, databases, ERPs, CRMs, public APIs, files. The contemporary canonical protocol is the **Model Context Protocol** (MCP), introduced by Anthropic in November 2024 and progressively adopted as an open industry standard. The rapid adoption of MCP — faster than almost any recent open protocol had achieved — reflects a real need: the field lacked a standard for connecting agents with tools, and all serious actors understood that fragmentation was a problem rather than an advantage. **Connectors** are the know-how to access source systems — the legacy API of the pre-agentive world brought into Layer 4 as an access capability with execution power, not as cognitive know-how (that lives in Layer 2 as a Capability). It is the materialization in the architecture of the destiny that the Bounded Concerns Architecture assigns to the API cell: it persists, intensifies, and is repositioned as a Connector.

`A2A` between AgencyDomains enables interaction between agents that live in distinct computational spaces — federation between AgencyDomains of different organizations, integration with agents of external providers. The **Trust Infrastructure exercised at the point of action** is probably the most critical component of Layer 4: governance, audit, validation, resilience, and transparency are exercised here, where the agent is about to act. The detailed description of Trust Infrastructure comes later in this chapter and is developed in detail in Chapter 5. The **CRUDLEX permissions** are the canonical model of granular control: Create, Read, Update, Delete, List, Execute, applicable by user, agent, or context. The complete operational description of CRUDLEX lives in Chapter 8. **Human approval** is optional, configurable by policy, for critical operations — sending external email, financial transfer, irreversible modification. **Intelligent routing and semantic cache** optimize the cost and latency of invocations. The **immutable append-only log** is the auditable record of every action of the agent, with complete lineage for later reconstruction.

> *Layer 4 turns cognition into real action with governance.*

The required properties of Layer 4 are four and all are mandatory for an enterprise production system. The first is **non-repudiation**: every action is recorded with the agent's identity, context, and result. When the system executes an operation, it must afterward be possible to reconstruct which agent executed it, on behalf of which human or organization, with what authorization, and with what result. Without non-repudiation, there is no possible audit and no regulatory defense. The second is **reversibility where applicable**: critical operations have a rollback or compensation mechanism. Not all operations are reversible — a sent email or an executed transfer typically are not —, but when reversibility is technically possible, it must be designed from the start. The third is **policy before execution**: no action executes without having passed through the governance plane. The policy is evaluated before, not after. A system that records decisions after making them and then waits for the human to correct them has a governance model that arrives too late. The fourth is **uniform observability**: every invo-

cation produces traces, metrics, and events in the same format. Without uniformity, observability data is operationally unconsumable.

Layer 4 is where most agentive projects fail, according to the field data of Chapter 2. The structural reason: teams coming from the traditional software world treat Layer 4 as an "API gateway with permissions", and it is not that. A traditional API gateway operates over human requests — the human makes the request, the gateway validates permissions, the system executes. The agentive Layer 4 operates over requests generated by agents acting autonomously. Validation cannot assume that a human supervised the request beforehand; it must assume that the agent decided on its own, and that the human will not see it until the audit log. This demands levels of validation, recording, and approval that traditional systems did not need.

Trust Infrastructure — the cross-cutting axis

Trust Infrastructure is not an additional layer. It is **cross-cutting to all four**. Without Trust Infrastructure, agent pilots die on the way to enterprise production — and "die" is not a metaphor; it is what produces the wave of cancellations Chapter 2 documents. Trust Infrastructure is the difference between experimenting and operating.

Five pillars constitute Trust Infrastructure. **Governance** defines configurable policies, CRUDLEX permissions, human approval for critical operations, AI registry. It is exercised principally in Layer 4, cross-cuttingly in the rest. **Audit** maintains an immutable append-only log, a trace of every action, lineage of decisions, identity tagging per action. It is exercised in Layer 4 and cross-cuttingly. **Validation** detects hallucinations, validates responses, prevents prompt injection, executes DLP and tokenization. It is exercised in Layer 2, in Layer 4 and — over the bounded inference of Agentlets — in Layer 3. **Resilience** guarantees fallback, handles errors, sandboxes the Lets. It is exercised in Layer 3 and cross-cuttingly. **Transparency** delivers complete observability, metrics, end-to-end traces, proactive alerts, governance dashboards. It is cross-cutting to all four layers.

The detailed description of each pillar — its canonical mechanisms, its required properties, its operationalization into concrete policies — lives in Chapter 5 §4 and in Chapter 8 (which operationalizes the five pillars into policies, the complete CRUDLEX model, the format of the append-only log, human-approval protocols). In this chapter it suffices to retain the fundamental architectural property: Trust Infrastructure is not added after the agent works — it is designed from the start, in the architecture itself.

The urgency of Trust Infrastructure is no longer only architectural — it is regulatory: the state and international frameworks that demand it are developed in Chapter 5 §4. The question is no longer whether regulators will require trust infrastructure — it is whether the organization can demonstrate it auditably when asked.

The state of the field with respect to governance is documented with figures in Chapter 2: most of the organizations that operate agents today are not prepared to defend what their agents do. What matters here is the architectural consequence of that diagnosis: if governance is not designed from the start, it is not built afterward.

The governing principle — Agent First

> *Faced with any dilemma, the agent's experience is prioritized over the human's. The agent is the primary user; the human's needs are resolved in a management layer without degrading what the agent sees and can do.*

The Agent First principle is a design-governance rule that orders any architectural dilemma. It inverts the logic of traditional software design, where the human is always the primary user and everything is designed so they can operate it. In the Agentive Architecture, the APIs, the schemas, the errors, and the control flows are designed **first** for the agent's consumption. The human surface — settings, dashboards, administrative interfaces — is secondary and does not condition the architectural decisions.

The inversion is not rhetorical — it has concrete operative implications. Any new capability of the system is specified first as a **tool with a declarative JSON schema**, then as a GUI if applicable. **Errors are structured and actionable**: codes and messages designed so the agent decides the next step, not so the human "reads the log". **Idempotency where applicable**: the agent's retries must be safe without requiring defensive logic in the caller. **Uniform pagination and filters** across tools: a consistent format, predictable for the agent. **Machine-readable documentation**: the public documentation is consumable by the agent as context, not only legible by humans.

Agent First is a **governance rule**. Any design dilemma that violates it requires explicit, documented justification. When the team faces a decision where "this would be easier for the human operator, but makes it harder for the agent", the default answer is to prioritize the agent. The exception must be argued and recorded. Without an explicit governance rule, the inertia of traditional software pushes all decisions toward the human side, and the system ends up being just another application with agentive makeup.

The structural reason behind the principle: in the Agentive World, the frequency with which the system interacts with agents is **orders of magnitude greater** than the frequency with which it interacts with humans. An agentive system in production typically has millions of agent invocations against hundreds of human operations. Optimizing for the minority case — the human — degrades the majority case — the agent — exponentially. Agent First is recognition of that asymmetry.

The evolution of agents

The architecture admits three evolutionary phases in the sophistication of the agent. Different organizations find themselves in different phases, and the conversation about architecture changes significantly according to the current phase.

Phase one is the **specialized agents**: one agent per domain. Each with its specific Capabilities, its clear role, its limited surface. The financial agent operates over cashflow; the customer-service agent operates over tickets; the operations agent operates over inventory. This is the current phase of the market. Most agents in production at the start of 2026 are phase-one specialized agents. The reason is practical: it is the easiest phase to govern — the scope is narrow, the risk is contained, the use case is clear.

Phase two is the **orchestrator agents**: one agent coordinates multiple specialists. Dynamic delegation according to the task. The orchestrator agent does not solve the problem directly — it decomposes the problem, identifies which specialists can solve each part, delegates, integrates the results. This is a transition phase that some leading actors are already exploring, especially in cases where tasks cross domains — a customer-service case that requires a query to finance, validation with operations, and

Figure 14: The three evolutionary phases of agents

a response to the end customer, for example. Phase two demands a robust Layer 3 so the specialized agents can coordinate intra-AgencyDomain, via the `A2A` protocol.

Phase three is the **multi-specialist agents**: deep multi-domain expertise in a single agent. The future phase — it requires a maturity of Capabilities and Pattern Recognition that the industry has not yet reached. A multi-specialist agent does not decompose the problem by delegating — it solves it directly by integrating know-how from multiple domains. The difference from the orchestrator is ontological: the orchestrator is a coordinator of specialists; the multi-specialist is a deep specialist in many things at once. The Capabilities in the multi-specialist case form a much broader and deeper tree, and cognition must be able to navigate it efficiently.

The architecture is **the same across the three phases**. What changes is the complexity of cognition and the depth of the Capabilities tree. A well-designed system in phase one can evolve to phase two without a rewrite — by adding more specialized agents to the computational space and enabling intra-AgencyDomain coordination. A well-designed system in phase two can evolve to phase three when Capabilities mature — by fusing trees of specialized know-how into broader trees. This capacity for evolution without rewriting is an emergent property of the four-layer design. A poorly designed system — with fused layers — needs a complete rewrite to go from phase one to phase two, and that is typically when projects collapse.

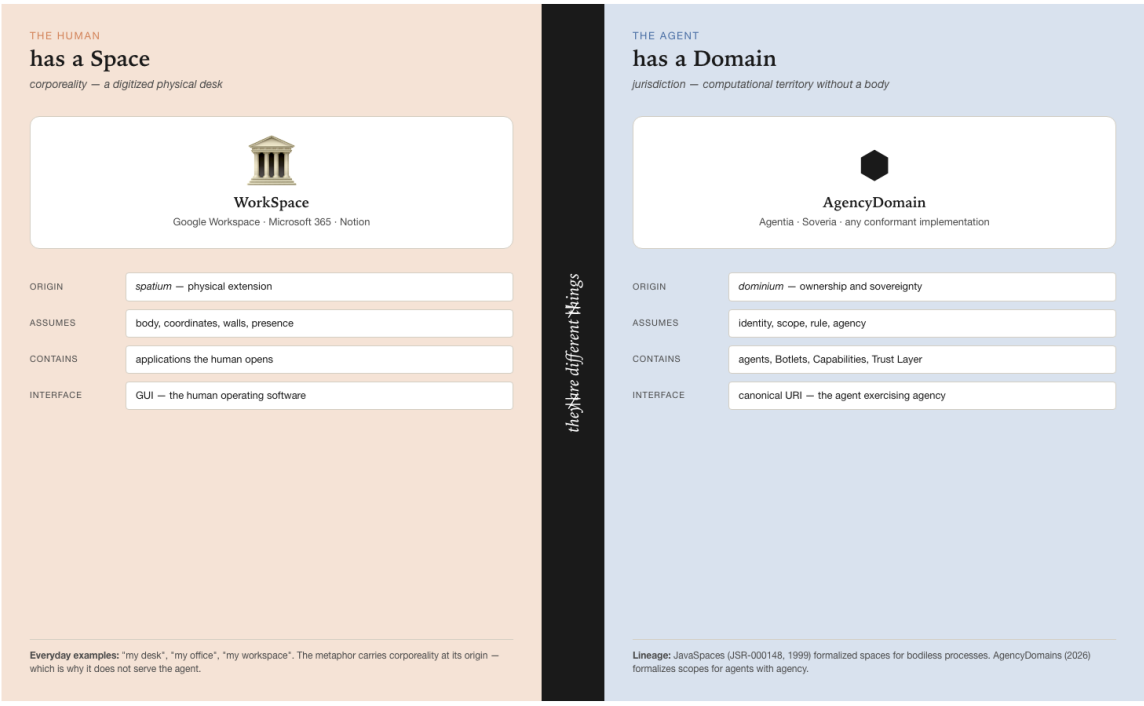


Figure 15: Space ≠ Domain · the ontological difference

The computational scope — AgencyDomains

Foundational premise — Space ≠ Domain

Where the human has a **Space** —corporeality inherited from the physical desktop, extended by the industry to **WorkSpace**—, the agent has no body: it exercises agency over a **Domain**, a scope of computational jurisdiction where its identity rules, its Capabilities apply, and its Botlets run. Chapter 5 §1 develops the complete derivation of this premise.

> *Where the human has a Space (WorkSpace), the agent has a Domain (AgencyDomain).*

The AgencyDomain as a formal construct

The architecture materializes in a formal construct: the **AgencyDomain** — a computational scope where autonomous agents dwell. It defines how agentive environments ought to be built — layers, cycles, primitives, interfaces — without prescribing a specific implementation; its historical parallel (JavaSpaces, the JSR-000148 spec of 1999) and the full derivation of the Space ≠ Domain premise live in Chapter 5 §1.

The formal specification of AgencyDomains lives in its dedicated document, which is the first section of Chapter 5. In this chapter it suffices to retain that the Agentive Architecture, seen as a concrete technical construct, is instantiated in AgencyDomains. When we speak of “the agentive system”, we refer to an instance of an AgencyDomain that materializes the four layers, exercises Trust Infrastructure, and respects the Agent First principle.

The specification covers aspects such as the identity and addressing model of agents and Botlets, the agent's lifecycle within the scope, intra-AgencyDomain coordination and `A2A` between AgencyDomains (both via the `A2A` protocol), federation between AgencyDomains (how two distinct scopes collaborate), and the tenancy and isolation model. All these details are developed by Chapter 5 §1.

Reference implementations

This architecture admits multiple implementations. The first coordinated implementation is the ultraBASE portfolio, where the responsibility for the four layers materializes through cooperating products — without each layer being exclusively assigned to one product. Each product contributes one or more of the agent's behaviors; the integrated stack composes them so the agent exhibits all four.

Other actors who adopt this architecture will have their own implementations — each valid insofar as it respects the four layers, exercises Trust Infrastructure, and honors the Agent First principle.

The book deliberately avoids describing the specific implementation of ultraBASE within its chapters, so as not to confuse the formal architecture with its particular implementation. The Epilogue, in "What is NOT in this book", develops why that separation preserves the claim to an industry standard.

Evolution frontier

The architecture admits legitimate extension over time along three technical horizons: **non-LLM cognition** —symbolic, hybrid, multimodal— which the Layer 2 frontier already anticipated above; **federation between AgencyDomains** —the `A2A` between non-related scopes, today an emergent capability and not a consolidated spec—; and the **Carbon World** —connecting Layer 4 to the physical world (IoT, industrial systems, manufacturing), link eleven of the value chain that Chapter 6 §3 develops—. The Epilogue develops the four living frontiers of the architecture, these three technical ones plus the institutional one.

The three vectors define the platform's innovation frontier. All three are sustained by the market projections documented in Chapter 2: mass penetration of agents in enterprise applications toward 2026, significant autonomous decision-making toward 2028, collective capital betting an entire decade on the agentive direction. The question this architecture answers — *how these systems are built with discipline* — only becomes more urgent with each passing quarter.

The four layers are the architectural answer to the paradigm. But the layers do not stand on their own — they need reusable pieces to populate them so an implementer can build against them with discipline. Chapter 5 delivers those pieces — eight canonical technical primitives that constitute the constructive vocabulary of a conformant agentive system: **AgencyDomain** as computational space, **Botlet** as the agent's muscle memory, **proto-Botlet** as its pre-forged piece, **Agentlet** as packaged routine judgment, **Capability** as the tree of cognitive know-how, **Trust Infrastructure** as the trust infrastructure, the **Assistant vs Autonomous Agent** distinction as the operative axis, and the **Facet** as the atomic unit of Layer 1. Whoever completes the two chapters holds the set of formal constructs with which the agentive category can be reasoned about and built.

A note on the numbering of the primitives: throughout the book the Facet is labeled the *sixth canonical primitive*, the proto-Botlet the *seventh canonical primitive* and the Agentlet the *eighth canonical primitive*. Those ordinals indicate the **order in which each primitive was incorporated into the canon**, not

their position in the enumeration above, where the proto-Botlet appears alongside the Botlet as its pre-forged piece and the Agentlet alongside both as its sibling species.

Visual summary

The four layers in parallel topology, with their principal components, the cross-cutting infrastructure, and the governing principle:

Layer	Role	Principal components
1 · Interaction	where the human communicates with the system	textual conversational · voice conversational · channels · direct API · generated GUI (on-the-fly · persistent as facade Botlet) · signage
2 · Cognition (slow · costly · decisive path)	where the system thinks	multi-LLM · Capabilities · Pattern Recognition · Botlet generation · reactive Assistant
3 · Autonomy (fast · cheap · repetitive path)	where the system lives with persistence	Botlets and Agentlets in execution · central + edge Botler · asynchronous tasks · monitoring · fallback guarantee
4 · Access	where the system acts upon the real world	tools (MCP) · Connectors (cloud · edge · hybrid) · A2A between AgencyDomains · CRUDLEX · human approval · append-only log

Trust Infrastructure is cross-cutting to the four layers (Governance · Audit · Validation · Resilience · Transparency). Layers 2 and 3 are **parallel paths** between Layer 1 and Layer 4 — not stages in series —, and the governing principle is **Agent First**.

Chapter 5 · Primitives

The four layers of Chapter 4 are the architectural answer to the paradigm, but they do not stand on their own: they need reusable pieces to populate them. This chapter delivers those pieces — the **canonical technical primitives** of the Agentive World. Each section formalizes one: the **AgencyDomain** (§1), the **Botlet** with its **proto-Botlet** (§2), the **Capability** (§3), the **Trust Infrastructure** (§4), the **Assistant vs Autonomous Agent** distinction (§5), the **Facet** (§6), and the **Agentlet** (§7).

AgencyDomains

Chapter 4 introduced the notion of the AgencyDomain as the formal construction where the Agentive Architecture materializes. This section develops that notion with the detail of a specification. What follows is the closest thing to a *normative spec* that this book delivers — a document an implementer can take and build from, knowing what is mandatory and what is optional, what properties it must satisfy and what decisions it may make freely.

Foundational premise — Space ≠ Domain

Words carry corporeality. **Space** is born describing physical extension: office, desk, home, city. When a human says *"my space"*, they invoke place — coordinates, walls, presence. The enterprise software industry extended the word to **WorkSpace** — Google Workspace, Microsoft 365, Notion — to name the collection of solutions that digitize what the person does at their physical desk: reading email, scheduling meetings, writing documents, filing. WorkSpace is the digital prosthesis of the human Space; both terms carry the same corporeality of origin.

The agent has no body. It does not open applications. It does not work on a desktop. It does not dwell in a physical space nor in any metaphor of one: it exercises agency over a **scope of computational jurisdiction** — digital territory where its identity rules, its Capabilities apply, and its Botlets run. The Latin word that names exactly that is **Domain** (`dominium`: scope of belonging and sovereignty).

Where the human has Space (WorkSpace), the agent has Domain (AgencyDomain).

The historical parallel — JSR-000148

The historical parallel holds: just as **JavaSpaces** (JSR-000148, 1999) formalized distributed spaces for Java systems without tying the implementation to a vendor, this specification formalizes **AgencyDomains** for agentive systems with the same agnosticism. Multiple implementations arose over JavaSpaces — GigaSpaces, Blitz, others —, all mutually compatible because they respected the common contract. The spec outlived the implementations; the implementations evolved without the spec needing constant rewriting. That is the pattern **AgencyDomains** seeks to replicate for the agentive field. This

book proposes the spec; implementations that comply with it may call themselves **conformant AgencyDomain**.

The difference in name from its predecessor is not a rupture — it is precision. A Java *Space* was computational space for bodiless processes; an Agency *Domain* is a scope of jurisdiction for agents with agency. What in 1999 was named "space" is better named in 2026 as "domain". The term **AgencyDomain** carries meaning in each word. *Agency* — agency, in its philosophical sense of the capacity to act — denotes that the scope is inhabited by entities with their own initiative, not by passive processes. *Domain* — `dominium`: scope of belonging and sovereignty — denotes territory where the agent's identity rules, not an ephemeral process. The union of the two words names exactly what the spec defines: a scope where the agent exercises agency.

> Terminological note: in commercial lore (brand, sales, customer communication) the short form `Domain` replaces `AgencyDomain`. It is the Apple iCloud / CloudKit, Stripe Connect / Account pattern: short brand + long technical name. The two forms are interchangeable; each record picks its own.

Definition

An **AgencyDomain** is a computational space with its own identity where autonomous agents and running Botlets dwell, where the **Capabilities** that give them their know-how are hosted and executed, and where the resources that sustain them live — cognition, tools, persistent storage. It constitutes the **minimal unit of deployment** of a productive agentive system.

The Capability is not a support resource: it is cognitive know-how, a first-order inhabitant of the space alongside the agent and the Botlet. The relation between the two primitives is direct — an AgencyDomain **hosts and executes** Capabilities; a Capability **runs in** a host AgencyDomain.

The spec defines how those spaces must be built — layers, identity, life cycles, communication protocols — without prescribing a specific implementation. Multiple implementations are admissible as long as they respect the contract. One implementation may use Kubernetes for containerization; another may use isolated microVMs; another may use native processes on a single machine. All three are valid if they satisfy the fundamental properties the spec requires.

> *Where an agent lives, that place is an AgencyDomain. Where there is no AgencyDomain, there is no life of the agent — there is model invocation.*

The quote above distinguishes the AgencyDomain conformant to this specification from any "endpoint that invokes an LLM". An endpoint that invokes an LLM responds to requests; an AgencyDomain **hosts agents that live**. The difference is structural, not one of degree. A system without persistent state, without its own Botlets, without the capacity for proactive operation, is not an AgencyDomain — it is a service. It can be a useful service, but it does not satisfy what the agentive field requires.

The minimal unit matters. An AgencyDomain is not a subcomponent of something larger — it is the atomic unit of deployment. A larger agentive system is a collection of cooperating AgencyDomains, possibly under a single governance or federated across distinct owners, but each one preserves its own identity and its fundamental properties.

Fundamental properties

Every implementation that intends to call itself an AgencyDomain conformant to this spec must satisfy five fundamental properties. Satisfaction is not optional — a system that does not meet them is not an

AgencyDomain, but something else with another name.

The first property is **own identity**. Each AgencyDomain has a unique identity — a canonical URI — that distinguishes it on any network. The identity survives the restart of the space, migration across infrastructures, and a change of implementation. If an AgencyDomain migrates from one cloud provider to another, its identity does not change: the agents that inhabit it, the humans who interact with it, the other AgencyDomains that invoke it, all continue to recognize it as the same space. The identity is stable, not ephemeral.

The second property is **materialization of the four layers**. An AgencyDomain materializes the four layers of the Agentive Architecture — Interaction, Cognition, Autonomy, Access — and exercises cross-cutting Trust Infrastructure. The materialization may be distributed technically — the layers may live in distinct infrastructures, on distinct physical servers, even across distinct cloud providers —, but responsibility for all four rests with the space. There is no conformant AgencyDomain that delivers only three layers, or that delegates some layer to an external system without assuming responsibility for it. Completeness is non-negotiable.

The third property is **persistence**. The state of the AgencyDomain — active agents, running Botlets, capabilities data, audit logs — survives disconnections, restarts, and migrations. An AgencyDomain is not an ephemeral process; it is a persistent entity. If the system restarts for maintenance, the agents that were active continue where they were. If the connection to an external service drops temporarily, the agent waits and resumes. Persistence is what makes the AgencyDomain a **place** and not a process.

The fourth property is **isolation**. Each AgencyDomain has an explicit boundary. Internal resources — compute, memory, data — are not accessible from outside except through defined interfaces. Communication with the exterior passes through Layer 4 (Access) and is recorded. Isolation is not only security — it is **fault containment**: an AgencyDomain that goes down does not affect other AgencyDomains that share infrastructure, because the boundary contains the failure. The implementation models that offer isolation vary — from strong sandboxing via MicroVMs to lighter isolation via Kubernetes namespaces —, but the explicit boundary always exists.

The fifth property is **addressability**. Both the AgencyDomain and the agents and Botlets that inhabit it are addressable via predictable URLs. The canonical syntax the spec adopts is:

```
{domain}/                          → the space itself
{domain}/agents/{agent}            → an agent that lives in it
{domain}/agents/{agent}/botlets/{botlet}   → a specific Botlet
{domain}/tools/{tool}              → a tool exposed via Layer 4
```

Addressability matters for two operational reasons. First, it is the basis of A2A communication — an agent that needs to invoke another agent does so through its canonical URL, with no need for ad hoc discovery. Second, it is the basis of MEO (Model Engine Optimization — the SEO→MEO shift Chapter 6 §1 develops): the frontier models that learn to reference AgencyDomains do so through predictable URLs that appear in their training corpus. Chaotic or unstable URLs make the AgencyDomain invisible to models that were not trained with its specific map.

Canonical data model

The internal structure of a conformant AgencyDomain follows a canonical model that the spec defines with precision (figure above).

Each component of the model has its specific role. **Identity** maintains the information that identifies

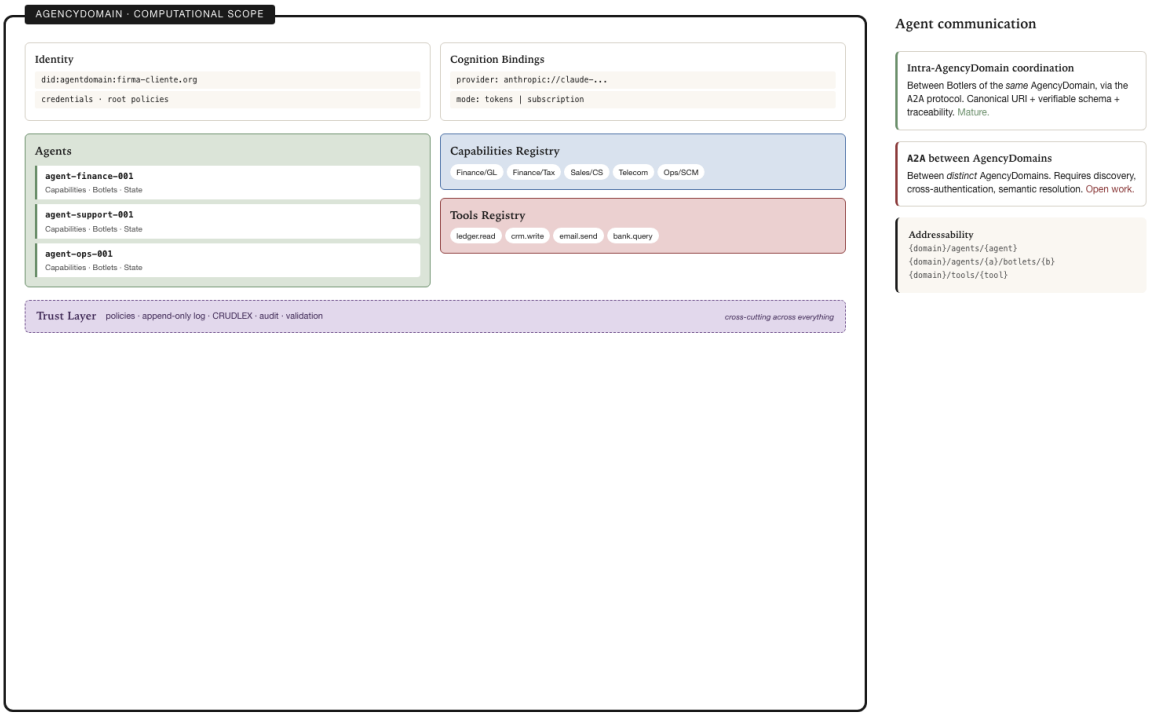


Figure 16: Anatomy of the AgencyDomain

the space to the world: its canonical URI, the credentials with which it authenticates to external systems, the root policies that no agent may contravene. **Agents** is the collection of agents that live in the space, each with its assigned Capabilities, its running Botlets, and its persistent state. **Capabilities Registry** is the tree of capabilities available to the agents of the space — shared know-how that agents can invoke according to their role. **Tools Registry** is the collection of tools that Layer 4 exposes outward — the interface through which the AgencyDomain touches external systems. **Trust Layer** exercises cross-cutting governance and audit — policies, append-only log, validation mechanisms. **Cognition Bindings** are the bindings to the cognitive resource — which model provider the AgencyDomain invokes, under what credentials, with what usage policies.

The notion of **Account** is a commercial concept superimposed on the technical model. An Account may own multiple AgencyDomains. The spec treats the Account as an opaque entity; each implementation defines its specific semantics — a client company, a federated organization, an individual user. The distinction between AgencyDomain (technical) and Account (commercial) matters because it allows the technical model to evolve without the commercial model needing a rewrite each time.

The regime model

One aspect that significantly distinguishes the AgencyDomains spec from more limited agentive solutions is its recognition of three possible **regimes** of an AgencyDomain, analogous to the regimes of cloud computing. The three regimes are technically equivalent in their internal structure; what changes between them is the **access boundary**, not the architecture.

The **private regime** corresponds to an AgencyDomain where the space and all its components live

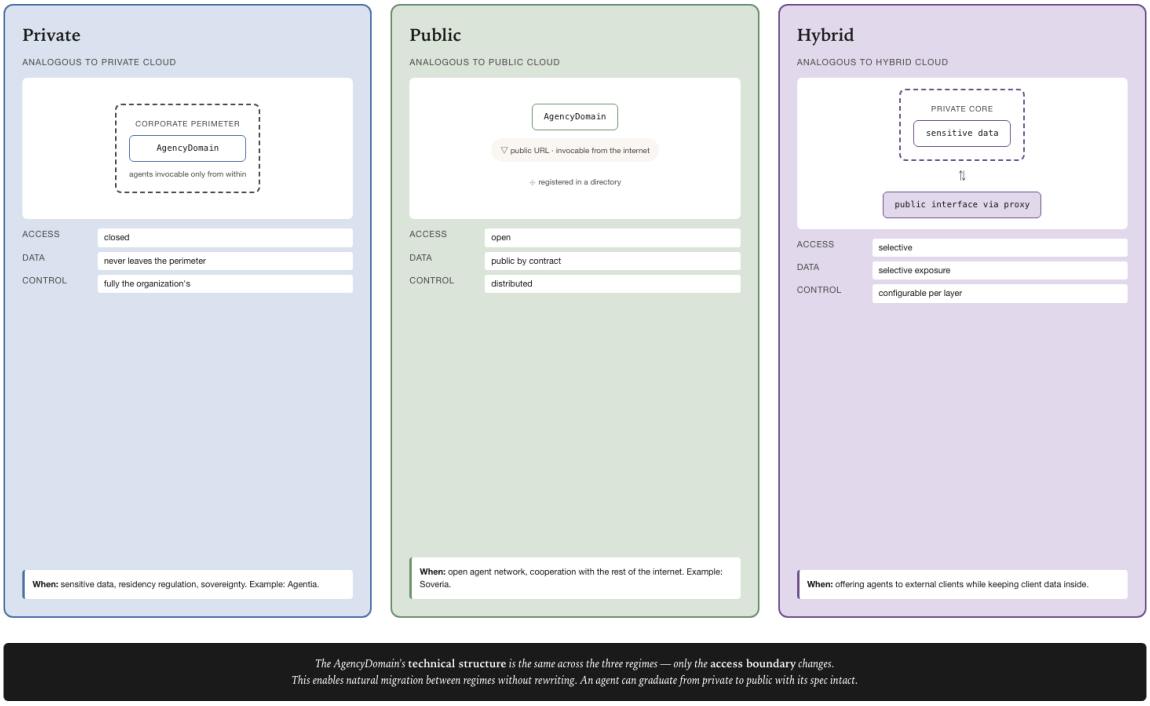


Figure 17: The three regimes · private · public · hybrid

within a perimeter controlled by an organization. There is no public access. The agents of the space are invocable only from within the organization. The data of the space do not leave the perimeter. It is the conceptual analog of the Private Cloud — the organization has total control of its resources, pays for that control in terms of operation but gains in terms of sovereignty. The private regime is typical for cases where the organization operates sensitive data or complies with regulation that demands local residency.

The **public regime** corresponds to a publicly accessible AgencyDomain. Agents, Botlets, and tools are invocable from outside the perimeter. The AgencyDomain has a public URL and the agents that inhabit it are registered in a directory that any external system can query. It is the conceptual analog of the Public Cloud — maximum accessibility, maximum exposure, a different operating model. The public regime is where the network of agents cooperates openly with the rest of the internet.

The **hybrid regime** is a combination of the two preceding ones. A hybrid AgencyDomain has components in a private perimeter and components accessible publicly via proxy. Sensitive data remain private, but the interface that exposes capabilities to the exterior is available publicly. It is the conceptual analog of the Hybrid Cloud — the organization chooses what to expose and what to retain, balancing sovereignty and accessibility. The hybrid regime is typical for organizations that need to offer public agents to their customers but want to keep customer data within the corporate perimeter.

What is critical about this regime architecture — and it is a strong property of the spec — is that **the technical structure of the AgencyDomain is the same across all three regimes**. An agent operating in a private AgencyDomain is technically equivalent to one operating in a public one; what changes is the regime, not the capability. A Botlet executing in private can move that same code to a public regime

without rewriting. This structural uniformity enables **natural migration** between regimes — an agent can graduate from private to public or vice versa without changing its internal logic. The organization governs the regime; the agent's logic never notices.

This natural-migration property is structurally important because it decouples the architectural decision (how the agent is built) from the commercial decision (in which regime it operates). An organization can begin building agents in the private regime while it validates their usefulness, and migrate them to the public or hybrid regime when maturity justifies it. The initial architectural investment is not lost in the transition.

To fix the idea with concrete instances as of May 2026 — both from the house portfolio; the general frame for implementations lives at the close of this spec and in Chapter 9 —: Agentia operates AgencyDomains in the private regime for firms that keep their agents within the corporate perimeter; Soveria operates AgencyDomains in the public regime where enabled agents are hosted with a public identity and a canonical URL; the same agent can graduate from the first to the second without a technical rewrite, keeping the agent's spec intact and moving only the regime.

Cognition access models

The spec recognizes two coexisting modes by which the components of the AgencyDomain access the cognitive resource (Layer 2). The two modes coexist because they solve distinct problems, and a serious AgencyDomain typically operates both simultaneously for distinct components.

The first mode is **Tokens**. The flow is: AgencyDomain → cognitive resource, centralized and billed to the space. The AgencyDomain centralizes credentials, billing, and policies. It provides cognitive access to all its active components. This mode applies when agents must operate in the background without user intervention, when the organization wants central control over consumption and costs, or when multiple agents share a single cognition provider. The organization that operates Autonomous Agents in the background — agents that monitor continuously, respond to events, execute asynchronous tasks — needs Tokens, because there is no human available whose individual subscription would subsidize the invocations.

The second mode is **Subscription**. The flow is: user's Assistant → cognitive resource, via the user's own subscription. The assistant the user interacts with — Claude, ChatGPT, Copilot, Gemini — accesses the cognitive resource directly under the user's subscription. The AgencyDomain consumes no tokens from the resource. This mode applies when the user already has an active subscription to the cognition provider, when the AgencyDomain exposes tools and data to the user's assistant without centralizing cognition, or when the AgencyDomain's operating economics favor minimizing inference costs. The organization that adopts ultraPRO — an implementation from the house portfolio (Chapter 9): the secure integrator between the user's agent and corporate systems — typically operates in Subscription mode, because users bring their own subscriptions to the cognition providers.

Both modes coexist in mature systems. A single AgencyDomain may operate user Assistants (Subscription mode) and background Autonomous Agents (Tokens mode), simultaneously. The spec requires the AgencyDomain to **explicitly declare** which mode applies to which component. The explicit declaration prevents the most recurrent source of economic errors in agentive systems: accidental confusion between modes, where an Autonomous Agent that should operate on Tokens ends up billing against the user's subscription and exhausts it in hours, or where an Assistant that should operate on Subscription ends up billing against the AgencyDomain and consumes tokens it should not.

The role of Botlets in the cognitive economy deserves particular emphasis. In fixed-Subscription plans,

Botlets are the mechanism for achieving autonomy without additional cost: the Botlets' `95/4/1` cycle is the economic basis of autonomy under subscription. The full development of this economy lives in §2.

Agent life cycle

An AgencyDomain conformant to the spec manages the complete life cycle of each agent that inhabits it. The cycle has six canonical phases, and each transition between phases is recorded in the Trust Layer's append-only log.

The **provisioning** phase is where the AgencyDomain creates the agent. It assigns identity, associates the initial Capabilities the agent may invoke, registers the agent in the space. The agent is born, in system terms, when this phase completes successfully. If the phase fails — by credential error, by name conflict, by quota restrictions —, the agent never comes to exist.

The **bootstrap** phase is where the agent enters operation. It loads its persistent state if it exists — if the agent had been hibernated or restarted, it recovers its prior context. It establishes bindings with the cognition and the tools it will use. It verifies that its Capabilities are available. After bootstrap, the agent is ready to respond or to operate proactively, according to its mode.

The **reactive operation** phase corresponds to the agent operating in Assistant mode. Layer 2 active. The agent responds to human requests: each request arrives, the agent processes it by invoking cognition and possibly Capabilities, returns a response. Between requests, the agent is passive — it consumes no active compute, executes nothing. This phase is the most frequent in systems that operate primarily with Assistants.

The **proactive operation** phase corresponds to the agent operating in Autonomous Agent mode. Layer 3 active. The agent pursues goals in the background, monitors events, executes Botlets when appropriate, escalates to the human when thresholds demand it. Pattern Recognition generates and maintains Botlets as the agent identifies repetitive patterns. This phase is where the model *"intelligence goes to people and acts on their behalf"* materializes — the agent is active continuously, the human intervenes only when necessary.

The **hibernation** phase is where the agent is left paused but persistent. State saved. It consumes no active compute. This phase is useful when an agent need not operate for extended periods — a support agent that only operates during business hours, for example, hibernates overnight and reactivates with the start of the next day. Hibernation preserves the context without spending resources.

The **decommissioning** phase is where the AgencyDomain retires the agent. The agent's state is archived or deleted according to policy. The Capabilities it had assigned are released. The agent's identity remains recorded in the historical log, but the agent ceases to exist as an operative entity. The decommissioning phase is important because it formally closes the cycle — a "decommissioned" agent is not the same as a "forgotten" agent. The record of the decommissioning is what allows, weeks or months later, an auditable reconstruction that the agent existed, what it did, and why it ceased to exist.

Communication between agents

The spec reserves the term **A2A** (`agent-to-agent`) for the *relation* between agents, and an agent is an AgencyDomain. The A2A relation is, therefore, **between AgencyDomains** — between distinct agents, each with its own identity and agency. Communication within a single AgencyDomain is not A2A in this relational sense: the components that sustain it are runtimes of the same agent, not distinct agents. The spec thus distinguishes two planes: the **intra-AgencyDomain** plane (an agent commanding its

own runtimes and muscle memory) and the **A2A** plane (an agent invoking another agent). When the **`A2A` protocol** is used within an AgencyDomain as transport between runtimes, one says **via the `A2A` protocol** — the proper name of the protocol —, never "internal A2A", so as not to attribute agency to runtimes that do not have it.

How does the Cognition command its muscle memory? — Layer 2 ↔ Layer 3 interface

The **Cognition** (the `LLM` agent, Layer 2) commands its own muscle memory — the **Botler**, a Layer 3 runtime without agency — through an **internal** interface within the same AgencyDomain. It is the Layer 2 → Layer 3 relation: the Cognition specializes, manifests, consumes, and controls the Botlets that the Botler hosts. The natural transport of this interface is **`MCP`** (`LLM`↔tool): the Botler exposes one or more `MCP` servers and the Cognition is the client. This interface **is not `A2A`** — it does not cross the AgencyDomain boundary nor mediate between distinct agents; it is an agent operating its own execution substrate. `A2A` is reserved for AgencyDomain↔AgencyDomain.

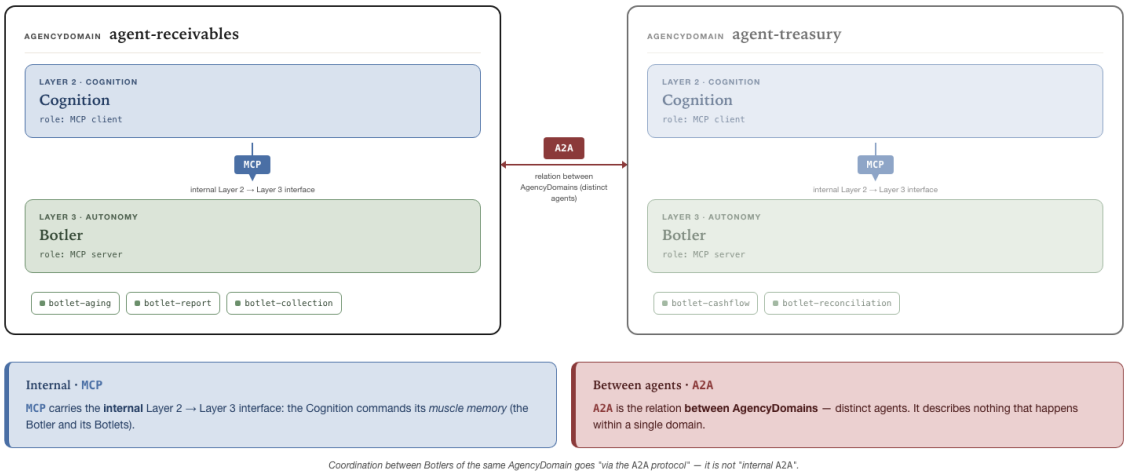


Figure 18: Layer 2 → Layer 3 interface via MCP — internal, not A2A

What properties does intra-AgencyDomain communication require?

All communication between components that live **in the same AgencyDomain** — be it the Layer 2 → Layer 3 interface via `MCP`, be it the transport between Botlers via the `A2A` protocol — satisfies three properties the spec requires. The first is **uniform addressability** — any component of the space can be invoked by its canonical URI, with no need for ad hoc discovery. The second is **message typing** — messages have a declarative, verifiable schema; the sender declares the schema and the receiver verifies that the message complies with it before processing it. The third is **traceability** — every invocation

is recorded in the append-only log with the identity of sender and receiver. If a component invoked another, the system knows who, when, and with what content.

How do distinct agents communicate? — A2A between AgencyDomains

`A2A` communication between AgencyDomains is between agents that live **in distinct AgencyDomains**. This modality requires additional protocols that the spec recognizes as necessary but does not fully normalize in its version 1.0. The open protocols for `A2A` are in evolution — the industry is converging toward certain directions, but full consensus has not arrived. What the spec does define is that `A2A` between AgencyDomains requires three mechanisms: **discovery** — how an AgencyDomain publishes the agents it offers to be invoked externally; **cross-authentication** — how two AgencyDomains verify each other's identity; **semantic resolution** — how two AgencyDomains that may have distinct glossaries negotiate the meaning of tools and capabilities when they interoperate.

The complete normative specification of `A2A` between AgencyDomains — discovery protocol, federated message format, identity resolution — is **open work**. Reference implementations may adopt emerging protocols, for example federated `MCP`, or the `A2A` protocol proposed by **Google**. When there is industry consensus on a specific protocol, a future version of this spec will incorporate it as normative. For now, serious implementations treat `A2A` between AgencyDomains as an emerging capability, not as consolidated spec.

Federation between AgencyDomains

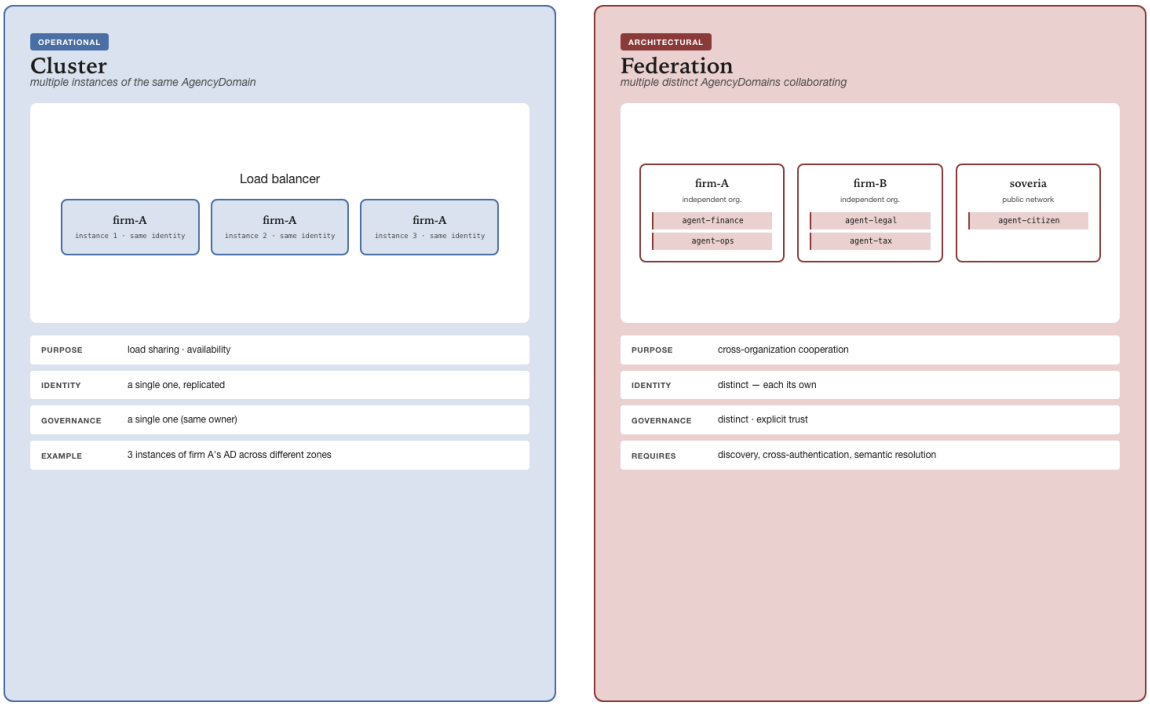


Figure 19: Federation vs. Cluster · a critical distinction

Federation is the formal mechanism by which multiple AgencyDomains collaborate as a network. It

must be distinguished from the close but distinct concept of the **Cluster** — multiple instances of the same AgencyDomain operating as a coordinated set. Cluster is operational; Federation is architectural.

Concept	Granularity	Example
Cluster	Multiple instances of the **same** AgencyDomain	Three instances of one firm's AgencyDomain sharing load
Federation	Multiple **distinct** AgencyDomains collaborating	Firm A's AgencyDomain invokes an agent from firm B's AgencyDomain

Federation enables ecosystems of agents that cross organizational boundaries. An AgencyDomain can invoke agents from another AgencyDomain, exchange data, coordinate operations — all under explicit trust models that each participant establishes. This extends the agentive model beyond the boundaries of an individual organization and allows cooperation networks that resemble the open internet more than closed corporate systems.

The normative specification of federation is open work. Version 1.0 of the spec recognizes the necessary mechanisms without prescribing their specific implementation:

An **open discovery protocol** must exist, possibly over DNS and well-knowns. When an AgencyDomain wants to discover what agents another AgencyDomain offers, it must be able to query a standard endpoint and obtain the list. The spec does not prescribe the exact format of the endpoint — that decision depends on the industry consensus that has not yet arrived.

Cryptographic identity standards for AgencyDomains and agents are necessary. Each federated AgencyDomain must be able to authenticate cryptographically — not by a shared API key, but by a verifiable mechanism that requires no central authority. Candidate technologies include W3C DIDs (Decentralized Identifiers), X.509 certificates, blockchain-based mechanisms. The spec admits any that satisfy the fundamental property: verifiable identity without central authority.

Explicit trust models are a requirement. When two AgencyDomains interact, each must declare the level of trust it extends to the other: what operations it permits, what data it shares, what audit it exercises. Trust is not binary — an AgencyDomain may trust another for low-impact invocations but not for high-impact ones, or may trust it for reads but not for writes. The spec requires these models to be explicit and configurable, not implicit in code.

Conflict resolution when two AgencyDomains apply contradictory policies. If AgencyDomain A invokes an agent from AgencyDomain B, and the policies of A and B have conflicts — A permits the operation but B prohibits it, for example —, there must be a clear mechanism to resolve the conflict. The spec defines that priority always belongs to the receiving AgencyDomain — that is, B in this case. The sender may request; the receiver decides.

Distributed Layer 3 — canonical pattern for multiple physical presence

Chapter 4 (in its Layer 3 — Autonomy section) anticipated that the four layers may be distributed technically across distinct infrastructures. This section formalizes the most frequent and operationally important particular case: the **geographic distribution of Layer 3** within a single AgencyDomain. The pattern resolves a scenario that any organization with multiple physical branches invariably encounters

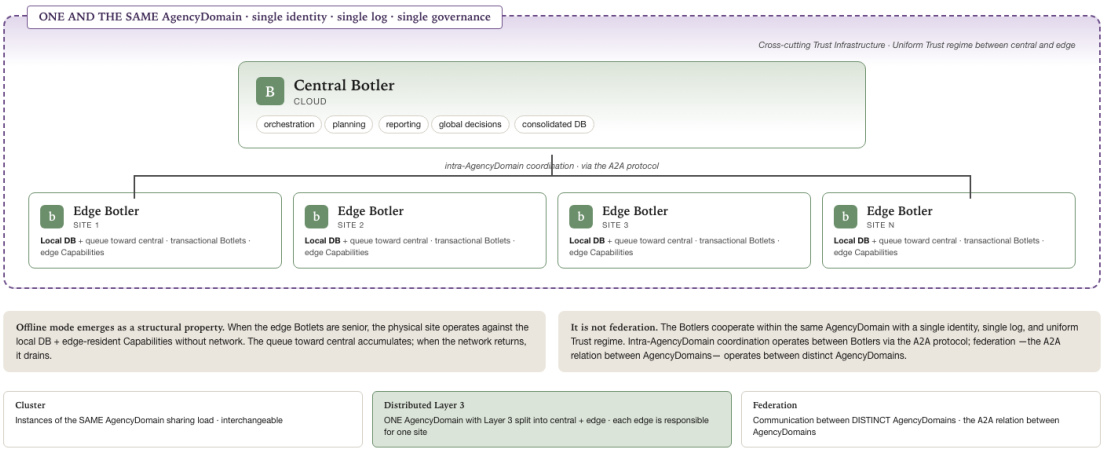


Figure 20: Distributed Layer 3 · central Botler + N edge Botlers

— multi-location food service, retail with a chain of stores, distributed logistics, healthcare with a network of centers, banking with branches, industrial plants with simultaneous production lines. Without formalization, each implementer reinvents the pattern with its own vocabulary and treats it as an exception to the model. With formalization, it stands as a **canonical pattern** that any serious implementation must contemplate.

The essential distinction is between **internal distribution** and **external federation**. Internal distribution occurs when a single AgencyDomain divides its Layer 3 across multiple coordinated physical nodes — a central Botler and N edge Botlers —, maintaining a single identity, a single governance, a single log, and a single data model. External federation occurs between distinct AgencyDomains, each with its own identity and governance. Cluster is an intermediate case (same AgencyDomain, same instances sharing load). Distributed Layer 3 is Cluster in terms of identity — all the Botlers belong to the same AgencyDomain — with the additional complication that the Botlers are not interchangeable: each edge Botler is responsible for a specific physical site.

Three pieces of the pattern

The canonical pattern distinguishes three pieces with distinct responsibilities:

1. Central Botler. Hosts the Botlets of orchestration, planning, reporting, global decisions. Lives typically in the cloud. It has a consolidated view of the state of all edge nodes. It executes operations that require crossing several sites — consolidating inventory, reconciling the day's cash, planning the next day's operation, sending consolidated regulatory reports. It maintains the consolidated DB and the unified audit log. It communicates with the cognition (Layer 2) for escalations and new decisions.

2. Edge Botlers. One per physical site. They host the local transactional Botlets — those that execute the site's daily operation: taking orders, charging, issuing receipts, managing local inventory, controlling physical devices (pinpads, printers, sensors). Each edge Botler maintains a local DB with the site's state and an **event queue toward the center** that synchronizes when there is network. They operate with full autonomy when the network is available and with local autonomy when the network goes down — the site keeps operating against its local DB; events accumulate in the queue; when the network returns, the queue drains and consolidation with the center resumes.

3. Coordination between Botlers via the `A2A` protocol. The central and edge Botlers communicate **via the `A2A` protocol** — the proper name of the coordination protocol. It is not `A2A` in the relational sense: these Botlers are runtimes of the same agent — the same AgencyDomain —, not distinct agents, so the coordination between them is **intra-AgencyDomain** communication, not `A2A` between AgencyDomains. The conversation traverses the corporate network but **does not traverse the federation** — it is entirely within a single AgencyDomain. The distinction is not rhetorical: the Trust Infrastructure regime is that of the single AgencyDomain, not that of federation between AgencyDomains; the log is unified; the identity model is internal; policies apply uniformly.

Offline mode as an emergent property

Under the parallel topology of Chapter 4 + the distributed Layer 3 pattern, the **offline mode** of a physical site emerges as a **structural property of the system**, not as a special capability that requires separate construction. When the network goes down at a site, two things happen simultaneously: the Cognition path becomes inactive (Layer 2 lives in the cloud and is not accessible), and the central Botler is also not accessible. But the edge Botler stays active: its Botlets run against the local DB, the edge-resident Connectors (ESC/POS printer, drawer, pinpad) remain available, the site's daily operation continues. The event queue toward the center accumulates pending transactions; when the network returns, it drains and consolidates.

The condition for offline mode to operate correctly is that the edge Botlets be **senior** in terms of the maturity proposal (section §2): Botlets that have already incorporated the environment's variants and operate with a ratio close to `99+ / <1 / ~0`. An edge Botlet in the junior phase — still discovering variants — cannot operate without the possibility of fallback to cognition. A senior edge Botlet can, because its only failure modes are exogenous (power, hardware, catastrophic network), not pending learning.

Properties required of the pattern

An AgencyDomain implementation with distributed Layer 3 must satisfy:

Property	Level	Description
Single identity of the AgencyDomain	**MUST**	All Botlers (central + edge) belong to the same AgencyDomain with a single canonical URI.
Local DB in each edge Botler	**MUST**	Operational state of the physical site, accessible without network.

Property	Level	Description
Event queue toward the center	**MUST**	Eventual-synchronization mechanism; pending transactions drain when there is network.
Conflict resolution in consolidation	**MUST**	When an event from edge reaches the center and conflicts with the consolidated state, an explicit policy decides.
Unified audit log	**MUST**	A single append-only log for the entire AgencyDomain, fed by all Botlers.
Single internal identity model	**MUST**	The Botlers do not authenticate to each other as external AgencyDomains; they share the AgencyDomain's identity model.
Uniform Trust regime	**MUST**	Policies apply the same at center and edge; there is no special regime for edge.
Offline operation capability at edge	**SHOULD**	When the edge Botlets are senior, the site operates with intermittent network or without network.

Portability of the AgencyDomain across conformant platforms

The regimes section formalized **natural migration** between regimes (private, public, hybrid) without rewriting. This section formalizes a complementary but distinct property: **portability across hosting platforms conformant to the spec**. A conformant AgencyDomain can be migrated to another conformant platform without rewriting its logic, its state, or its policies. This is a structural property of the spec — not the commercial commitment of a particular provider.

The motivation is operational before philosophical. Without an explicit portability commitment, the AgencyDomain repeats the lock-in of the application era — the client remains tied to its agentive provider exactly as it used to be tied to its SaaS provider. The structural promise of the spec — that the AgencyDomain is the **real property of the client**, not of the hosting — depends on portability being a property of the spec, not a concession negotiated case by case.

Three technical conditions

Portability requires three technical conditions that any conformant implementation must satisfy:

1. Botlets against canonical primitives. The AgencyDomain's Botlets must invoke Capabilities and the conformant AgencyDomain SDK, **not the current hosting's proprietary APIs**. If a Botlet invokes a provider-specific API — `cloudprovider.lambda.exec`, `vendor.workflow.run` —, that invocation is portability debt. When the time comes to migrate, that Botlet must be regenerated to invoke the equivalent canonical primitive. A conformant implementation provides SDKs and registries that abstract from

the hosting; the Botlet sees the primitive, not the implementation.

2. Exportable operational DB. The persistent state of the AgencyDomain — agents, Botlets, capabilities, audit log, operational data — must be exportable in a neutral format, without dependencies on the hosting's storage engine. A documented schema (portable DDL or equivalent representation). A complete dump (all the information needed to reconstruct the space on another platform). No proprietary data types. No engine-specific extensions that have no equivalent in standard engines. The DB's portability is what allows migration not to be a rewrite.

3. Portable Trust Layer. The policies, the append-only log, the Capabilities configuration, and the identity bindings must be maintained in a neutral reproducible format — typically structured Markdown or YAML/JSON with an explicit schema. The spec does not prescribe the exact format, but it requires the format to be **readable by any conformant implementation**, not only by the current one. A policy that only one provider's policy engine knows how to interpret is not a policy of the AgencyDomain — it is provider configuration.

Natural migration vs portability

The two properties — natural migration between regimes and portability across platforms — are complementary but distinct:

Axis	Natural migration between regimes	Portability across platforms
What changes?	The AgencyDomain's regime (private → public)	The hosting platform
What remains?	The hosting platform	The AgencyDomain's regime
Who decides?	The owning organization, by usage maturity	The owning organization, by economics or strategy
Expected frequency	Once or twice in the AgencyDomain's life	Zero or few times, but the possibility must exist

Portability is not a promise that migration will be trivial — there will always be a cost of orchestration, validation, cutover window. It is a promise that migration will be **possible without rewriting the agent's logic**. That difference — between possible-with-work and impossible-without-rewrite — is what separates a conformant AgencyDomain from a proprietary agentive system in disguise.

Conformance

An implementation that intends to call itself **conformant AgencyDomain** to this specification must satisfy the following list of requirements. We mark them with the IETF convention: **MUST** for mandatory, **SHOULD** for strongly recommended, **MAY** for optional.

Requirement	Level
Own identity	**MUST**
Materialization of the four layers	**MUST**
State persistence	**MUST**
Isolation between spaces	**MUST**

Requirement	Level
URL addressability	**MUST**
Canonical data model	**MUST**
Support for the three regimes	**SHOULD** (at least one; ideally all three with migration)
Support for Tokens and Subscription modes	**MUST** both
Complete agent life cycle	**MUST**
Intra-AgencyDomain communication (Layer 2 → Layer 3 interface via `MCP`; coordination between runtimes via the `A2A` protocol)	**MUST**
`A2A` between AgencyDomains	**SHOULD**
Federation	**MAY** (when the normative spec is available)
Cross-cutting Trust Infrastructure	**MUST**
Agent First principle	**MUST**
Distributed Layer 3 (central + edge Botler)	**SHOULD** (when there is multiple physical presence)
Portability across conformant platforms	**MUST** (Botlets against canonical primitives, exportable DB, portable Trust Layer)

An implementation that meets all the **MUST**s is an AgencyDomain conformant to version 1.0 of the spec. An implementation that meets the **MUST**s and the **SHOULD**s is what we would call a reference AgencyDomain — an exemplary implementation the industry can take as a base. Implementations that also meet the **MAY**s are frontier implementations, which typically lead the evolution of the field.

Reference implementations

As we mentioned in Chapter 4, this specification is agnostic to implementation. The **public reference implementation** is **Vergis**: distributed under `AGPL`, with a public repository at AgencyDomains.org, designed so that any developer or student can download it, read it, run it, and learn how the spec translates into a living system. **Chapter 9** develops it in detail; here it suffices to leave the pointer and assert that it is conformant.

On the same canonical structure, product implementations in complementary regimes also operate: **Agentia** materializes AgencyDomains in the **private regime** within the infrastructure of the client firm, and **Soveria** materializes them in the **public regime** as a network of agents with a public identity. Other implementations are admissible and welcome. The specification intends to be an industry standard, not the intellectual property of a single actor.

Evolution frontier

Three areas of the specification are under active evolution and a future version of the book will probably normalize them in greater detail.

Federation is the first. As we mentioned, the normative protocol is not yet agreed upon by the industry. Version 1.0 recognizes the necessary mechanisms without prescribing them in detail. When consensus arrives — probably within the next two to three years —, the spec will incorporate it.

Agnostic cognition is the second. The spec admits non-LLM cognition — symbolic, multimodal, hybrid. The contemporary implementation is predominantly LLM-centric. The extension to other cognitive substrates requires refinement of the interfaces between Layer 2 and the rest of the AgencyDomain's components.

Cryptographic identity of agents is the third. The model of verifiable on-chain or DID-based identity is under exploration. Adoption depends on the broader decentralized-identity ecosystem maturing sufficiently to support the agentive use case.

Two of these frontiers — federation and agnostic cognition — coincide with those of Chapter 4; cryptographic identity is specific to this spec. All of them are frontiers of the architecture itself, not only of its materialization in AgencyDomains.

Botlets

When a pianist learns a new piece, the first minutes are a conscious, costly experience. The pianist looks at the score, identifies each note, decides the fingering, executes each finger movement with full attention. The piece, in that first reading, is intense cognitive work. An hour later, after deliberate practice, the fingers begin to play on their own. The pianist still follows the score, but no longer has to consciously decide each note — the fingers know where they go. A week later, the piece is embedded in **muscle memory**: the pianist executes it without thinking. If at some point a mistake occurs, or something departs from the expected — an odd sound, an awkward fingering — consciousness reappears briefly, evaluates the problem, adjusts, and then withdraws again. Muscle memory takes back control.

This dynamic of human motor learning is not an arbitrary metaphor. It is the neurobiological basis documented by Squire and Wixted in their 2011 work on human memory systems, extending Larry Squire's earlier work on memory modalities. The prefrontal cortex learns new patterns by spending costly cognitive resources. It hands them off to subcortical structures — the cerebellum, the basal ganglia — that execute them without conscious intervention. The prefrontal cortex reactivates only when it detects a significant deviation that the encoded routine does not handle. This architecture is what lets the human brain operate efficiently: the costly is minimized, the cheap is maximized, and conscious attention is reserved for when it is genuinely needed.

The Agentive Architecture replicates this biological architecture with discipline. What the brain does with prefrontal cortex and procedural memory, the agentive system does with LLM cognition and Botlets. When an agent faces a task for the first time, cognition — Layer 2 — processes it with the costly resources of the LLM: it reasons, decides, executes. When the task recurs frequently, cognition delegates execution to a **Botlet** — traditional, non-LLM code that cognition itself generated — which lives in Layer 3 and executes without invoking the model. If the environment changes and the Botlet fails, cognition takes back control: it regenerates the Botlet adapted to the new environment or, in the worst case, executes the task manually. The costly is minimized, the cheap is maximized, cognition is reserved for genuinely new cases.

> *Cognition thinks once. The Botlet executes ten thousand times.*

This section develops the Botlet primitive with the detail it deserves. The Botlet spec is probably the most important primitive for the operational economy of an agentive system — without Botlets, inference costs make continuous autonomy unviable; with well-designed Botlets, the organization can operate agents in production at predictable, stable costs.

Definition

A **Botlet** is a self-evolving unit of automation: traditional, non-LLM-based code, generated by an agent to execute a repetitive task without invoking costly cognition. Botlets are the agent's **muscle memory** — the computational equivalent of the automated gestures a human executes without conscious thought.

> **Botlets belong to the agent.** The decision that a Botlet should exist may come from the design — it is the seed Botlet — or from experience — it is the emergent one; in both cases the agent that serves the flow makes it, and from then on it is the agent's: it is its muscle memory, crystallized when it sees a task recur or when the design planned it, and its evolution, its regeneration, and its replacement belong to the agent. No one builds it separately for the agent to use. The agent may start from a proto-Botlet or a Botlet that already exists — from its catalog, the public one, or a codex — but by taking it, it makes it its own; what it keeps from the origin is set by the catalog's membership mode (Chapter 9). In `G1` the agent does not write the body of the piece: it configures it, decides when to replace it with another and, if none fits, specifies it to be forged in its Preparation.

Four properties distinguish the Botlet from any traditional "macro" or "automated script". The first is that **the Botlet's code was not written by a human**: the agent's cognition generated it when it recognized a repetitive pattern in its activity. This matters because dynamic code generation lets the system adapt the automation to each particular context, without depending on a programmer to anticipate every case. The second is that the Botlet **executes without invoking cognition** during normal operation. Once generated, the Botlet runs as traditional code — Python, JavaScript, Bash, whatever — independent of the model that created it. The third is that it **regenerates automatically** when it detects that the environment has changed. If the Botlet fails because an API changed, a data structure varied, or a screen was renamed, cognition regenerates the Botlet adapted to the new environment. The fourth — and critical — is that it has a **fallback guarantee**: if the Botlet fails catastrophically and cannot execute, cognition executes the task manually. The process never stops.

The difference from a traditional script is structural. A traditional script that fails leaves the operation halted until a human intervenes: someone must identify the problem, modify the script, redeploy it, validate. A Botlet that fails activates cognition, which executes the task — in that particular case, without a Botlet — and logs the event to regenerate afterward. The organization can **depend** on the Botlet without operational risk, because the Botlet's failure is not the system's failure.

The `95/4/1` cycle

The operational lifecycle of a Botlet in production is typically distributed in the proportion that gives this model its name: **`95/4/1`**. The proportions are approximate but structurally correct: most of the time the Botlet executes without invoking cognition; occasionally the environment changes and the Botlet fails; rarely does cognition have to regenerate the code.

Ninety-five percent of the time, the Botlet is in **normal execution**. Cognition is not invoked. The Botlet runs, completes its task in seconds or minutes depending on the case, returns a result. This is the efficient phase — where the whole economy of the agentive system rests. An organization operating a thousand agents with well-designed Botlets executes ninety-five percent of its tasks at traditional compute cost, not at LLM inference cost. The economic difference is one or two orders of magnitude.

Four percent of the time, the Botlet detects a **change in the environment**. It fails or returns an anomalous result. The environment changed: an API field moved, a data structure varied, an external system's response has a different format. The Botlet, written for the environment of two weeks ago, no

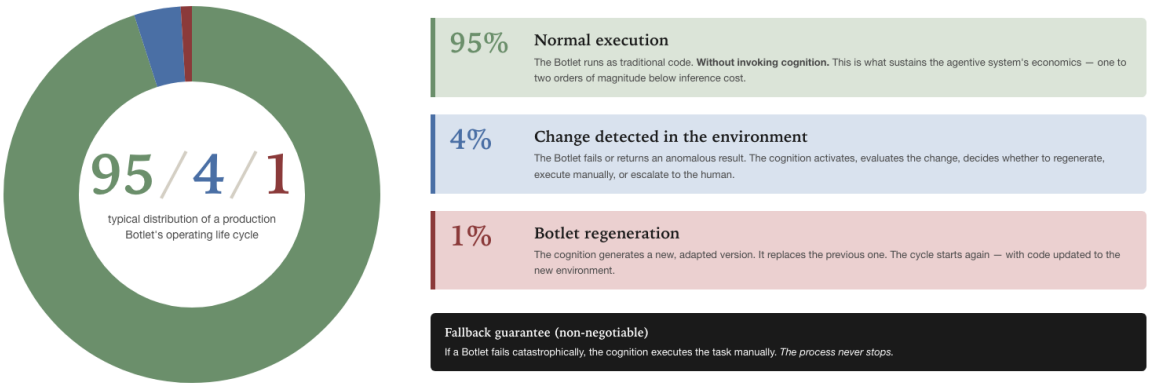


Figure 21: The `95/4/1` cycle · the agent's muscle memory

longer works. Cognition activates. It evaluates the change: is this something that can be handled by regenerating the Botlet? is this something that requires a one-off manual execution in this case? is this something that requires escalation to a human?

One percent of the time, cognition decides to **regenerate the Botlet**. It generates a new version adapted to the changed environment. The new version becomes the active Botlet, replacing the previous version. The next invocations — the new ninety-five-percent phase — use the regenerated version. The cycle closes and begins again.

The exact proportion varies by case. A Botlet operating against a very stable external system may hold ninety-nine percent normal execution and only one percent change detected. A Botlet operating against a volatile external system may drop to eighty percent normal execution with fifteen percent changes and five percent regeneration. What matters is not the specific proportions: it is the **structure** of the cycle. The Botlet executes most of the time without cognition; cognition is reserved for the cases where the environment changes.

Botlet maturity — junior, learning, senior

The `95/4/1` cycle is a useful didactic presentation, but it is **static**: it describes the steady state of an already-formed Botlet, not the trajectory by which the Botlet reaches that state. Operational reality demands an additional distinction: a freshly generated Botlet **does not operate at the `95/4/1` proportion**. It operates at a worse proportion. Only after incorporating the environment's variants does it reach the maturity the canonical cycle describes. This section formalizes the trajectory.

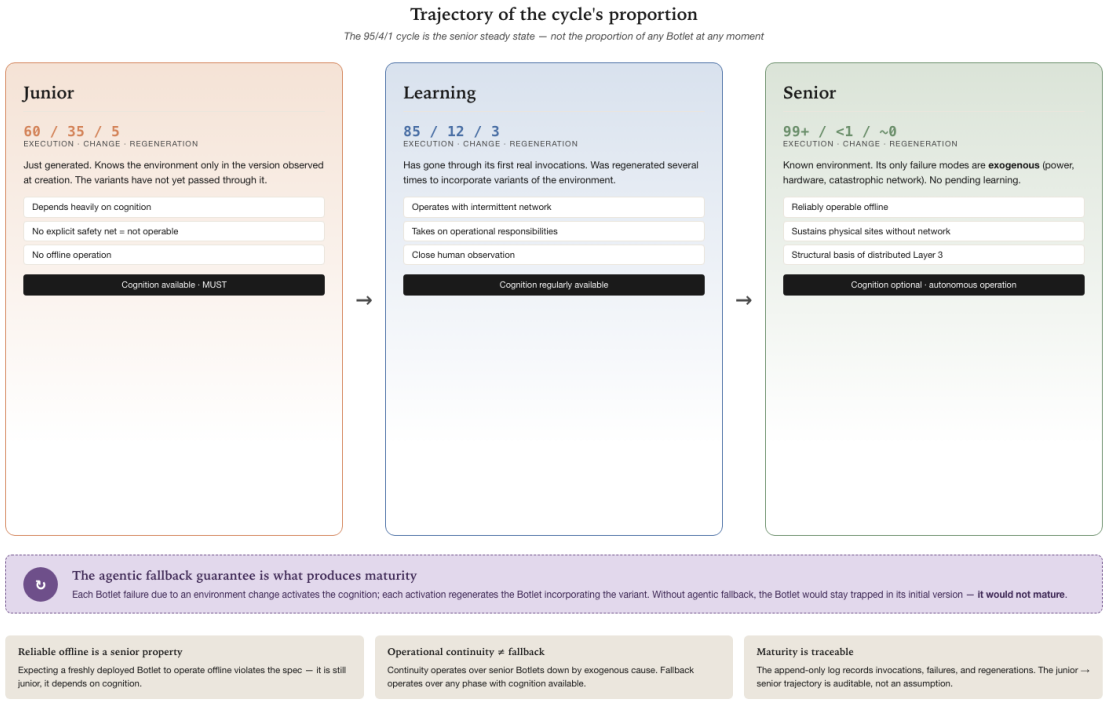


Figure 22: Botlet maturity · junior → learning → senior trajectory

The spec recognizes **three canonical phases** in a Botlet's maturity: junior, learning, senior. The phases are not administrative labels; they are states with distinct properties that the system tracks in order to decide how much to delegate to the Botlet and when to escalate it.

Junior phase

A **junior** Botlet is a freshly generated Botlet. It has just come out of cognition. It knows the environment only in the version cognition observed when it created it. The environment's variants — dates in a different format, error messages with new wording, optional fields that appear only sometimes, edge cases — have not yet passed through it, so its code does not account for them.

The typical proportion of a junior Botlet is something like `60 / 35 / 5`: only 60% of invocations are successful normal execution; 35% are failures due to environment variants the Botlet does not anticipate; 5% are regenerations when cognition decides the observed variant is structural and must be incorporated. The proportion is unfavorable, but it is not a problem — it is the normal phase of any freshly generated Botlet, and cognition is there precisely to rescue it.

Operationally, a junior Botlet depends heavily on the availability of cognition (Layer 2). It cannot operate offline because too many of its invocations require rescue. Nor can it assume critical responsibilities without an explicit safety net.

Learning phase

A **learning** Botlet is a Botlet that has already passed through its first real invocations. It has faced environment variants and has been regenerated several times to incorporate them. Its proportion moves toward something like `85 / 12 / 3`: most invocations are successful, failures due to new variants are less frequent, regenerations are occasional.

The learning phase is the longest phase of the Botlet's useful life — it can last weeks or months depending on invocation frequency and the environment's volatility. Each regeneration consolidates operational know-how: each variant incorporated is one less variant that can surprise the Botlet in the future.

Operationally, a learning Botlet can operate with an intermittent safety net — failures are still frequent enough to need cognition available regularly, but not at every invocation. It can assume operational responsibilities under close human observation.

Senior phase

A **senior** Botlet is a Botlet that has already incorporated the environment's variants. Its proportion tends toward `99+ / <1 / ~0`: practically all invocations are successful normal execution; failures due to environment changes are rare because the environment now rarely presents it with something it does not know; regenerations are exceptional.

A fundamental property of the senior Botlet changes relative to the earlier phases: **its failures in the senior state are not environment changes; they are exogenous causes**. When a senior Botlet fails, the typical cause is something that would halt any stable system: a power outage, downed hardware, catastrophically lost network, a downed external resource (a tool provider, a regulated system). These failures are not pending learning — they are the same thing any operating system encounters occasionally and resolves with redundancy, restart, or human intervention.

Operationally, a senior Botlet can operate **offline reliably**. The reason is structural: if its only failure modes are exogenous, the presence or absence of cognition does not significantly change the probability of failure — cognition has no way to rescue from a power outage. The senior Botlet, against a local DB and edge-resident Connectors, sustains the physical site's operation even when cognition is unreachable. This property is the structural basis of offline mode in systems with a distributed Layer 3 (Chapter 5 §1).

Implications of the trajectory

The distinction among the three phases has three implications worth keeping in mind.

First, reliable offline operation is a property of senior Botlets, not of Botlets in general. To expect a Botlet freshly deployed at a new site to operate offline is to violate the spec — it is still junior, it depends on cognition. The trajectory of an edge node from production launch to full offline operation requires time of exposure to the environment; it is not an instantaneous property.

Second, the **agentic fallback guarantee is what produces maturity**. Each Botlet failure due to an environment change activates cognition; each activation of cognition regenerates the Botlet incorporating the variant. Without agentic fallback, the Botlet would be trapped in its initial version, with no way to learn. The connection with the operational business-continuity section of §4 is direct: the fallback guarantee resolves the junior phase and the transition toward senior; operational continuity resolves the exogenous failures of the senior phase.

Third, a Botlet's maturity **is traceable**. The Trust Layer's append-only log records each invocation, each failure, each regeneration. The proportion of each phase is observable, and the trajectory of a Botlet from junior to senior is auditable. This traceability is what lets the organization make operational decisions — *"this Botlet is now senior, we can delegate critical responsibilities to it"* — on evidence, not on assumption.

Seed Botlets vs emergent Botlets — the origin of the Botlet

The cycle described so far assumes that **Pattern Recognition** — the auxiliary primitive developed further below — triggers the generation of a Botlet upon detecting an unanticipated repetitive pattern. That is the **emergent** modality of generation. It is the modality the neurobiological model inspires and the one the `95/4/1` cycle describes in its purest form. But **it is not the only modality**, and for real productive systems it is not even the most frequent.

In a productive system, the MVP's critical Botlets **do not emerge**: cognition implements them because the design team planned them as part of the product spec. The team knows, before the system sees its first transaction, that it is going to need a POS Botlet, an order-ticket Botlet, a charge Botlet, a shift-close Botlet. Cognition executes the implementation of those Botlets; but the **decision to exist** was made by the design, not by Pattern Recognition.

The spec therefore distinguishes **two canonical origins** of the Botlet:

Seed Botlets. Generated by cognition at the design team's request, as part of the initial product. Cognition executes the implementation — it writes the Botlet's code, registers it in the Botler, deploys it to the corresponding environment — but the decision of which Botlets there should be, what tasks they cover, and under what contracts they operate, belongs to the design team. Pattern Recognition does not participate in seed generation.

Emergent Botlets. Generated by Pattern Recognition when cognition, during operation, detects a repetitive pattern not anticipated by the design. Cognition evaluates whether the pattern merits automation, decides affirmatively, and generates the Botlet. This is the modality the previous section described.

Both live and operate identically once generated — both are subject to the `95/4/1` cycle, both pass through the junior → learning → senior phases, both have a fallback guarantee, both are auditable. The difference is in the origin.

> *Pattern Recognition is not the only path to a Botlet. Design is not technical debt either. The two paths coexist.*

The distinction has three practical consequences.

First, a productive agentive system **does not require waiting for Pattern Recognition to discover the critical Botlets**. Seed Botlets are generated at the outset according to the product spec, and the system enters production with the battery of Botlets needed to operate. Pattern Recognition comes later, during the system's life, to optimize what the design did not anticipate.

Second, seed Botlets can live **in any layer**, not only in Layer 3. The **persistent GUIs generated as facade Botlets** of Chapter 4 §1 illustrate it: the facade Botlet is a Layer 3 seed Botlet that exposes its stable surface in Layer 1, and the presentation Botlets that compose that surface (shells and views) are Layer 1 seed Botlets — generated by cognition at the design team's request because the operational role (cashier, cook, plant operator) justifies it. The canonical definition of the seed Botlet allows these materializations without the spec treating them as exceptions.

Third, the maturity trajectory applies equally to seed Botlets and emergent Botlets. A freshly deployed seed Botlet is **junior**; a seed Botlet that has already operated thousands of times and consolidated its know-how of the environment is **senior**. The origin distinction does not change the trajectory; only the moment of onset.

proto-Botlet — the pre-forged piece

The seed-vs-emergent distinction describes *who decides* that a Botlet should exist. A prior question remains: when cognition generates a seed Botlet, does it write its code from scratch every time? In operational practice, no. A Botlet's code rarely springs from nothing: it springs from a pre-forged piece that the agent configures.

A **proto-Botlet** is a pre-forged piece of operational capability that the agent, in its Engineering time, **configures** to instantiate a Botlet specific to the case. The proto-Botlet contains the code; the Botlet is the configured instance. The relation is generic → instance: the proto-Botlet lives in a catalog and serves many cases; the Botlet is one of those cases resolved. The catalog belongs to the agent: the agent refines it in its own Preparation, and what it takes from others it incorporates as its own.

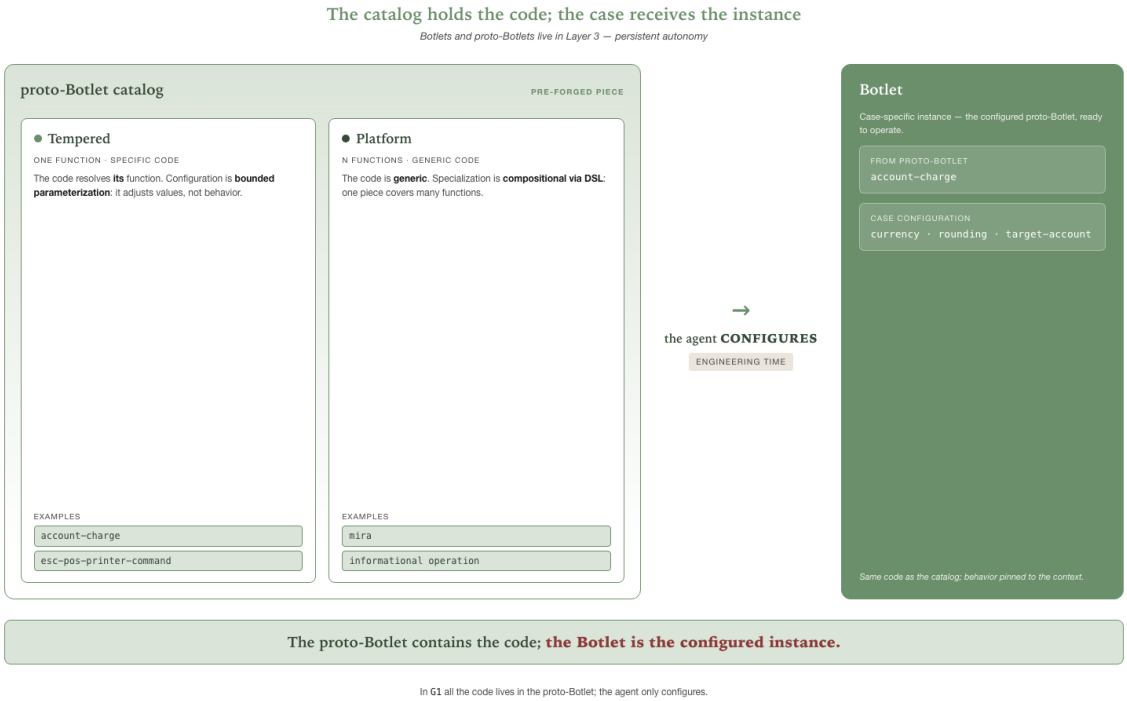


Figure 23: proto-Botlet — the pre-forged piece from the catalog (tempered · platform)

The connection with the origin of the Botlet is direct. A seed Botlet that the design team planned does not force cognition to write its entire logic: if the catalog has a proto-Botlet that covers the function — charging an account, an order against a ticket printer, an informational operation — cognition instantiates the Botlet by **configuring** that proto-Botlet rather than generating it. The decision to exist still belongs to the design (it is seed); the materialization of the code rests on the pre-forged piece.

The spec recognizes **two classes** of proto-Botlet, according to the nature of their code:

Class	What is its code?	How is it configured?	Anonymized example
Tempered	Code specific to its function	Bounded parameterization	Charging an account; an order against a ticket printer
Platform	Generic code whose specialization lives in a compositional configuration	Compositional configuration, covering N functions of the domain	An informational-operation piece that serves reports and dashboards in many forms

A **tempered** proto-Botlet resolves a function and resolves it in full; configuring it is adjusting parameters within a foreseen range. A **platform** proto-Botlet is an engine: its code is generic and the specific function emerges from a rich configuration — compositional, not a flat list of parameters — so that a single platform proto-Botlet covers N functions of its domain. **Mira**, in the reference implementation's catalog, is a platform proto-Botlet for informational operation.

An agent may seed its catalog with others' proto-Botlets — public in AgencyDomains.org, private in proprietary codices — but by adopting them it makes them its own: their evolution, regeneration, and replacement become the agent's, just like those of the proto-Botlets it forged itself. Adoption is a fork; depending on the catalog's membership mode (Chapter 9), the agent may take its origin's improvements when it decides. The protocol by which catalogs reconcile versions is open work. And the degree to which the agent configures the pre-forged piece, co-writes its code, or generates it entirely defines the **Botlet generations** — the next section fixes them.

The Botlet generations — `G1`, `G2`, `G3`

The generations are the evolutionary model of how a Botlet's code is born as the state of the art of cognition advances:

- **`G1`** — the agent, in its Engineering time, configures pre-forged proto-Botlets from the catalog. If none serves, it specifies a new one to be forged in the next Preparation.
- **`G2`** — the agent co-writes proto-Botlets with human or model assistance. Part of the work that in `G1` happened in Preparation migrates to Engineering.
- **`G3`** — the agent generates the Botlet's complete code in its Engineering time, pre-forging nothing. The asymptotic scenario.

The architecture is the same across all three generations; what changes is the **scope of the Engineering** the agent performs. An implementation can operate in `G1` today and migrate incrementally toward `G3` as the state of the art allows, with no re-architecture.

A higher generation is not a destination. The previous sentence — migrating toward `G3` — induces, read alone, a false conclusion: that `G3` is the destination and `G1` a primitive way station. The error comes from projecting **two distinct axes onto a single arrow**:

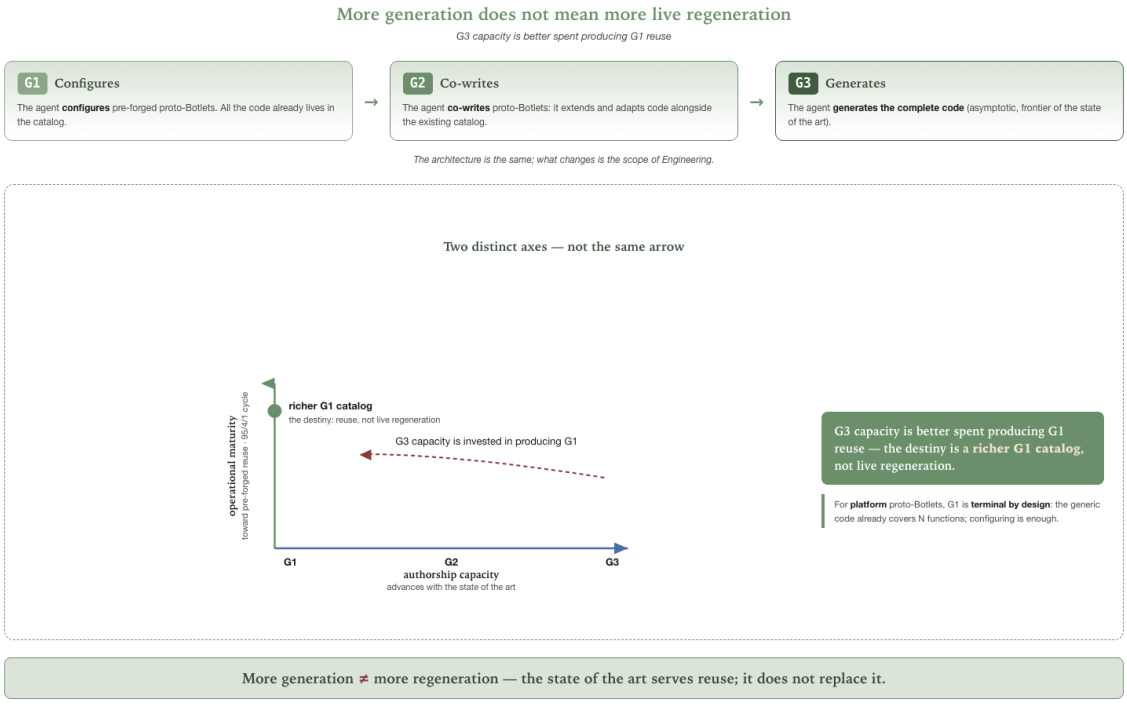


Figure 24: Botlet generations — authoring capacity vs operational maturity

Which axis?	What does it measure?	Direction of "progress"?
Authoring capacity	How much the agent can forge: configure (`G1`) → co-write (`G2`) → generate whole (`G3`)	Toward `G3`, as the state of the art of cognition advances
Operational maturity	For a recurring operation, how much is reused pre-forged vs regenerated each time (the `95/4/1` cycle)	Toward reuse (`G1`), as the Botlet matures junior → senior

They are not the same arrow. An agent with `G3` capacity that regenerates every artifact from scratch on every execution is not advanced: it has the muscle and chooses to re-learn the move every time. The reconciliation is direct: `G3` capacity is best spent **producing `G1` reuse**. The generations describe what the agent *can* author; the `95/4/1` cycle describes what a mature agent *reuses*. The destiny of `G3` capacity is a richer `G1` catalog, not the live regeneration of everything.

There is a corollary for **platform** proto-Botlets. For one of them, `G1` is **terminal by design**, not a way station: its identity is generic code plus configuration. A platform proto-Botlet "in `G3`" — where the agent regenerates the engine for each piece — is not a more advanced version; it dissolves the proto-Botlet and collapses back into the agentic mode the architecture exists to transcend.

`G1` is not poor configuration. What defines `G1` is that the agent does not write the body of the proto-Botlet — but the configuration it fills in can be as rich as a compositional `DSL` with evaluable formal

expressions. The `G1`/`G3` distinction is about **authorship of the proto-Botlet's body**, not about the expressiveness of the configuration. A platform proto-Botlet with a rich `DSL` is pure `G1`.

That leaves one borderline case: configuration that admits evaluable formal expressions — `SQL`, chart specifications, filter expressions. The **`G1`/`G2` edge** resolves it:

- A formal evaluable expression that is a **parameter of a well-defined Capability** (`SQL` → `execute-sql`, a chart specification → `render-chart`, a filter expression → `filter-stream`) is configuration → **`G1`**.
- An expression that **extends or overrides the proto-Botlet's internal logic** — callbacks, lambdas the proto-Botlet evaluates internally, fragments concatenated to its body — is code written by the agent → **`G2`**.

The test is a single one: *"does the code belong to the invoked Capability or to the proto-Botlet itself?"*. If a catalog Capability evaluates it, `G1`; if the proto-Botlet evaluates it in its internal logic, `G2`.

The reference implementation, Vergis, operates today in `G1`: its catalog exposes proto-Botlets — Mira among them — that the agent specializes by configuring, not regenerating (Chapter 9). The deeper sense of the generations — why the advanced agent generates *less*, not more — is the essay that closes the book.

Fallback guarantee — the non-negotiable property

The fallback guarantee deserves detailed treatment because it is what makes the Botlet **operationally reliable** instead of **fragilely automated**. An organization that depends on a Botlet for a critical operation — processing a nightly batch, sending regulatory reports, reconciling transactions — must be able to trust that the Botlet will execute or, failing that, that someone will execute the task in its place. The fallback guarantee is what sustains that trust.

When a Botlet fails catastrophically — not because the environment changed slightly and cognition can regenerate the code, but because something genuinely prevents execution — cognition executes the task manually. *Manually* in this context means the LLM does the work step by step, invoking the underlying tools the Botlet would use, but without the efficiency of compiled code. The process is slower and more costly — cognition consumes tokens — but **the work gets done**. The organization is not left halted.

This guarantee is not decorative. It is what distinguishes the Botlet conforming to this spec from any fragile "smart macro" or "AI automation". Traditional macros fail and leave the operation halted; Botlets fail and cognition takes over. The difference is structural and translates directly into operational availability: an organization with correctly designed Botlets can promise operational SLAs that would be impossible with traditional automation.

> *The Botlet does not replace cognition. It frees it from repetitive work, but it remains as a safety net.*

The quote above captures the relation between the two layers well. Cognition is not residual — it remains the general intelligence that sustains the system. The Botlet is operational efficiency that operates while the environment allows it. When the environment leaves the range, cognition returns.

When to use Botlets, and when not?

Not every task benefits from being delegated to Botlets. The spec defines clear criteria for deciding when it is worth generating a Botlet and when it is worth keeping the task under continuous cognition.

It is worth generating a Botlet when the task is repetitive — more than ten invocations is a useful rule of thumb — when the pattern is stable at its core even though the environment may change, when the process is critical and must be fast, when the cost of cognition per invocation is material at scale, and when there is tolerance for sporadic regeneration of the code without that affecting the operation.

It is not worth generating a Botlet when the task is unique or low-frequency, when the pattern is highly variable and each invocation requires fresh judgment, when the task demands deep reasoning a script cannot capture, when it is a prototype or exploration where flexibility matters more than efficiency, when total cost is irrelevant and continuous cognition is practical, or when changes in the environment are so constant that the Botlet would regenerate all the time, losing its benefit. There is an intermediate case that deserves a name of its own: the task **recurrent in its form but interpretive in every instance** — the pattern is stable, but each execution demands fresh judgment that no deterministic code captures. That task belongs neither to the Botlet nor to continuous cognition: it belongs to the **Agentlet**, the sibling primitive that §7 of this chapter formalizes.

The rule of thumb that synthesizes these criteria: **if the task executes more than ten times and its logic is stable at its core, it is worth generating a Botlet**. Below that threshold, cognition is more efficient. Above it, the economic difference begins to be material.

An important observation: the decision of when to generate a Botlet is not made by a human. It is made by cognition itself, assisted by Pattern Recognition, which detects the repetitive patterns. The human defines the general rules — what types of tasks are candidates, what frequency thresholds are relevant, what types of environments are sensitive — but the specific decision in each case emerges from the agent's behavior. This is a property of the agentive system: the optimization decision is the agent's own, not external to it.

The three Botlet tests — minion, specialty, and life

The criteria above say when it is *worth* generating a Botlet. What is missing is the prior question, which is one of identity, not convenience: faced with an artifact that already exists — a program, a configuration, a report, an instrument — is it a Botlet or something else? The question is not academic. Two readers with this very chapter in front of them can disagree within minutes on whether each Information Product of Mira is a Botlet or whether Mira is, and the disagreement is not settled by rereading the definition: it is settled with tests. There are three, and an artifact is a Botlet only if it passes all three.

The minion test. A Botlet is a minion with a trade. The question is what it does with its life. If it **executes** a task over and over — each execution being one case of its life — it is a minion. If it is only **applied**, read, or consumed, it is not the minion: it is what a minion handles. The short form of the question is *does this execute, or is it applied?* An Information Product executes: every pulse of a discrete Botlet and every subscription to a continuous one is an execution, and its life is the succession of those executions adapting to a source that changes. An assessment instrument is applied: someone sits it, and the instrument did nothing. The subtlety this test protects is that the cases of a minion's life are its executions over time, not different assignments: a worker at a single station on the line has a full trade even though they always do the same thing. And the corollary: when one counts as many Botlets as there are cases, look at whether those cases execute or are applied; if they are applied, one is counting instruments.

The specialty test — `RISC` versus `CISC`. It decides granularity when there are two ways to count. A `CISC` architecture offers few complex instructions, each doing a lot; a `RISC` one, many simple instructions with a single function each, which compose. The canon chooses `RISC` — the same preference that

made Unix a collection of small tools chained together —: **one Botlet, one specialty.** When the doubt is between "one Botlet that handles N things" and "N Botlets that each do one thing", the canonical answer is the latter, and what keeps N Botlets from being N programs is the shared engine: the platform proto-Botlet. A Mira conceived as a single Botlet administering N Information Products would be `CISC`: its identity, its maturity, its fallback, and its quality contract would be averages of a monolith, and when one source changed the whole would degrade. One Botlet per Information Product is `RISC`: each one falls, matures, and retires on its own. The test does not say "always smaller"; it says **one specialty per unit**.

The life test. Does the artifact have a life cycle to manage? A Botlet is running or stopped, active or retired; it can fail on its own; it matures from junior to senior; it has temporality and a quality contract; it answers to `status`, `activate`, `deactivate`, and `retire`. If there is something that can be alive, down, degraded, or retired, there is a Botlet. If all the artifact has is an editorial state — current or withdrawn as content, a version — it is not a Botlet: it is catalog. An Information Product has a life: it is serving or it is down, its source changed, it matured. An assessment instrument has a version, not a life.

Applied to the reference implementation, the three tests give the same result as the granularity rule below: **Mira is the platform proto-Botlet — the engine —; each live Information Product passes all three tests and is a Botlet; the Information Product's spec is what specializes it.** Applied to a sibling case outside the information family — a study platform that applies assessment instruments to a student — they give the opposite result: the assessor that applies, grades, and records passes all three and is a Botlet; the instrument it applies passes none, and is catalog content, immutable once published. Both answers come from the same tests, which is what is asked of them. That assessor has a proper name in the reference implementation — **Daftar**, the second proto-Botlet in Vergis's catalog (Chapter 9) — and it belongs to the **capture** family, which the next section defines.

Manifestation and temporality of the Botlet

A Botlet is muscle memory: a latent disposition, a stored know-how that is nothing until it is exercised. When the Botlet executes, that latent is actualized in the world. That actualization is its **manifestation**: the Botlet's passage from potency to act, perceptible or not.

The word demands care. **Manifestation is not appearance.** The common term suggests "becoming visible", and that would leave out legitimate cases: a Botlet that triggers a periodic ingestion manifests — it actualizes its latent, produces an effect — even though it leaves no visible artifact. That is why the canon defines it as *actualization of the latent*, not as *appearance*: the invisible effect counts as much as the artifact in view.

Manifestation is the **abstract genus**; each Botlet family specializes it, and each practice gives it its loaded name. The families are distinguished by the **direction of the effect** — from where to where what the Let moves goes — and there are four of them:

- the **information** family → from the Let to the human: it **represents** something the system already has; its manifestation leaves an **Information Product** (`PI`),
- the **capture** family → from the world to the Let: it **records** something that was not in the system before — a measurement, an answer, an ingested file —; its manifestation leaves a **record**, and each domain's practice gives it a name: an attempt, a reading, an entry,
- the **action** family → from the Let to a system: it **changes** something, and the Let itself closes the loop; its manifestation is an effect on the world, with no artifact,
- the **decision** family → it **resolves** among alternatives; named by its own practice.

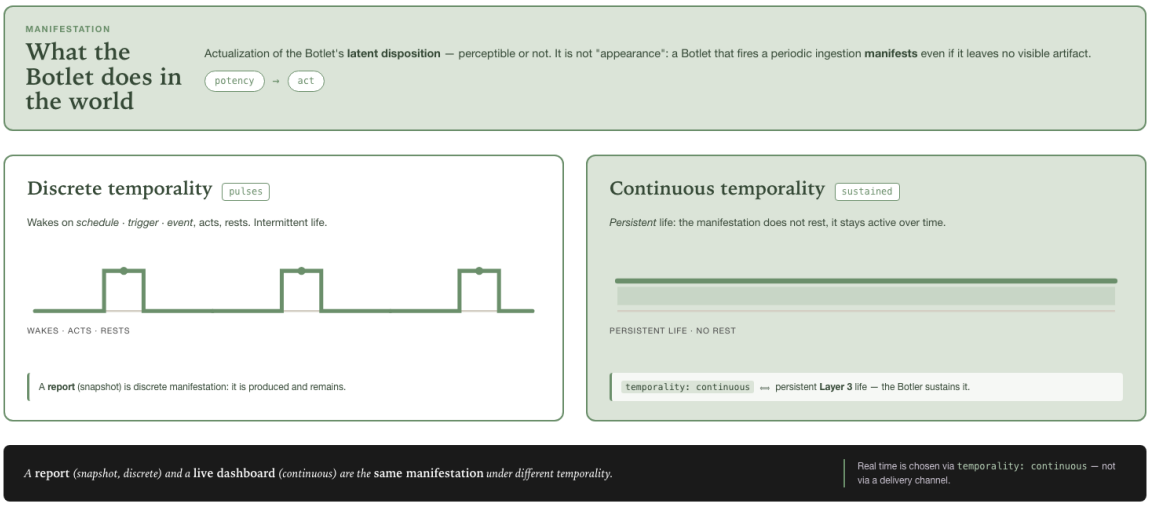


Figure 25: Manifestation and temporality of the Botlet — discrete vs continuous

The membership test is a single question: *what was in the system before and after the Let?* If the system is left unchanged and only the human knows more, it is information; if a datum that was not there came in, it is capture; if something outside the system changed by the Let's own hand, it is action; if an alternative was closed, it is decision. The periodic ingestion that served as an example a few lines above is capture: it records what the world had and the system did not. And an interactive Let — the one that asks and records what it is answered — **composes** information and capture without being a new family: a "communication family" was evaluated and discarded precisely because each of its acts decomposes into representing and recording. Families are primitives of direction; Lets are composed of them.

Daftar is the reference proto-Botlet of the capture family (Chapter 9): its specialty is to **assess** — apply an instrument and record what the student knows, in its two modes, practice and exam (the measurement). The immediate grading of practice mode is feedback on the capture — like a form that validates what is written into it — not action: the learning is executed by the student outside, and the Let changes nothing in the world by its own hand. Two classifications were tried and discarded: "information", because it looked at the byproduct — the results report — and not at the purpose; and "action", because the transfer of knowledge is not executed by the Let. The membership question settles it: before the Let the system did not know what the student knows; after it, it does.

The `PI` **is not a primitive of the canon**: it is the manifestation of *one* family. The canon stops at `manifestation`; the Information Product is a **normed term of this spec** — not a primitive, but vocabulary with rules — and its governance load is added here without contaminating the canonical level. This is its reference description:

Multi-view Information Product · drill-through. A `PI` is not necessarily a single piece. It can be

composed of **N named pieces**: each **view** is one more piece of the same `PI`, selectable from a picker, with a default view (the first). The `PI` is **authz-blind** — neither the views nor the edges that connect them declare authorization; that policy lives in the policy store, not in the composition.

The connection between views is the **drill-through**: a **navigation edge with context**. A table declares *"on clicking a row, go to the destination view passing that row's key"*; the destination view renders **filtered by that key**. The critical property is **data-anchored / no-bypass**: the context that travels with the edge **narrows within what the viewer can already see** — the destination view applies its own row policy (`RLS`, *row-level security*) over the source, and the context enters as an additional filter, never as an override of the policy (**MUST**). The drill **narrows, never widens** — intersection with what is authorized, never union. If the viewer does not reach the origin row, they do not reach the edge; if they reach it, the destination is still governed by its own policy.

A receivables / balance-aging report illustrates the pattern: named views (Customers, Suppliers, Related parties, Detail) over the same `PI`, a hierarchical Company→Partner table, and a Partner→Detail drill-through edge that opens that partner's documents — filtered by the partner's key and narrowed to what the viewer already had the right to see. The multi-view composition is orthogonal to the operation Botlet's family: what changes is how many pieces compose the manifestation, not its nature.

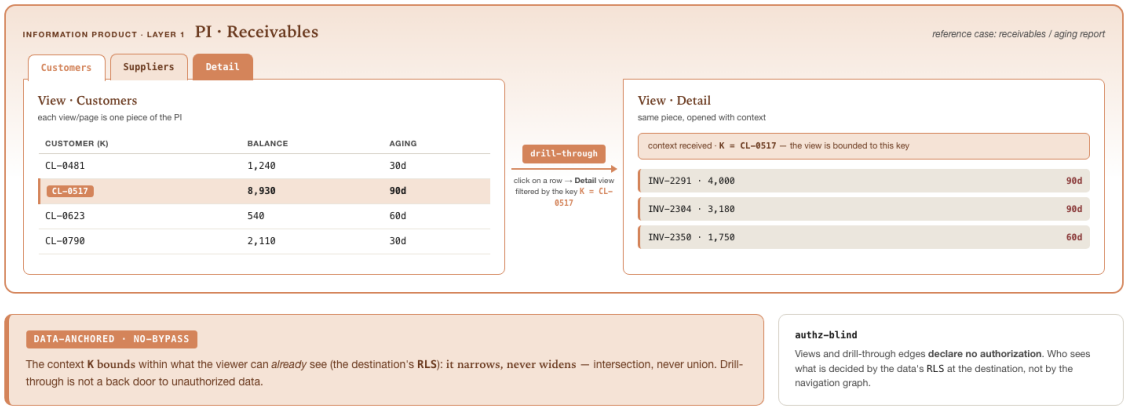


Figure 26: Multi-view PI and drill-through — navigation with context, data-anchored

Temporality is the regime of the manifestation. It is a declared attribute of the Botlet, with two values:

Temporality	How does it manifest?	Relation to the runtime
`discrete`	In pulses: wakes on schedule, trigger, or event, acts, rests	The Botler invokes or schedules; the Botlet does not live between pulses
`continuous`	Sustained: lives persistent and manifests without cease	The Botler sustains execution as long as the Botlet lives

`temporality: continuous` is equivalent to the persistent life of Layer 3: it obliges the Botler to sustain the Botlet's execution without restart for each manifestation. A conforming Botlet with continuous temporality MUST be sustainable by the persistent runtime; a Botlet with discrete temporality manifests via schedule, trigger, or event.

The operational consequence is strong: **real time is not chosen in a delivery channel.** It is not obtained by marking a channel as `push`; it is obtained by giving the Botlet continuous temporality, which in turn obliges the persistent runtime. The delivery mode is the symptom; continuous temporality is the cause. This relocates "real time" from the channel level to the Botlet level, and connects with the distinction *Online enterprise ≠ Real-time enterprise*: they are not two classes of information, but two points on the continuum of temporality.

From here follows a runtime economy. A report — a snapshot at a point in time — and a live dashboard are not two distinct types of *what*: they are the **same manifestation under different temporality**. That is why a single runtime covers both: one builds the hardest case (continuous) and the simple cases are degenerate configurations of that case, not separate codepaths. The distinction holds precisely because temporality is orthogonal to what manifests.

Botler — the framework runner

Botler is the infrastructure that executes the **Lets** — the proper name of the genus of the packaged units of Layer 3 (Autonomy): the Botlets this section develops and the Agentlets that §7 formalizes as their sibling species. It is invisible to the user and to the agent; it is the responsibility of the AgencyDomain's implementation. The canonical relation is simple and is stated over the genus: an AgencyDomain process contains a Botler, and the Botler manages N Lets that live within that process — **1 Process = 1 Botler + N Lets**. In this section, where the Lets are Botlets, the relation instantiates as 1 Botler + N Botlets.

The Botler provides four critical functions. The first is **execution isolation** — sandboxing appropriate to the environment, which we detail in the next section. The second is **management of the Botlet's lifecycle**: invocation when needed, monitoring during execution, failure detection, triggering regeneration when appropriate. The third is **communication with cognition** when the Botlet detects a failure or change in the environment that exceeds its capacity to handle. The fourth is **traceability**: every Botlet invocation, every result, every failure, every regeneration is recorded in the Trust Layer's append-only log. This traceability is what allows reconstructing, auditably, what the agent did and why — and it is indispensable for governance.

The Botler as an abstraction matters because it decouples the isolation implementation from the agent that uses it. The agent does not know — nor does it need to know — whether its Botlet runs in a Docker container, a WASM sandbox, or a microVM. It requests execution from the Botler; the Botler executes under the isolation model the AgencyDomain's implementation chose. This separation is what lets

the spec be agnostic to isolation technology — different implementations choose different technologies according to their specific tradeoffs.

The Botler is generic by definition

The Botler **does not understand the domain** of the Botlets it executes. It manages the lifecycle, isolation, and execution of *any* Botlet without knowing what that Botlet does or what discipline it belongs to. All domain specialization lives in the Botlets and in their proto-Botlets, never in the runtime that hosts them. The architecture is flat: a generic runtime hosts self-contained specialist components.

From here follows a structural property: **no Botler subtypes exist by family of operation.** There is no "informational" Botler, no "transactional" one, no "for information artifacts" one — an informational-operation Botlet already carries its own freshness, its cache, its distribution, so a Botler subtype that duplicated it would contradict the runtime's genericity without adding anything.

> *Botler subtypes are distinguished by deployment topology and role — central, edge, operational facade for Botlets invocable from Layer 1 —* ***never by domain****. Domain specialization lives entirely in the Botlets that the Botler executes.*

The axes of topology and role are legitimate because they answer *where* the runtime runs and with what autonomy, not *what* domain it executes: a central Botler and an edge Botler differ in connectivity and offline operation, not in business knowledge. The normative note is the boundary: any Botler distinction that appeals to the family of operation it executes is ill-posed.

The Botler validates by orchestrating, not by executing

The registration of a Botlet requires its spec to be valid against the declared type before accepting it. This poses an apparent tension: if the Botler does not understand the domain, how does it validate a spec whose meaning is domain-specific? The answer distinguishes **orchestrating** the validation from **executing** it.

The Botler **enforces** the validation; it does not execute it with domain knowledge. At registration time it invokes the validation point the Botlet itself provides — or that its proto-Botlet provides in G1 — hands it the generic context it does control (the catalog of available Capabilities, the identity, the AgencyDomain's policies), and acts on the verdict: it accepts, rejects, or records the result in the append-only log. The judgment of what makes a spec valid lives entirely in the Botlet; the Botler decides whether to admit it or not according to a verdict it did not produce. Thus no invalid spec gets in — the rejection happens before admitting the Botlet and remains traced — without the runtime ever interpreting the domain.

This pattern has a sibling in the invocation of Capabilities. The Botler is the only point through which a Botlet invokes Capabilities, and bypass through parallel channels is not forbidden by policy alone: it is made structurally impossible. On each invocation the Botler hands the Botlet a **controlled handle** — an object with access to Capabilities and to the log bound to the Botler itself — instead of letting the Botlet build accesses on its own. The Botlet can only act on the world through that handle. Both cases share the principle: the generic Botler exposes **control points** and the specialist plugs into them; the runtime stays thin without giving up the guarantees the contract requires.

The Layer 2 ↔ Layer 3 interface via `MCP`

The Botler is the only point of execution for Botlets, and that includes the operation that cognition itself directs: **Cognition** (the `LLM` agent, Layer 2) commands the Botler (Layer 3 runtime, without agency) over

an **internal** interface whose natural transport is `MCP` — the Botler exposes `MCP` servers and Cognition is the client. This interface **is not `A2A`**; the formal correction of the `A2A` nomenclature and the Layer 2 ↔ Layer 3 interface are developed in Chapter 5 §1.

Source code vs spec · two surfaces · one Botlet per `PI`

The operation of a Botlet involves two things worth not confusing. One is the Botlet's **source code**: its implementation. For a platform proto-Botlet, that source code is the **engine**, shared by all the Botlets instantiated from it. The other is the **spec**: what specializes the Botlet's behavior to its instance. The spec is not the code; it configures it.

That separation projects onto **two surfaces** of management:

Surface	What does it manage?	Granularity	Cadence
Source-code lifecycle	Install, version, load, and unload the engine	Botler-level	Releases (Product)
Operation	Specialize, manifest, consume, and control each Botlet	Per-Botlet	Fluid (Instance)

Operation **includes configuring the spec**. For a platform proto-Botlet, the spec is the operational input: there is no "operating" without a spec. Hence a principle: **configuring the spec is operating**. The agent evolves the spec by operating — the `specialize` verb — fluidly; the crystallization of the spec into a versioned registry **follows, it does not precede**, and provenance is given by the append-log. The operation API verbs are `specialize`, `invoke`, and `schedule` (for discrete temporality), `read` and `subscribe` (for continuous temporality), and `status`, `activate`, `deactivate`, `retire`.

From this follows the granularity rule: **one Botlet per `PI`** over a shared engine. Each Information Product is its own Botlet — its own service, with its own identity, temporality, maturity, and fallback — specialized from the shared engine (the platform proto-Botlet). It is not one Botlet with N configurations, which would lose that independence; it is not N programs, because the engine is one. It is the relation `1 Process = 1 Botler + N Botlets`, with the Botlets as specialized instances of the same proto-Botlet. The rationale is `RISC`: many simple, focused Botlets — one per `PI` — compose better than a monolith. The three Botlet tests, formulated earlier in this chapter, reach the same place from the other side: the Information Product executes, has a specialty, and has a life.

The `subscribe` consumption is the bridge to the **agentive north**. Today a human consumes it — a browser subscribed via `SSE` that receives the stream of manifestations of a continuous Botlet — tomorrow Cognition consumes it, feeding on that same stream to decide. Continuous temporality, the persistent runtime, and the `subscribe` verb are, together, the infrastructure of that north.

Isolation model — sandboxing

Botlets are **code generated dynamically by an agent**. This demands strict isolation. A badly written Botlet, or a Botlet generated by an agent that fell victim to prompt injection, could attempt malicious actions — exfiltrating data, modifying system files, opening unauthorized network connections. Isolation is what contains those risks.

The spec admits four sandboxing strategies, with their trade-offs:

Processes with seccomp offer low isolation and minimal overhead. This strategy is useful only in controlled environments where the risk of malicious code is low — for example, Botlets running inside the private perimeter of an organization with implicit trust in its own agents. It is not an adequate strategy for Botlets that touch sensitive data or that operate under a public regime.

Containers — Docker, Podman, equivalents — offer medium-high isolation with medium overhead. It is the most practical strategy for generic Botlets that need to invoke operating-system tools or networks. Containers have a mature ecosystem, reasonable portability, abundant operational tooling. They are a reasonable default for most cases.

WASM (WebAssembly) offers high isolation with low overhead, but the ecosystem is more limited. WASM is ideal for pure transformational Botlets — calculations, data transformations, algorithmic logic — that do not need access to the underlying operating system. The startup speed is very fast (milliseconds), which matters when an agent needs to invoke many Botlets simultaneously.

MicroVMs — Firecracker, Kata Containers — offer maximum isolation with high overhead. They are suitable for Botlets that handle highly sensitive data or that operate in shared multi-tenant environments where the risk of cross-tenant leakage is unacceptable. The overhead typically means tens of milliseconds of additional startup, which in some cases is prohibitive.

The canonical recommendation for reference implementations is **hybrid**: WASM for pure transformers (without system access), containers for generic Botlets that need to invoke operating-system tools, MicroVMs for Botlets that handle highly sensitive data or that operate in shared multi-tenant environments. The specific choice for each Botlet depends on its risk profile and its performance requirements.

Language of the Botlets

The spec is **agnostic to the implementation language** of the Botlets. Cognition can generate Botlets in any language as long as the Botler can execute them within the chosen sandbox. This agnosticism matters because the language landscape evolves — a system tied to a specific language can become obsolete when that language loses traction in the AI ecosystem.

The practical recommendations for reference implementations are:

Python is the first choice. The maturity of the ecosystem, the libraries for almost any integration, the highly reliable LLM generation — contemporary models generate Python correctly with very high frequency — make Python the reasonable default for generic Botlets.

JavaScript / TypeScript are suitable for Botlets that touch HTTP APIs or that automate browsers. The npm ecosystem has broad coverage, and LLM generation is also reliable.

Bash or shell are suitable for Botlets that orchestrate operating-system commands. LLM generation of bash is reliable for simple cases, less reliable for complex cases where bash syntax has quirks.

Rust or Go — compiled languages — are suitable for very-high-frequency Botlets where interpreter overhead matters. LLM generation is less reliable than Python, but the resulting Botlets execute faster. This combination makes sense when a Botlet is going to execute millions of times and the millisecond difference per invocation accumulates into material impact.

There is an important property: the Botlet **is not editable by humans**. It is regenerated by cognition. This matters because any manual improvement to the Botlet's code — a human who opens the file and improves the logic — becomes debt the moment cognition regenerates it due to an environment change.

Regeneration eliminates the human improvements. That is why the spec defines that Botlets are **read-only for humans in production**: if a human wants to improve the logic, they must improve cognition or the Capabilities, not the Botlet directly.

Pattern Recognition — the entry to the cycle

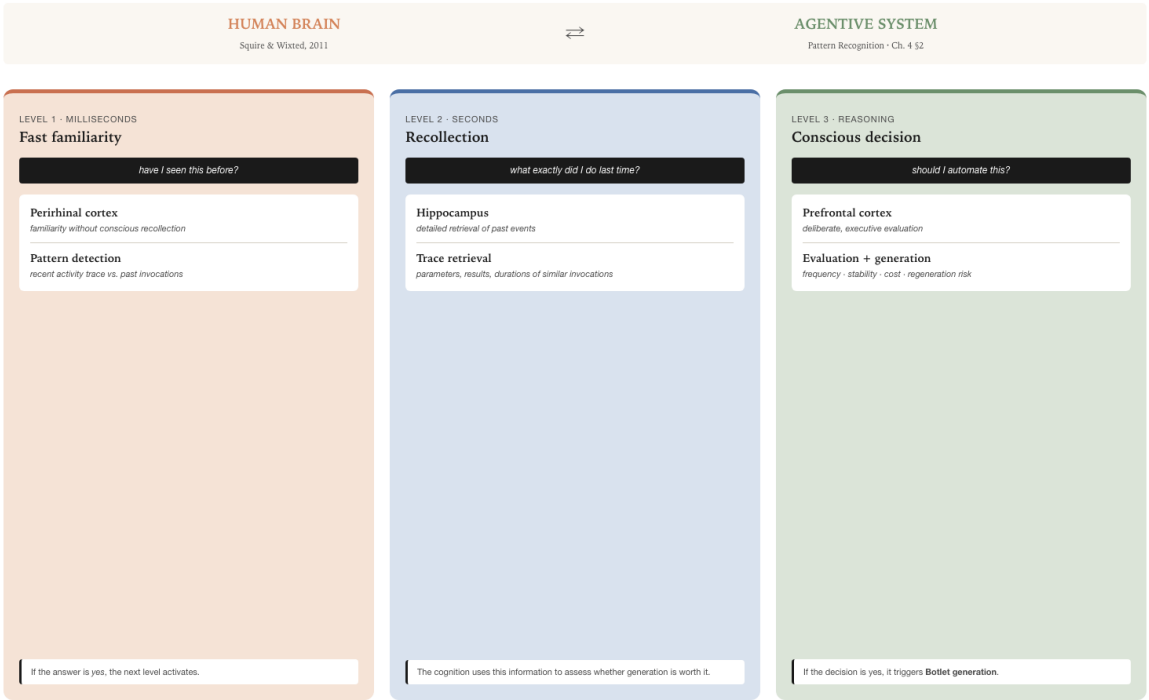


Figure 27: Neurobiological inspiration · human brain ↔ agentive system

Botlets are not generated at random. Cognition decides to generate a Botlet when it recognizes a **repetitive pattern** in the agent's activity. The component that detects those patterns is **Pattern Recognition** — an auxiliary primitive of Layer 2 that we already mentioned in Chapter 4.

Pattern Recognition operates on the trace of the agent's activity: what tasks it did, with what frequency, with what similar inputs, with what results. When a pattern crosses a threshold — the same task repeated more than N times with bounded variability in its inputs, the results stable — cognition evaluates whether it is worth generating a Botlet.

The neurobiological inspiration we mentioned at the outset is reflected in the design of Pattern Recognition. The **perirhinal cortex** of the human brain implements rapid familiarity — the "have I seen this before?" that occurs in milliseconds, without conscious recollection. Agentive Pattern Recognition implements the analog: rapid detection of tasks similar to past tasks, without the need for deep reasoning. If the answer is yes, the system activates the next level.

The **hippocampus** implements recollection: the "what exactly did I do last time?". Pattern Recognition retrieves the specific trace of past invocations, with their parameters, their results, their durations. This information is what cognition uses to decide whether generating a Botlet is worthwhile.

The **prefrontal cortex** implements conscious decision: the "should I automate this?". Here cognition evaluates the criteria — frequency, pattern stability, cost of continuous cognition, regeneration risk — and decides. If it decides yes, it triggers the generation of the Botlet.

Three stages, three levels of processing. Pattern Recognition is not a single step — it is a hierarchical process that filters from "potentially interesting" down to "worth automating". The specification of Pattern Recognition as a Layer 2 tool is out of scope for this book; it suffices to keep in mind that it is the auxiliary primitive that activates the Botlet cycle.

Botlets and the economy of autonomy under subscription

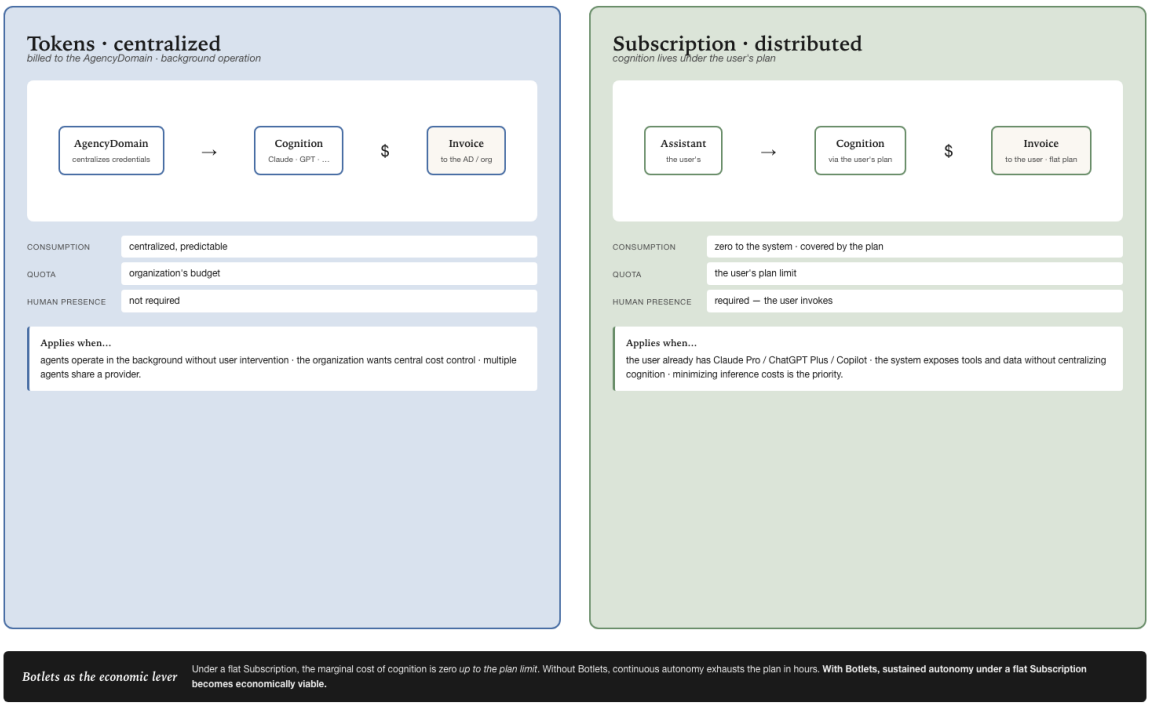


Figure 28: Tokens vs. Subscription · two modes of access to cognition

A critical operational property of the Botlet, which we already introduced in Chapter 4 §2 but which deserves development here, is its role in the economy of autonomy under fixed Subscription plans — Claude Pro, ChatGPT Plus, Copilot enterprise.

In these plans, the marginal cost of invoking cognition is **zero up to the plan's limit**. The user pays a fixed monthly fee and can invoke the model as many times as they want, until the plan exhausts its quota. This economic structure is very different from pay-per-token models, where each invocation costs and the economy is calculated by real usage.

The problem is that the limit exists. An Autonomous Agent operating continuously in the background, without Botlets, would invoke cognition thousands of times a day — each decision, each validation, each action would consult the model. Under a fixed Subscription plan, the agent would exhaust the user's quota in hours. Continuous autonomy would be economically impossible.

Botlets are the architectural mechanism for extending autonomy without saturating the plan. An agent that executes its everyday work via Botlets — and only invokes cognition when the environment changes — can operate in continuous background without exhausting the user's quota. Cognition remains available for real reasoning, not for repetitive tasks that traditional code executes better.

Without Botlets, sustained autonomy under a fixed Subscription is economically impossible.

This is what makes the Botlet an **economic lever**, not merely a technical optimization. The difference between an agent that costs two hundred dollars a month to operate and one that costs twenty dollars a month to operate, with identical effective capability, is almost always the proportion of tasks it executes via Botlets versus via continuous cognition. An organization that adopts Botlets disciplinedly can operate agents at an order of magnitude less cost than an organization that invokes cognition for every operation.

The commercial consequence is direct: providers that deliver agents with good Botlets can offer competitive pricing and reasonable margins; providers that depend on continuous cognition for every operation face an economic dilemma — either they raise pricing to levels the market does not accept, or they operate with negative margins. This pressure is probably what will lead most serious providers to adopt Botlet architectures in the next two to three years.

Derivation chain and catalog of proto-Botlets

A system's Botlets do not appear loose: they are derived from what the system needs to do, and they rest on pre-forged pieces from the catalog. The spec fixes that **derivation chain** as a structural relation:

1. **Documented use cases** — each case requires…
2. **Necessary Botlets** (zero, one, or several; some cases are resolved by cognition without a Botlet) — each Botlet is an instance of…
3. **Required proto-Botlets from the catalog**.

The chain is structural, not methodological. A use case may require no Botlet at all — cognition resolves it directly — require one, or require several; each Botlet that does exist is an instance of some proto-Botlet from the catalog. From here a required property: every conforming Botlet **MUST** be traceable in this chain, and the append-only log **MUST** record the origin proto-Botlet of each instantiated Botlet. The *method* by which use cases are discovered and Botlets are derived does not enter the canon — it is one of several possible ones and lives in the complementary bodies of each implementer; what the canon fixes is the relation and its traceability.

The lower end of the chain — the proto-Botlets — accumulates in **common catalogs** that produce network effects. Each implementer that consumes a proto-Botlet contributes to its maturation: new variants, tested configurations, refinements. The more implementers consume it, the more cases it covers and the more reliable it becomes; implementer n+1 receives versions refined by implementers 1 through n. A proto-Botlet belongs to a catalog community under one of these modes:

Membership mode	What is it?
Private contract	Closed catalog between client and provider
Proprietary codex	Private catalog that an implementer curates (ucodex is an exemplar)
Open public catalog	AgencyDomains.org: any implementer consumes and contributes

Membership mode	What is it?
Sovereign agreement	AgencyDomains that adopt common standards without a direct commercial contract

The modes coexist without tension: one and the same implementer can consume the open public catalog and, on top of it, curate a proprietary codex with its refinements from real cases. The derivation chain and the common catalog are, together, what makes proto-Botlets an economy and not a collection of isolated pieces.

Conformance

A Botlet implementation conforming to this specification must satisfy the following requirements:

Requirement	Level
Code generated by cognition, not written by humans	**MUST**
Execution without invoking cognition in normal operation	**MUST**
Fallback guarantee to manual cognition on failure	**MUST**
Traceability: each invocation recorded in the append-only log	**MUST**
Isolation appropriate to the environment	**MUST**
Automatic regeneration on environment change	**SHOULD**
Pattern Recognition as the trigger of generation	**SHOULD**
Language-agnostic (not tied to a single language)	**MUST**
Recognition of maturity phases (junior, learning, senior)	**SHOULD**
Distinction between seed origin and emergent origin	**SHOULD**
Traceability of the maturity trajectory in the append-only log	**MUST**
Botlet traceable in the use-cases → Botlet → proto-Botlet chain	**MUST**
Origin proto-Botlet of each Botlet recorded in the append-only log	**MUST**
Generic Botler: no Botler subtypes by domain	**MUST**
Botler enforces spec validation without executing it with domain	**MUST**
Support for both temporalities (`discrete` and `continuous`)	**MUST**
Persistent runtime that sustains continuous-temporality Botlets	**MUST**

Capabilities

When an experienced finance consultant faces a new problem at a client, they do not start from scratch. They bring along an organized body of know-how that they apply to the particular case: accounting principles, regulatory frameworks, analytical frameworks, professional practices that the industry has consolidated over decades. When an operations consultant tackles a supply-chain problem, they do the same with their own body of knowledge: SCOR, Six Sigma, lean manufacturing, demand planning. Every professional discipline has its tree of know-how, and the consultant's quality depends in good part on how deep and well organized that tree is in their head.

AI agents face the same problem. An agent operating in finance needs to know finance; an agent operating in legal needs to know legal; an agent operating in marketing needs to know marketing. But "knowing" in this context does not simply mean that the underlying model read finance documentation during training. It means that the agent has modular, composable access to an organized body of professional know-how that it can apply selectively according to the task. This organization is what the Agentive Architecture calls **Capabilities**.

This section develops the concept of Capability as the canonical primitive of Layer 2 (Cognition). What makes the Capability special with respect to other nearby concepts in the field is its **hierarchical structure**, its **composability**, and its explicit distinction from plugins and prompts. An agent without Capabilities has monolithic cognition that conflates domains; an agent with well-organized Capabilities operates with the discipline of a professional who knows the distinctions of their field.

Definition

A **Capability** is a unit of specialized know-how that an agent understands and applies. It encapsulates the procedural and declarative knowledge of a domain: what is known, how it is applied, what is decided, what is asked when data is missing. Capabilities reside in **Layer 2 (Cognition)** of the Agentive Architecture. Cognition selects and applies Capabilities according to the task — in the same way that an expert consultant selects and applies professional frameworks according to the problem in front of them.

> *The agent does not know what the model knows. It knows what its Capabilities allow it to know.*

The quotation above distinguishes what Layer 2 knows from what the underlying model knows. The underlying model — Claude, GPT, Gemini — was trained on a massive amount of information that includes, among many other things, professional knowledge. But the agent that lives in Layer 2 **does not operate directly with the model's knowledge**. It operates with the Capabilities assigned to it, which are specific curations of know-how applied to particular contexts. An agent that has the *General Ledger* Capability but not the *Tax* Capability answers with authority on general accounting but knows that it is not the right source for tax questions. An agent without specific Capabilities operates with the model's diffuse knowledge, which may be correct in general but is rarely precise on the professional plane.

What is NOT a Capability?

To fix the definition with the precision the spec demands, we contrast the Capability with neighboring concepts that the industry uses carelessly. Each of these neighboring concepts has its legitimate role in AI systems, but none is a Capability, and conflating them leads to architectures that end up being poorly integrated collages.

The term **Capability** is reserved, in the strict sense, for the **cognitive know-how** of **Layer 2 (Cognition)**: knowledge that interprets, decides, and reasons about a domain. This reservation matters because other agentive deliverables — connections to source systems, presentation makings — live in other layers, have a different nature, and are built with their own development schemes. Treating them all as Capabilities would force a qualifier onto the term at every use and dilute its value. The section *Capability, Connector, and Template — three deliverables per layer* fixes the proper terms for each layer.

A **plugin** lives in the context of a host application. It extends that application with a new function. It is tied to the host: the Excel plugin only operates within Excel, the Notion plugin only operates within Notion. A Capability is not tied to a host — it lives in the agent's Capability tree and is invocable from any context where the agent operates.

A **prompt** is a specific linguistic formulation injected into the model on a particular invocation. Prompts are useful, but they are a transient artifact: they change from invocation to invocation, they do not encode persistent knowledge. The Capability can use prompts internally as one of its implementation mechanisms, but it does not reduce to a prompt. A well-designed Capability includes vocabulary, procedural knowledge, declarative knowledge, heuristics, and references to tools — the prompt is only one of the components.

A **system prompt** establishes the model's general tone and behavior. It is configuration of the mode of operation, not modular know-how. Capabilities **are not system prompts** — they can coexist with one, but they capture something else: domain knowledge applicable according to the task, not the agent's global configuration.

A **tool** lives in Layer 4 (Access). It is an executable action on the external world: invoking an API, reading a file, sending an email. The Capability **decides which tool to invoke and how to interpret its result**, but it is not the tool itself. The Capability is knowledge; the tool is action. A finance agent with the *Treasury* Capability knows when and how to invoke the tool that queries bank balances — but the tool itself is a separate component that lives in another layer.

A **skill**, in the field's popular terminology, is a term the industry uses so carelessly that it has practically lost meaning. Various vendors call "skills" plugins, prompts, tools, configurations — without precise distinction. The Capability is not a skill in the generic sense, but the partial overlap of popular usage may confuse the reader. To avoid the confusion, this book reserves the term Capability with a precise definition and avoids using "skill" except when quoting sources that use it.

Three traits distinguish a genuine Capability. **Modular**: it can be activated or deactivated independently of the others. If an agent has *Finance* and *Legal* Capabilities and *Legal* is withdrawn, the agent keeps operating coherently with *Finance* alone. **Composable**: multiple Capabilities can coexist in a single agent and combine when the task crosses domains. An agent that has *Finance* and *Operations* can handle a logistics case with financial components without tripping in the transition. **Knowledge, not action**: the Capability knows; the tools (Layer 4) act. The Capability decides which tool to invoke and how to interpret its result.

Capability, Connector, and Template — three deliverables per layer

An agentive delivery project rarely delivers only a Capability. The cognitive Capability is the protagonist, but it usually comes accompanied by deliverables that live in other layers and are built with different schemes. The spec canonizes three terms, one per layer, so that each deliverable is named for what it is without qualifying the term Capability.

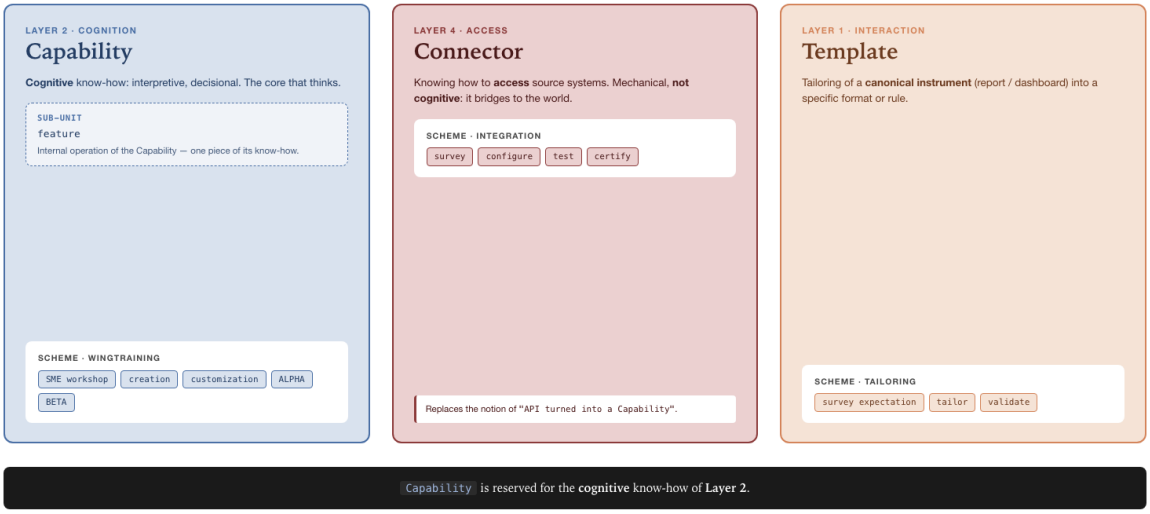


Figure 29: Three deliverables per layer — Capability (Layer 2) · Connector (Layer 4) · Template (Layer 1)

A **Capability** (strict sense) is **cognitive** know-how — interpretive, decisional. It lives in **Layer 2 · Cognition**. It is the project's protagonist.

A **Connector** is knowing how **to access source systems** — a connection with execution power. **It is not cognitive knowledge.** It lives in **Layer 4 · Access**. The Connector is what the pre-agentive field knows as the technical integration that touches the external world. A legacy API, when brought into the Agentive World, becomes a **Connector** (Layer 4), not a Capability.

A **Template** is the **client-specific making** over a canonical instrument — a report or dashboard that the Capability produces — in a particular format or rule required by the client. It lives in **Layer 1 · Interaction**, alongside the Facet, the surface Botlet, and the view Botlet. Anonymized examples: a regulatory template of a basic interconnection offering over the canonical cost report; a managerial monthly financial-close format over the canonical profitability dashboard. The Template is not a Capability, not a Facet, and not a Botlet: the buyer already knows the term "template" from the pre-agentive vocabulary and does not need to learn a new one.

How is each deliverable developed?

Each term has its layer, its nature, and its own development scheme:

Deliverable	Layer	Nature	Development scheme
Capability (protagonist)	Layer 2 · Cognition	Cognitive, interpretive, decisional know-how	Wingtraining 5 steps (SME workshop · creation · customization · ALPHA · BETA)
Connector (companion, if it requires access to source systems)	Layer 4 · Access	Knowing how to access systems; not cognitive knowledge	Integration scheme (survey · configure · test · certify)
Template (companion, if the delivery must conform to a presentation expectation)	Layer 1 · Interaction	Making of a canonical instrument in a client format or rule	Making scheme (survey the expectation · make over the canonical instrument · validate)

Wingtraining, cited in the table, is the canonical development scheme for a Capability in five steps: **workshop with the SME** (*subject-matter expert* — the human expert whose knowledge is transferred to the agent), **creation**, **customization**, **ALPHA** (validation with the SME), and **BETA** (validation in real operation). Its development belongs to the delivery practice, not to this canon; the classification test below uses it as a criterion because it is the operational proof that a scope is a transfer of knowledge — and not an integration or a format.

What is the conceptual structure of an agentive delivery project?

An agentive delivery project is structured as a **protagonist Capability** (Layer 2) typically accompanied by one or more **Connectors** (Layer 4) and one or more **Templates** (Layer 1). In addition, the Capability itself produces, with no extra effort, its **information instruments** — canonical reports and dashboards

—, which remain implicit in the delivery: they require no development scheme of their own because they emerge from the Capability and its interaction layer. The Template appears only when one of those instruments must conform to a specific client form.

Capability or deliverable of another layer? — the test

To decide whether a scope is a Capability in the strict sense, the following three tests must all be **yes**:

1. **Is it cognitive know-how?** Does it interpret, decide, or reason about a domain — beyond merely connecting or merely formatting?
2. **Does it have an identifiable SME?** Is there a human expert whose knowledge is transferred to the agent?
3. **Does it pass through the five Wingtraining steps without forcing?** Does it make sense to apply SME workshop · creation · customization · ALPHA · BETA to it?

If one or more are **no**, the scope is not a Capability: it belongs to another layer — Connector or Template, per the mapping in the section *Capability, Connector, and Template* — or, if it is cognitive knowledge but subordinate to a larger Capability, it is a **feature** of that Capability (see the next section).

The hierarchical structure — the tree of knowledge

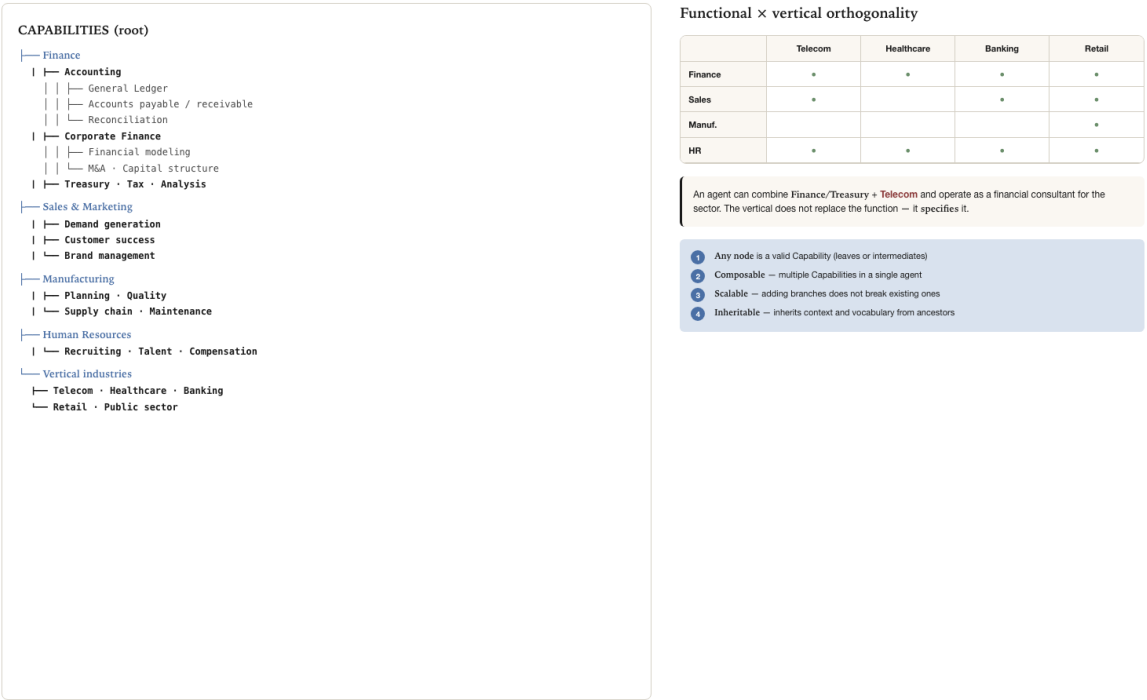


Figure 30: The tree of knowledge + functional × vertical orthogonality

Capabilities are organized in a **hierarchical tree**. The structure is not decoration — it is the natural way professional knowledge is organized in any serious discipline. Financial analysts think hierarchically: finance has sub-disciplines (accounting, corporate finance, treasury, tax), each sub-discipline has areas

(general ledger, accounts payable, reconciliation), each area has specific practices. The hierarchical organization reflects how the professional masters the field.

Below is an example of the canonical tree the spec proposes as a starting point. The tree is not exhaustive — it extends according to the needs of each implementation —, but it illustrates the structure (figure above).

Four structural rules govern the tree. The first: **any node is a valid Capability**. Both leaf nodes — *Reconciliation*, *Demand generation*, *Quality* — and intermediate nodes — *Accounting*, *Sales & Marketing*, *Manufacturing* — are valid Capabilities. Granularity is decided by context. An agent specialized in daily accounting operation adopts leaf nodes with high specificity; an orchestrator agent that coordinates several specialists adopts intermediate nodes with a panoramic view; a multi-specialist agent combines several roots.

The second rule: **it is composable**. Capabilities can be shared across agents. A *Customer success* Capability can live in a commercial team's agent and simultaneously in a support team's agent, without duplication. This is operationally critical — the organization does not need to re-build the knowledge for each new agent, but rather assigns already-built Capabilities according to the agent's role.

The third rule: **it is scalable**. New branches are added without breaking the existing ones. The *Telecom* Capability can rise in sophistication — adding sub-branches *Network APIs*, *5G core*, *Customer experience* — without the other branches of the tree changing. This property is what allows the tree to grow with the organization: each time the organization enters a new domain, it adds branches; each time it deepens an existing domain, it extends branches. The tree never needs a complete rewrite.

The fourth rule: **it is inheritable**. A Capability inherits context and vocabulary from its ancestors. A *General Ledger* Capability does not need to re-explain what Accounting or Finance is — that contextualization comes from the tree. This matters in implementation because it allows leaf Capabilities to be more concise: they only need to describe what is specific to their node, not what is already implicit in the tree path.

Anatomy of a Capability

The canonical specification of a Capability includes nine components, which we lay out one by one. Components one and nine are metadata; components two through seven are the body of the know-how; component eight is the binding with the action layer.

The first component is **identity**: canonical name plus position in the tree. Identity is what makes the Capability referenceable — agents invoke it by name, policies are applied to specific Capabilities, logs record which Capability was invoked.

The second component is **vocabulary**: technical terms of the domain that the agent must recognize and use correctly. An agent with the *General Ledger* Capability must distinguish debit from credit without hesitation, must know the difference between manual and automatic entries, must use the concept of fiscal period correctly. Vocabulary is what allows the agent to converse with domain professionals without sounding amateur.

The third component is **procedural knowledge**: how the typical tasks of the domain are done. How an account is reconciled. How an invoice is validated. How a monthly close is prepared. Procedural knowledge is sequential and operational — it describes steps, not just concepts.

The fourth component is **declarative knowledge**: verifiable facts and rules of the domain. That ac-

counts payable have due dates. That revenue is recognized when certain criteria are met. That bank reconciliation must occur monthly. Declarative knowledge is factual and permanent — it describes what is true of the domain.

The fifth component is **heuristics**: professional decision rules of the domain. When an invoice deviates more than five percent from the typical amount, escalate. When a reconciliation has more than ten unmatched transactions, suspend the process and escalate. When a supplier has three claims in six months, flag for review. Heuristics are professional judgment encoded — they capture not what is true of the domain, but what decision a professional would make in each situation.

The sixth component is the **associated tools**: Layer 4 tools that the Capability typically invokes. The *General Ledger* Capability invokes tools that query the accounting database, that post entries, that generate reports. The Capability declares which tools it uses so that the agent knows what it needs available when it exercises that Capability.

The seventh component is the **parent Capabilities**: the hierarchical position from which the Capability inherits context. A *General Ledger* Capability declares that it is a child of *Accounting*, which in turn is a child of *Finance*. The declaration of parents is what enables the inheritance described in the previous section.

The eighth component is the **maturity state**: Draft / Active / Deprecated. The maturity state is operational metadata — it lets the organization manage the lifecycle of its Capabilities. A Capability in Draft is still being validated; an Active one is approved for productive use; a Deprecated one is legacy that is kept for compatibility but should not be used in new agents.

The ninth component is **version**: traceability of evolution. Capabilities change over time — professional practices evolve, regulations change, accumulated knowledge is refined. Versions let the organization track the evolution of knowledge and let different agents operate with different versions according to their requirements.

Components two through five — vocabulary, procedural, declarative, heuristics — are what **deeply differentiates** a well-built Capability from an elaborate prompt. A prompt gives the model context transiently, for a single conversation. A Capability encodes professional knowledge with persistence and modularity. The difference is structural: the agent can consult the *General Ledger* Capability without the accounting knowledge flooding all its other conversations — it can at the same time have the *Marketing* Capability without marketing frameworks contaminating the accounting rigor. This separation is what allows agents to operate with the discipline of a professional who switches frameworks according to the task.

Features of a Capability

A Capability exposes internal operations. The spec canonizes the term **feature** to name them — the practical equivalent of what other vocabularies call *feature*, *operation*, *skill*, or *method*. A feature is an internal operation that the Capability exposes; it shares with the containing Capability its data model, its SME, its installation, and its runtime. A single costing Capability, for example, exposes as features cost allocation, profitability, and *pricing*: three dimensions of a single know-how, not three Capabilities.

Capability or internal feature? — the test

A scope is treated as its own Capability if and only if the following three tests are **yes**:

1. **Operational independence?** Can it be installed and operate without the other capability?

2. **Cognitive identity?** Does it have a data model and SME distinct from the other capability?
3. **Reusability?** Does it have value for more than one consumer or context outside this case?

If one or more are **no**, the scope is a **feature** of the containing Capability, not its own Capability.

ID convention

An implementation MAY adopt an identifier convention to trace the tree of deliverables: the Capability (or deliverable) carries the ID `E<n>` —`E1`, `E2`—; the feature carries the composite ID `F<n>.<m>` —`F1.1`, `F1.2`— where `<n>` is the ID of the containing Capability. The convention is optional; what is canonical is that the feature construct has a name.

What pathologies does the test prevent?

Distinguishing feature from Capability prevents two pathologies observed in real projects:

- **Inflation of the deliverable codomain** — dimensions of a single Capability are documented as separate Capabilities, multiplying the inventory and diluting portability. The costing example already cited —allocation, profitability, and *pricing* treated as three Capabilities when they are features of a single one— is the typical case.
- **Hiding lock-in** — scopes that really are their own Capabilities, with independent lifecycle, data model, and reusability, end up embedded as "sub-capabilities" of another, hiding that they could be installed and ported separately.

The two tests are applied in order: the first decides whether the scope is cognitive at all; only then does the second decide its granularity within Layer 2. If the scope is not cognitive, it is classified directly as Connector or Template by its layer, without running the feature test.

How does cognition operate over Capabilities?

Given an agent with a set of active Capabilities, when the agent receives a request, the processing follows a canonical seven-step flow.

Step one: the user's request reaches the agent. The agent receives it in natural language — *"reconcile last month's revenue account"*, for example.

Step two: cognition classifies the domain of the request. This operation is semantic routing — the agent identifies that the request is about accounting, specifically reconciliation. This identification is based on the vocabulary that the active Capabilities provide to the agent.

Step three: cognition selects the relevant Capabilities. In the example's case, *Finance/Accounting/Reconciliation* is the most specific applicable Capability. Cognition selects it together with its ancestors — Accounting, Finance — to have all the inherited context.

Step four: cognition applies the procedural and declarative knowledge of the selected Capabilities. It knows what steps make up a reconciliation, what rules it must follow, what typical errors it may encounter.

Step five: cognition invokes the tools the Capability indicates. It queries the accounting database for last month's entries, queries the bank statement, runs the matching process.

Step six: cognition composes the response using the domain's vocabulary and the Capability's heuristics. It reports the matches found, the unmatched items, the recommendations on how to proceed with the discrepancies. It uses terminology that an accounting professional would recognize as correct.

Step seven: if the request is repetitive — the agent has already done similar reconciliations in the past —, Pattern Recognition suggests generating a Botlet that automates the cycle one through six for future similar requests. Cognition evaluates the suggestion and, if conditions are appropriate (high frequency, stable pattern), generates the Botlet.

An important observation: the Capability **does not execute** — it is executed by cognition. This matters structurally: the same Capability can be applied with different depth — fast and superficial, or slow and exhaustive — according to the agent's cognitive model and mode of operation. An agent with sophisticated cognition can apply the *General Ledger* Capability with all the rigor of an experienced controller; an agent with more limited cognition can apply the same Capability with the depth of a junior. The Capability defines the know-how; cognition defines how deeply it is applied.

Industry verticals as root Capabilities

The tree includes a root dedicated to **industry verticals** — Telecom, Healthcare, Retail, Banking, Public sector. This is deliberate: each vertical has its own vocabulary, its own regulations, its own canonical processes. The existence of a vertical root separate from the functional roots (Finance, Sales, Operations) reflects a structural property of professional knowledge: verticals and functions are **orthogonal**.

A generalist finance consultant has functional knowledge — Finance — but no specific vertical knowledge. A finance consultant for telecom has both: Finance (the function) and Telecom (the vertical). The telecom consultant can apply generalist financial frameworks, but also knows the particularities of the sector — the revenue-assurance models specific to telecom, the regulations of the sector's regulator, the operational systems typical of a carrier. Vertical knowledge is **additional**, not a substitute, to functional knowledge.

The Capability tree reflects this orthogonality. An agent can simultaneously have *Finance/Treasury* and *Industry verticals/Telecom* Capabilities — and the combination produces an agent that operates with functional and vertical knowledge at the same time, exactly like the sector's expert consultant.

This architecture has two important consequences. The first: **vertical specialization is not a prompt — it is a Capability**. A "legal agent" or "medical agent" or "telco agent" is not a sophisticated prompt of the general model. It is an agent that loads the corresponding vertical Capability, with its own vocabulary, heuristics, and normative knowledge. This distinguishes Capabilities from generic System Prompts: vertical knowledge is modular, persistent, and composes with other non-vertical Capabilities without contamination.

The second consequence is commercial. This architecture explains the commercial success of the **vertical specialists** of today's market: Cursor (coding), Harvey (legal), Jasper (marketing), Fin (customer support). What these products sell is not "a specialized GPT" in their vertical — it is a robust vertical Capability that cognition applies with confidence. The difference from a generic GPT is not marketing; it is structural. The user who tries Cursor for programming does not notice that the difference is the *Coding* Capability the tool loads; they notice that the answers are correct more often than with a generic GPT, and that is exactly what the well-built Capability produces.

> *The vertical Capability is the difference between a useful agent and an agent serious about the domain.*

Capabilities Marketplace

An emergent property of the architecture is that Capabilities admit a **market**. A well-built Capability can be distributed across AgencyDomains, versioned and maintained by a specialized provider distinct from the AgencyDomain operator, charged by subscription or license, and audited by third parties as to its correctness and completeness.

This gives shape to a **Capabilities economy** analogous to the open-source software package economy. Whoever builds and maintains expert Capabilities — for example *General Ledger* IFRS-compliant, or *Telecom 5G core* — can operate as a specialized provider without building agents or AgencyDomains of their own. Their product is the Capability itself. Their business model is license or subscription of the Capability. Their clients are organizations that operate AgencyDomains and need verified professional know-how.

The emergent economy has clear precedents. The software industry has had similar economies of specialized components for decades: compiled libraries sold or licensed, certified modules for specific frameworks, best-practice configurations that consultancies sell as assets. The Capabilities economy would be the natural evolution of those models to the agentive field.

The normative specification of the **Capabilities Marketplace protocol** — package format, versioning model, signature system, charging model — is **open work** in version 1.0 of this book. Contemporary implementations may adopt ad-hoc packages; consolidation as an industry standard is pending. When consensus arrives, a future version of the book will incorporate it as normative spec.

Locality and availability — operational classification of Connectors

The canonical description above treats know-how as applicable in any context. The operational reality of systems with multiple physical presence — restaurants with locations, bank branches, retail stores, industrial plants — requires an additional classification that the spec formalizes explicitly: **locality** and **offline availability**. The classification belongs to the **Connectors** — the Layer 4 face of the pair that accompanies the Capability —, because what resides somewhere and needs (or doesn't need) a network is the *access*, never the knowledge: the Capability that decides when and how to invoke has no locality. It is documented in this chapter because the Capability–Connector pair is delivered and reasoned about together. Without this classification, decisions about what can be invoked from an edge Botlet in offline mode are made in the dark.

The classification operates over **two orthogonal axes**:

Locality axis

Where the Connector's components physically reside:

- **Cloud-resident** — the Connector lives in a remote service. Canonical examples: Connector `DTE-SII` (Chile's SII service for issuing electronic receipts and invoices), `Transbank-Onepay` (bank gateway), `Stripe-Connect` (payment processing). The agent invokes them over the network; without network there is no access.
- **Edge-resident** — the Connector lives at the physical site, associated with local hardware or systems. Canonical examples: Connector `ESC/POS-Printer` (thermal printer for tickets and receipts with the ESC/POS protocol connected by USB or serial), `Cash-Drawer` (the cash drawer of the till), `Local-Pinpad` (card pinpad connected to the POS), `Sensor-Temperatura` (cold-room sensor

connected by GPIO). The agent invokes them against the site's hardware; they need no network to operate.

- **Hybrid** — the Connector has a local component and a cloud component. Canonical examples: Connector `Client-DTE` (signs the document locally, queues it if there is no network, sends it to the SII when the network returns), `Client-Pinpad-Deferred-Processing` (authorizes locally with PIN and batch, sends to the acquirer when the network returns). The local part operates offline; the cloud part synchronizes when there is network.

Offline availability axis

Whether the Connector can execute without network:

- **Online-only** — requires network to execute. Without network, the invocation fails. Cloud-resident Connectors are typically online-only in the strict sense, although some have variants with a local client that turn them hybrid.
- **Offline-capable** — executes without network. If its external contract eventually requires cloud communication (a voucher that must reach the SII, a transaction that must consolidate at headquarters), it **queues** and emits outward when the network returns. Edge-resident Connectors are typically offline-capable; hybrid Connectors are too, by design.

Canonical classification matrix

Each conformant Connector explicitly declares its position in the matrix:

	Online-only	Offline-capable
Cloud-resident	DTE-SII (without local client) · Transbank Onepay · weather API	(unusual combination; typically migrates to hybrid)
Edge-resident	(unusual combination)	ESC/POS-Printer · Cash-Drawer · Sensor-Temperatura · Local-Pinpad
Hybrid	(unusual combination)	Client-DTE · Client-Pinpad-Deferred-Processing · Sync-Inventario

Connection with distributed Layer 3

The classification is structurally necessary when Layer 3 is distributed (Chapter 5 §1). An edge Botler must **know** which Connectors it can invoke offline. If it does not know, its edge Botlets will attempt to invoke cloud-resident Connectors without network and will fail catastrophically — without network, not even the agentic fallback applies, because cognition lives in the cloud.

The operational rule the classification enables is direct: **a senior edge Botlet, at a physical site without network, operates by invoking exclusively edge-resident Connectors and the local part of hybrid Connectors**. The cloud-resident ones and the cloud part of the hybrids remain temporarily inaccessible; the deferred effects (sending to the SII, consolidation with headquarters) are queued; when the network returns, the queues drain.

Required properties

Property	Level	Description
Explicit declaration of the Connector's locality	**MUST**	Cloud-resident, edge-resident, or hybrid.
Explicit declaration of offline availability	**MUST**	Online-only or offline-capable.
Specification of offline behavior for offline-capable	**MUST**	What it does when there is no network, what it queues, how it drains.
Deterministic resolution of which component runs in hybrids	**MUST**	Under what conditions the local component runs; under which it invokes the cloud.

Capability portability

The previous section classifies *where* the Connector that accompanies the Capability physically resides — cloud, edge, or hybrid. A distinct property, which the spec formalizes explicitly, is **Capability portability**: a conformant Capability can be installed and run on **any conformant AgencyDomain**, without rewriting. This portability is what makes the Capability **real property of the client** — not of the AgencyDomain that hosts it, nor of the hosting that sustains that AgencyDomain.

The argument is the same no-lock-in one the canon makes for the AgencyDomain, applied one level lower. Just as a conformant AgencyDomain migrates to another conformant hosting platform without being held captive by it, a conformant Capability migrates to another conformant AgencyDomain without being held captive by it. The client who acquires a Capability acquires a portable asset, not a rental tied to a platform.

The two portabilities?

It is worth not confusing two portabilities that operate at different levels:

Portability	What migrates?	To where?
AgencyDomain portability	The complete AgencyDomain	To another conformant hosting platform
Capability portability	A single Capability	To another conformant AgencyDomain

What is the Capability ↔ AgencyDomain relation?

The relation is asymmetric and explicit: an **AgencyDomain hosts and runs** Capabilities; a **Capability runs on** a host AgencyDomain. The Capability is a first-order inhabitant of the AgencyDomain's Layer 2 — the know-how that gives cognition to its agents —, not a support resource. This is the reason the canonical definition of AgencyDomain names Capabilities among what the AgencyDomain hosts and runs, on a par with autonomous agents and Botlets.

Regulatory certification resides in the certified component, not in the Botlet

A structural property that appears with force in productive agentive systems in regulated industries — gastronomy, health, finance, retail with DTE, pharmacy, telecommunications — and which the spec needs to formalize explicitly: **the regulatory certification of operations resides in the certified component the Botlet invokes — the certified Connector, with the regulated Capability that carries its normative knowledge —, never in the Botlet that orchestrates**. The separation is necessary because the generated nature of the Botlet makes it impossible to certify a priori, and certifying it a posteriori contradicts its regenerable nature.

The problem

The book defines that cognition generates the Botlet's code (Chapter 5 §2). But some operations a Botlet executes are **regulated**: issuance of DTE under SII norm, card payment under PCI-DSS, pharmaceutical dispensing under sanitary registration, financial communication under the corresponding regulator's norm. For these operations, **the regulation requires certification of the component that executes the operation**. A system that issues an electronic receipt without SII certification is not legal; a system that charges a card without PCI certification cannot operate.

If certification resided in the Botlet, each Botlet that executes a regulated operation would have to be certified individually. But a Botlet is **code generated by cognition** that regenerates when the environment changes. Each regeneration would produce a technically distinct Botlet that would require re-certification. Regulatory certification over Botlets turns the `95/4/1` cycle into an operational impossibility: each 1% change would require a regulatory process.

The canonical solution

The spec resolves the tension by separating responsibilities with discipline, and the division respects the chapter's layer doctrine:

- **The Botlet orchestrates.** It knows the process flow, validates operational pre-conditions (are there products?, is the table open?, does the customer have their tax ID registered?), captures the event, formats the request according to the certified component's contract.
- **The certified Connector executes the regulated operation.** It receives the request from the Botlet, executes under all applicable norms, returns the voucher. The `DTE-SII` Connector receives the sale's detail, signs with the tax certificate, transmits to the SII, receives the folio and electronic stamp, returns the voucher to the Botlet. It is certifiable precisely because it is stable access, not generated knowledge.
- **The regulated Capability carries the normative knowledge.** What the norm demands, how it is interpreted, when a receipt applies and when an invoice, what to do upon a regulator's rejection — the cognitive knowledge with which cognition decides and validates. It lives in Layer 2, is developed with an SME of the regulatory domain, and accompanies the Connector as its pair.

The separation has three structural consequences:

First, certification is of the certifiable component. The `DTE-SII` Connector can be formally certified — its code is stable, its contract with the SII is explicit, its behavior is auditable. Certification is one-time work; it holds for all the Botlets that invoke it.

Second, generated Botlets coexist naturally with regulatory compliance. A `Charge-Table-9` Botlet that regenerates when the kitchen changes its menu does not break the tax certification — it

keeps invoking the same certified `DTE-SII` Connector. The Botlet's regeneration affects orchestration logic, not the regulated operation.

Third, the audit boundary becomes sharp. When the regulator audits, the AgencyDomain exposes: the Botlet (business logic, mutable, regenerable) and the certified component (regulated operation, frozen, auditable). The regulatory inspection concentrates on the certified Connector — where the certification resides —, while the business logic is governed with the Trust mechanisms of Chapter 5 §4 without contradicting the regulation.

Canonical pattern

The pattern applies to any regulated industry:

Industry	Botlet (orchestrates)	Certified component (executes the regulated operation)
Gastronomy	`Charge-Table`	Connector `DTE-SII` (electronic receipt or invoice)
Banking	`Process-Payment`	Connector `Gateway-PCI-DSS` (tokenization + authorization)
Pharmacy	`Dispense-Prescription`	Connector `Sanitary-Registry` (validation and registration of dispensing)
Telecom	`Activate-Service`	Connector `Subtel-Registry` (regulatory registration of activation)
Health	`Issue-Prescription`	Connector `MINSAL-E-Prescription` (certified medical signature)

The pattern is uniform: the Botlet contains the mutable business logic; the certified component contains the frozen regulated operation; the regulated Capability supplies the normative knowledge with which cognition governs the whole. The boundary between them is the boundary between what the organization can freely regenerate and what it must keep under certification.

Required properties

Property	Level	Description
Regulated components declare their regulatory regime	**MUST**	Which norm it complies with, before which regulator, with what certification number.
Certified Connectors are immutable between audits	**MUST**	The certified Connector's code does not regenerate; it changes only under a regulatory process.
Botlets may invoke certified components without restriction	**MUST**	The component's contract is stable; the Botlet invokes it like any other.

Property	Level	Description
Auditability of the boundary	**MUST**	The log clearly distinguishes Botlet operations (business logic) from certified-component operations (regulated operation).

Capabilities and Botlets — the relation

Capabilities and Botlets live in different layers and solve different problems, but they interact in a structured way. The Capability lives in Layer 2 (Cognition). It is know-how. The Botlet lives in Layer 3 (Autonomy). It is learned doing. The Capability has the form of vocabulary plus procedures plus heuristics plus tools. The Botlet has the form of traditional executable code. The Capability is persistent and versioned. The Botlet is self-regenerable and ephemeral. The Capability is applied when the agent recognizes its domain. The Botlet is invoked when the agent recognizes the pattern. The Capability is created by human experts or by specialized providers. The Botlet is created by the agent's cognition, automatically.

The canonical interaction is the following: a Capability can give rise to multiple Botlets. The agent that applies *General Ledger* repeatedly for a specific company begins to generate Botlets that automate the routine steps of the process — classifying transactions, reconciling accounts, generating monthly reports — without invoking cognition. The Capability remains the same; the Botlets are the efficient residue of its repeated application in a particular context.

This relation is what allows the agentive system to scale economically. Capabilities are the stable knowledge that the organization acquires, maintains, evolves. Botlets are the efficient residue that cognition generates by applying Capabilities repeatedly in specific contexts. The organization invests in Capabilities; the Botlets emerge from use. The investment in knowledge generates savings in operation.

Conformance

An implementation of Capabilities conformant to this specification must satisfy the following requirements:

Requirement	Level
Hierarchical tree structure	**MUST**
Any node is a valid Capability	**MUST**
Composability across Capabilities	**MUST**
Anatomy with vocabulary + procedural + declarative + heuristics	**MUST**
Explicit versioning	**MUST**
Declared maturity state (Draft / Active / Deprecated)	**MUST**
Selection by cognition, not direct execution	**MUST**
Verticals as a dedicated root	**SHOULD**
Open marketplace	**MAY** (when the normative spec exists)

Requirement	Level
Explicit declaration of the companion Connector's locality (cloud / edge / hybrid)	**MUST**
Explicit declaration of offline availability	**MUST**
Portability across conformant AgencyDomains	**MUST**
Regulated components (normative Capability + certified Connector): regime declared	**MUST**
Certified Connectors: immutability between audits	**MUST**

Evolution frontier

Three active areas of evolution of the Capability primitive deserve mention.

The **Capabilities marketplace** is the first. The normative protocol is not yet consolidated; when it is, version 2.0 of this book will incorporate it.

Cross-vertical Capabilities are the second. How a Capability can be combined with multiple verticals without contradictions — an agent operating in finance for banking and for telecom simultaneously, for example — requires refinement of the inheritance mechanisms that version 1.0 of the spec describes.

Capability auditing is the third. How to certify that a Capability does what it says it does — that an *IFRS Compliance* Capability effectively reflects IFRS and not its informal approximation — is a governance problem that the field has not yet solved. The likely solutions will come from the side of traditional professional auditing adapted to the agentive context.

Trust Infrastructure

There is an operational asymmetry that any organization which has tried to bring agentive AI to production recognizes with discomfort: what works in a pilot rarely works in enterprise production. A controlled pilot, with a handful of sophisticated users, closely supervised by the team that built it, can run successfully without explicit governance. Pilots demonstrate technical capability, not operational fitness. The distance between the two is exactly what this section develops.

The figure that most worries the field in 2026 is Gartner's projection: more than forty percent of agentive AI projects will be cancelled before the end of 2027. The reasons Gartner identifies are three: unforeseen costs, unclear business value, and inadequate risk controls. The third reason — inadequate risk controls — is the one that connects directly with the content of this section. Organizations cancel projects not because the technology does not work, but because the organization cannot defend what the technology does when something goes wrong. The difference between a successful pilot and a cancelled project is typically the maturity of the **trust infrastructure**.

Trust Infrastructure is the set of cross-cutting properties that allow an organization to **trust that its agents operate with autonomy without losing control**. It is not an additional layer of the Agentive Architecture — it is a property that cuts across the four existing layers (Interaction, Cognition, Autonomy, Access) and is exercised at different points depending on the specific pillar. This section develops the five pillars with the detail the architect reader needs to design and the executive reader needs to evaluate.

> *Trust Infrastructure is not what you add after the agent works. It is what separates pilots from production.*

The urgency of Trust Infrastructure is no longer only architectural. It is regulatory. Singapore's IMDA published in January 2026 the first state framework for governance specifically targeting agentive AI. The European Union does the same with the EU AI Act. NIST with its AI Risk Management Framework. ISO/IEC with standard 42001. The question is no longer whether regulators will demand trust infrastructure — it is whether the organization can demonstrate it auditably when asked. Organizations that do not have it operational will face, over the horizon of the next twenty-four months, a binary decision: invest at speed to reach conformity, or suspend agentive operations in regulated markets.

The five pillars

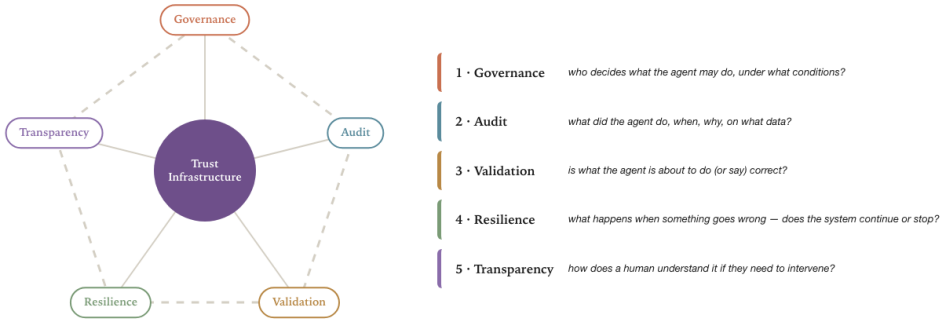


Figure 31: The five pillars of Trust Infrastructure

Five pillars constitute Trust Infrastructure. Each answers a specific operational question the organization needs to be able to answer when someone — an auditor, a regulator, a client — questions it.

Pillar	Question it answers
Governance	Who decides what the agent may do, under what conditions?
Audit	What did the agent do, when, why, over what data?
Validation	Is what the agent is about to do (or say) correct?

Pillar	Question it answers
Resilience	What happens when something goes wrong — does the system stop or continue?
Transparency	How does a human understand it if intervention is needed?

Each pillar is exercised in one or more layers and is materialized with concrete mechanisms. The next five subsections develop each pillar in detail. Afterward we show the cross-cutting map — which pillar operates mainly in which layer — and we close with the regulatory reading of the field.

Pillar 1 — Governance

The Governance pillar defines the set of mechanisms by which the organization establishes **what** the agent may do, **under what conditions**, and **with what level of supervision**. It is the most visible pillar to someone coming from the traditional IT world, because it has the most direct equivalent in known mechanisms — IAM, SSO, RBAC. But agentive Governance is structurally distinct from traditional governance, and conflating the two is a recurrent source of failed projects.

Traditional Governance asks *"who can see what data?"*. The subject of control is a human with a stable identity; the object is a discrete resource. The question is static: permissions rarely change, and when they change it is by explicit human event (someone was hired, someone was promoted, someone left the company). Agentive Governance asks *"what can an agent do, under what conditions?"*. The subject is an agent acting autonomously; the object is a sequence of actions the agent can execute with varying degrees of impact. The question is dynamic: conditions change with context, with the moment, with the state of the system. Traditional Governance tools — IAM, SSO, RBAC — are insufficient for this model. They work well for human subjects with stable identity; they do not work for agents that act continuously with varying degrees of autonomy.

The canonical mechanisms of agentive Governance are four. **Configurable policies** are declarative rules — not embedded code — that define which tools the agent may invoke, over what data, at what times, with what impact thresholds. The separation between policy (declarative) and code (imperative) matters: policies must be changeable without a system redeploy, must be versionable independently of the code, must be auditable without requiring a code review. **CRUDLEX permissions** — Create, Read, Update, Delete, List, Execute — are a granular model of permissions over tools and data, applicable by user, agent, or context. The full operationalization of CRUDLEX lives in Chapter 8. **Human approval for critical operations** establishes that the agent may execute low-impact operations autonomously, but high-impact ones stop and request approval. The definition of "high impact" is policy, not technical — the organization decides which thresholds trigger approval. The **AI registry** is a formal inventory of which agents are active, which Capabilities apply, which tools they are authorized for, who approved them. It is what a regulator will see when auditing, and what the organization must keep updated and accessible.

The field data regarding Governance is raw — Chapter 2 documents it. The story they tell is consistent: most organizations know the problem exists but have not invested enough to solve it, and regulators are on their way to forcing the investment.

Pillar 2 — Audit

The Audit pillar defines the capacity to reconstruct, after the fact, **what the agent did, when, why, and over what data** — with enough fidelity for forensic analysis, regulatory compliance, or contractual dispute. It is the pillar the organization needs when something goes wrong: if an agent made a decision that produced a bad outcome, or executed an action a client questions, or carried out an operation the regulator wants to examine, the organization needs to be able to reconstruct auditably what happened.

The canonical mechanisms of Audit are four. The **immutable append-only log** is the central component: every agent action is recorded in a log that admits no retroactive modification. It is only appended to — never edited nor deleted. The log is typically chained cryptographically: each record contains the hash of the previous one, forming a verifiable chain where a retroactive alteration would be immediately detectable. The **trace of each action** records the agent's identity, the capability invoked, the tool executed, parameters, result, timestamp, context. Each action generates a complete record that reconstructs the state of the system at the moment of the decision. The **decision lineage** is the causal chain of reasoning that led to a particular action: what information the agent consulted, what Capabilities it applied, what heuristics it used, what cognition it evaluated. Without lineage, an action appears as an isolated event; with lineage, it appears as the result of a reconstructible reasoning process. **Per-action identity tagging** ensures that each action is unequivocally attributable to an identifiable agent, not to "the system" or to "the AI". The distinction matters for accountability: when something goes wrong, it must be possible to identify which specific agent was responsible, and by extension, who configured it, who approved its capabilities, what policy covered its operation.

ISACA — the professional association for audit — notes that agentive AI presents a growing challenge for audit functions because its decision processes lack clear traceability when the system was not designed with audit in mind. The observation is important because it captures a structural property: an agent that combines probabilistic cognition (LLM) with deterministic tools makes decisions whose path is hard to reconstruct if the log is not designed to do so. A log that only records "the agent executed X tool with Y parameters" is not enough — it also needs to record "because the cognition evaluated Z reasoning based on W context". Agentive Audit demands explicit design from the start. It does not emerge naturally from an agentive architecture — it is built with discipline from the start.

> *Agentive audit demands explicit design. It does not emerge naturally from an agentive architecture — it is built with discipline from the start.*

Pillar 3 — Validation

The Validation pillar defines the capacity to verify that the agent's response or action is **correct** before it affects the world. It is the most recent pillar to mature as a category — the AI validation industry is of the last three years — and, simultaneously, the most critical for use cases where the cost of error is high.

The difference from traditional validation is important. Traditional validation verifies formats: is the JSON valid? does the date have the correct format? is the amount a number? Agentive validation verifies **semantics**: is the agent telling the truth? is it acting within the reasonable limits of the domain? is it exposing data it should not? Traditional validation operates over structure; agentive validation operates over meaning.

The canonical mechanisms of Validation are five. **Hallucination detection** verifies that the agent's factual claims are consistent with the sources consulted. It is an active area of research; contemporary mechanisms include self-consistency (asking the same thing in several ways and comparing responses),

retrieval-augmented verification (consulting authoritative sources before asserting), and model-as-judge (a second model evaluates the first's response). **Structured-response validation** verifies that outputs with a schema — JSON, XML, tables — comply with the expected contract before being emitted. It is the most direct validation, equivalent to traditional validation but applied over outputs the model generated. **Prompt injection prevention** detects manipulation attempts through malicious inputs disguised as legitimate data. A user who tries to inject instructions into a comments field so the agent executes them as if they were legitimate instructions is a common attack that validation must detect. Products such as Lakera and Lasso Security productize precisely this mechanism. **DLP** — Data Loss Prevention — automatically detects personal data (PII), sensitive financial information, or classified material in places where it should not appear. If an agent is about to include a social security number in a response to an external user, DLP detects and blocks it. **Tokenization** replaces sensitive data with tokens before they reach the model. It allows the agent to reason over the data without exposing it to the external cognition provider. The organization keeps the token-to-data mapping in a hardened store — typically an HSM (Hardware Security Module) or dedicated service.

Informatica formulates with precision the transition that Validation represents: *"Because agents act without human approval loops, the data they use must be fully trusted, verified, and monitored."* The phrase captures something important. In traditional systems, a human supervises each important operation before executing it — it is the last loop of validation, made of flesh. In agentive systems, that human loop does not exist on every operation — only on critical operations that escalate. Validation has to **stand in for the human loop** for all the other operations, ensuring that what the agent is about to do is correct before doing it. Without that substitution, the system falls short: either it executes incorrect actions, or it requires human supervision on every operation, nullifying the agent's productivity.

The validation of a specialist component's spec admits a structural pattern that deserves note: **validation by delegation**. The generic Layer 3 runtime (the Botler) **enforces** the validation of a Botlet's spec without understanding its domain — it orchestrates the validation point the Botlet or its proto-Botlet provides, hands it the generic context it controls (Capability catalog, identity, AgencyDomain policies), and audits the verdict in the append-only log. The judgment of what makes the spec valid lives in the specialist; the runtime demands and records the validation without executing it with domain knowledge. The development of this pattern lives in the Botlets chapter; here we only link the principle to the Validation pillar.

With the **Agentlets** (Chapter 5 §7), the Validation pillar also gains a seat in **Layer 3**: the bounded inference of every Agentlet passes through the `cognition_call` control point of the Botler's handle, and there the mechanisms of this pillar apply — hallucination detection over its verdicts, structured-output validation against the charter's contract, DLP and tokenization over what enters and leaves the model. The statistical correction that Agentlets introduce into the muscle-memory layer thus stays confined to declared, auditable Lets — the thinness discipline the Bounded Concerns Architecture teaches, exercised inside the Agentive World.

Pillar 4 — Resilience

The Resilience pillar defines the guarantee that the system keeps operating — and the organization retains control — when something goes wrong. It is the pillar closest to traditional software engineering practices — the field of DevOps and SRE has developed resilience patterns for fifteen years — but adapted to the particularities of the agentive system.

The canonical mechanisms are five. The **fallback guarantee** is the fundamental property we already described in Chapter 5 §2 (Botlets) and that §7 extends to the Agentlet: if a Botlet fails catastrophically,

the cognition executes the task manually; if an Agentlet cannot resolve within its charter, it escalates to full Cognition; if the cognition fails, the operation escalates to the human. This guarantee is what distinguishes the agentive system conforming to this spec from any fragile "AI automation". **Structured error handling** ensures that errors are typed, actionable, propagated with enough context for the next level to decide. An error that says "something failed" is not structured handling; an error that says "API X returned code Y, parameter Z, in operation W of agent V" is structured handling. **Sandboxing** ensures execution isolation of the Layer 3 units and of generated code, with strict limits on what they can touch. Detail in the Botlets section. **Circuit breakers** stop and notify when an agent or Botlet fails repeatedly, before continuing to consume resources. It is a classic resilience pattern adapted to the agentive context: if a Botlet has failed N consecutive times, its automatic execution is stopped and the matter escalates to the human for review. **Rate limiting** establishes configurable limits on the frequency of invocations, both to the cognition (controlling cost) and to external tools (protecting downstream systems). Without rate limiting, an agent with a badly designed loop can exhaust the system's resources in hours.

The non-stopping principle the spec guarantees — declared in Chapter 5 §2 — is what allows the organization to delegate operation to agents with the confidence that an isolated failure does not stop the business. Resilience is what makes that confidence reasonable.

Operational business continuity vs agentic fallback guarantee

The fallback guarantee that Pillar 4 describes assumes **cognition available**: when the Botlet fails due to an environment change, the cognition rescues. This assumption holds in most scenarios — the cognition lives in a highly available cloud and Botlets fail occasionally due to minor changes the cognition resolves effortlessly. But in physical productive systems — premises without network, downed hardware, cut power — **the cognition is not available either**, and operational continuity needs an additional protocol that the agentic fallback guarantee does not cover.

The spec therefore distinguishes **two complementary mechanisms** that resolve distinct scenarios:

Agentic fallback guarantee — the cognition executes when the Botlet fails due to environment changes. It is a property of the Agentive Architecture, codified in the Botlet's spec (§2). It covers the vast majority of failures: the environment changes, the Botlet detects the change, the cognition rescues. This is the property that produces the Botlet's maturity trajectory from junior to senior.

Operational business continuity — documented manual protocols for when the senior Botlet goes down by exogenous causes and the cognition is not available either. It is an operational property, equivalent to the one any traditional business already has when its system goes down (power cut, downed hardware, catastrophic network, critical provider down). It does not depend on the agentive spec — it depends on the client's protocol.

The two are not mutually exclusive: they complement each other. The first resolves the Botlet's learning and most operational failures. The second covers the catastrophic exogenous residue that no system prevents completely.

Connection with the Botlet's maturity

The connection between the two mechanisms and the Botlet's maturity trajectory (§2) deserves explicit treatment because it reveals a structural property of the agentive system:

The agentic fallback guarantee is what produces maturity. Every time the Botlet fails due to an environment change, the cognition rescues, regenerates, and returns operation — and that regeneration is

precisely what produces the progressive incorporation of variants until reaching senior maturity. Without agentic fallback, the Botlet would be trapped in its initial version, with no way to learn. **Without agentic fallback, the Botlet does not mature.**

Operational continuity, by contrast, does not bear on maturity. It operates over already-mature Botlets that go down by exogenous causes, not by pending learning. When a senior Botlet goes down because the premises suffered a power cut, it is not a learning problem — it is an operational problem the continuity protocol must resolve (manual till, paper records, later reconciliation).

Mechanism	When does it operate?	What does it resolve?	Who provides it?
Agentic fallback guarantee	Junior, learning, or senior Botlet with a new variant	Environment changes the Botlet did not anticipate	The agentive spec (Layer 2 + Layer 3)
Operational business continuity	Senior Botlet down by exogenous cause, no cognition available	Operational continuity when no computational component operates	Client's protocol (documented manual procedure)

Why does the distinction matter?

Without the distinction, operational continuity plans get confused with the agentic promise. A client reads "fallback guarantee" in the product documentation and assumes it covers any failure, including power cuts. When the cut occurs and they discover the system does not operate, they attribute the failure to the agentive architecture — and conclude that "the system fails".

Recognizing the two properties as separate but complementary mechanisms reduces anxiety about offline and makes clear what the architecture resolves and what the client's operational protocol resolves. The conversation with the client changes: it is no longer promised that "nothing ever happens"; it is promised that "learning failures are handled by the architecture, exogenous failures are handled by the continuity protocol — and both are documented".

The conceptual distinction between agentic fallback and operational continuity is the seat of this section. Its operationalization — the per-site field protocol, the degradation modes, and the log marks with which the transition to continuity is recorded — lives in Chapter 8, where the operational requirements are developed (almost all of them **MUST**, including the traceability of the transition to continuity mode).

Pillar 5 — Transparency

The Transparency pillar defines the human's capacity to understand, in real time, **what the agent is doing and why** — with enough detail to intervene if necessary. It is the pillar that connects the other four: Governance defines what it may do, Audit records what it did, Validation verifies what it is about to do, Resilience ensures it keeps operating — Transparency ensures a human can understand all of the above.

The canonical mechanisms are five. **Full observability** delivers end-to-end tracing of each operation, metrics of latency, cost, quality, and structured events. It is the agentive equivalent of traditional observability systems (Datadog, New Relic, Splunk) adapted to the particularities of the agentive system. **Operational metrics** measure the success rate of Botlets, regeneration frequency, cognition latency,

token consumption, errors per layer. These metrics are what an operations team consults daily to understand the health of the system. **Human-consultable traces** ensure that the trace of a decision is readable by a technical human, not only by another machine. This matters for reactive audit: when something went wrong and a human needs to reconstruct what happened, they must be able to read the traces directly, not depend on an intermediate automated analysis system. **Proactive alerts** notify when the agent detects that it is near a limit, failing with unusual frequency, or making high-impact decisions. Proactivity matters: the system does not wait for the human to consult in order to report problems; it reports before they become critical. **Governance dashboards** give the human in charge a view: which agents are operational, what they do, with what success, over what resources. It is the control interface for the person who governs the system.

AI observability is a mature market category. Products such as Langfuse, LangSmith, Helicone, Arize AI, Braintrust, Weights & Biases cover distinct layers. Agentive transparency does not demand building these products from scratch — it demands integrating them coherently with the other pillars of Trust Infrastructure. The organization adopts the products that best fit its stack and integrates them with the rest of the infrastructure.

Declarative quality contract

The **Resilience** and **Audit** pillars are exercised best when what the organization must audit and sustain is **declared**, not buried in code. The **declarative quality contract** is the mechanism that enables it: any conforming Botlet MAY declare its quality attributes as structured properties, not as embedded code. Trust Infrastructure reads them uniformly — without coupling to each Botlet's implementation.

The canonical attributes of the contract are five. **Freshness** declares the maximum admissible age of the data the Botlet consumes. The **SLA** declares the expected end-to-end latency, expressed as percentiles (`p50`, `p99`). The **degradation policy** declares the behavior on failure — `refuse` (rejects and delivers nothing), `warn_and_show` (warns and shows anyway), `show_last_valid` (delivers the last known valid result), `agentic_fallback` (escalates to the cognition). **Audience** declares the applicable `RLS`/`CRUDLEX` policy for who may consume the manifestation. The **refresh policy** declares how it is renewed — `on-demand`, `scheduled`, or `push`.

Declared this way, these attributes become cross-cutting mechanisms that serve two pillars at once. For **Resilience**, the degradation policy and the refresh policy are auditable configuration and not fragile logic scattered through the implementation: the runtime knows what to do on failure and when to renew without reading the Botlet's body, and Freshness and SLA give explicit thresholds against which to measure whether a result is still trustworthy. For **Audit**, the five attributes allow Trust Infrastructure to audit them uniformly, run them through the AgencyDomain's **global policies** (a policy can harden the minimum Freshness or veto `warn_and_show` for a class of operation), and report them as comparable **standard metrics** across Botlets. The organization does not invent a format per Botlet; it declares against a common vocabulary that the append-only log and the governance dashboards already understand.

Cross-cutting map — which pillar operates in which layer?

The five pillars are exercised in different layers of the Agentive Architecture, as the figure above synthesizes.

Three readings of the map are useful. The first: **Layer 4 concentrates the greatest load**. Governance, Audit, and final Validation operate mainly in Layer 4. It is coherent with its nature: Layer 4 is where

PILLAR / LAYER	LAYER 1 INTERACTION	LAYER 2 COGNITION	LAYER 3 AUTONOMY	LAYER 4 ACCESS
Governance policies, CRUDLEX, approval	–	–	–	PRIMARY
Audit append-only log, lineage	–	–	–	PRIMARY
Validation hallucination, prompt injection, DLP	–	partial	–	PRIMARY
Resilience fallback, sandboxing, circuit breakers	–	–	PRIMARY	–
Transparency observability, traces, alerts	cross-cutting	cross-cutting	cross-cutting	cross-cutting

absent · partial · primary — pillar operates centrally in the layer · cross-cutting — operates in all layers

1 · Layer 4 carries the heaviest load
Governance, Audit, and final Validation operate primarily in Layer 4 — where cognition becomes real action.

2 · Resilience lives in Layer 3
Operational continuity rests on persistent autonomy — Botler, Botlets with fallback.

3 · Transparency is a cross-cutting property
It spans all four layers. It demands explicit instrumentation design in each one, not added at the end.

Figure 32: Cross-cutting map · which pillar operates in which layer

cognition becomes real action, and therefore where control is exercised. The decisions of what may be done, what is about to be done, and what was done all pass through Layer 4. The second: **Resilience lives in Layer 3**. Operational continuity rests on the agent's persistent autonomy, its Botler, its Botlets with fallback. When the agentive system "keeps working" despite some component having failed, that continuity is ensured by Layer 3. The third: **Transparency is a cross-cutting property**. It does not live in a specific layer — it cuts across all. This demands explicit design of instrumentation in each layer, not added at the end as an additional layer. Each layer emits events, each layer has metrics, each layer contributes to the audit log.

Trust Infrastructure and the regulators

Trust infrastructure is not an architectural decision alone. It is a decision of **regulatory conformity** before a growing framework of regulation specific to agentive AI. The main frameworks the field faces at the start of 2026 are:

The **EU AI Act** of the European Union, in force with gradual application between 2024 and 2026. The **NIST AI Risk Management Framework** of the United States, in force as a voluntary framework but adopted by regulated industries. **ISO/IEC 42001**, published in 2023 with voluntary certification but growing enterprise adoption. The **MGF — Model AI Governance Framework for Generative AI** of Singapore's IMDA, published in January 2026 and notable for being the first state framework specifically for agentive AI (the name keeps generative AI; the body governs agents). The **World Economic Forum guidelines** on agent onboarding and governance. The **NACD guides** — National Association of Corporate Directors — for corporate boards.

A common pattern: all these frameworks demand — in formally distinct but functionally equivalent terms — the five pillars described in this section. It is no coincidence. The list of pillars emerges from a functional analysis of the problem, not from imitation of a particular regulator. Any serious regulator that analyzes the risks of the agentive system arrives at a similar list: governance, audit, validation, resilience, transparency. The equation is structural.

The operational consequence for the organization is that investing in Trust Infrastructure is not only an architectural decision — it is a **regulatory investment**. An organization that adopts the five pillars correctly positions itself to satisfy the four main frameworks simultaneously — EU AI Act, NIST AI RMF, ISO/IEC 42001, IMDA MGF — because they converge on similar functional requirements. An organization that neglects them is left exposed to all four at once, with no structural defense.

Conformance

An implementation of Trust Infrastructure conforming to this specification must satisfy:

Requirement	Level
The five pillars exercised in some layer	**MUST**
Immutable append-only log of every action	**MUST**
Granular CRUDLEX permissions per user / agent / context	**MUST**
Fallback guarantee on failure	**MUST**
Human-consultable traceability	**MUST**
Hallucination detection	**SHOULD**
Prompt injection prevention	**SHOULD**
DLP / tokenization of sensitive data	**SHOULD**
Configurable human approval for critical operations	**MUST**
Conformity with at least one recognized regulatory framework	**SHOULD**
Explicit distinction between agentic fallback and operational continuity	**MUST**
Documented operational continuity protocol for systems with a physical presence	**MUST**

Evolution frontier

Three active areas of evolution of Trust Infrastructure deserve mention at the close.

Auditable audit is the first. Cryptographically verifiable protocols that allow an external auditor to verify the log without access to the system. It sits at the intersection with confidential computing technologies and zero-knowledge proofs. When it matures, it will allow external audit without exposing operational data to the auditor.

Agent trust scoring is the second. Composite reliability metrics that evolve with the agent's behavior, similar to a credit score. It would allow the organization to adopt agents with confidence modulated by their trust score: agents with a high score receive more autonomy; agents with a low score require more supervision.

Trust Infrastructure federation is the third. When two AgencyDomains collaborate (federation, see section 1 of this chapter), how their respective Trust Infrastructures recognize and compose. Does AgencyDomain A's policy also apply to AgencyDomain B's invocations? How are contradictory policies reconciled? It is a problem without a general solution in version 1.0 of the spec.

Assistant vs Autonomous Agent

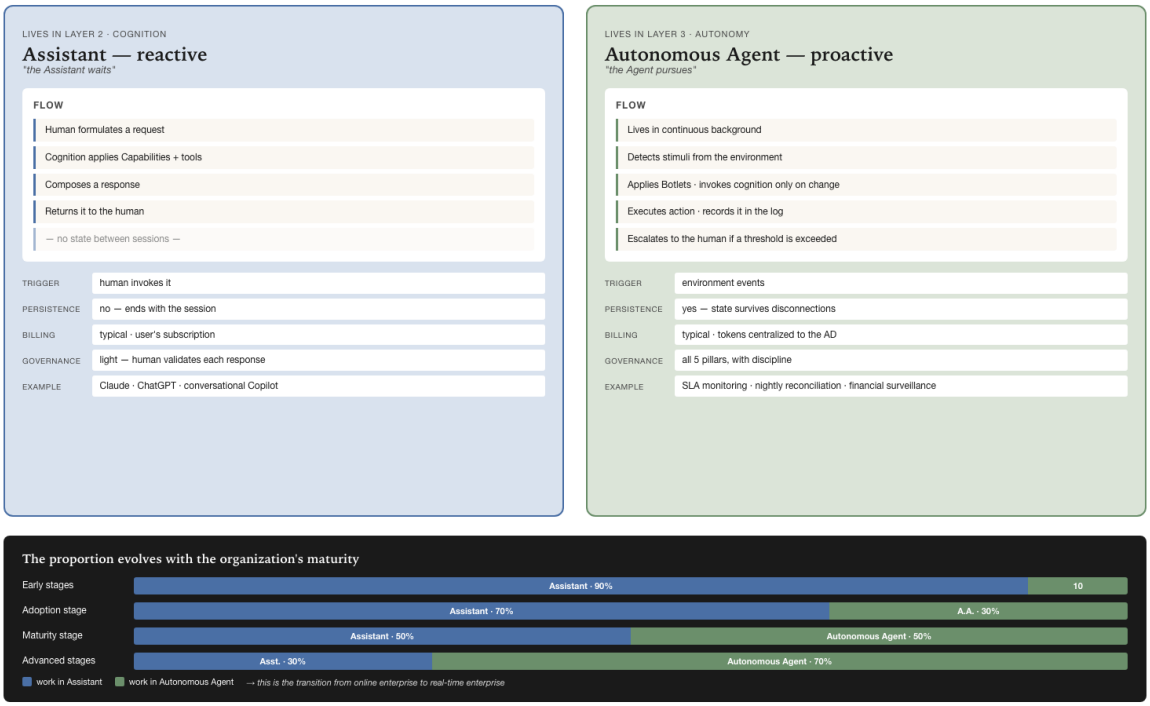


Figure 33: Assistant vs. Autonomous Agent · distinct roles

The industry talks about "AI agents" as if they were a single thing. They are not. Beneath that generic term coexist two distinct operating modes, with distinct purposes, distinct economic models, distinct governance models. Conflating the two is probably the most recurrent source of agentive projects that fail in the move from pilot to production.

The two modes are the **Assistant** and the **Autonomous Agent**. This section develops the distinction with the detail it deserves, because the practical consequences of maintaining it — or of ignoring it — are enormous.

The distinction

The **Assistant** lives in Layer 2 (Cognition). It is **reactive**: it responds when asked, it waits for input, it maintains no Botlets of its own, it has no persistent life between sessions. The user speaks to the Assistant, the Assistant responds, the conversation ends. When the user returns later, the Assistant resumes as if it were the first time — with no memory of the prior exchange except when it is explicitly injected as context. Paradigmatic examples of the Assistant are Claude, ChatGPT, conversational Copilot. They

are mass-market and useful products, but structurally they are Assistants — they wait for someone to speak to them.

The **Autonomous Agent** lives in Layer 3 (Autonomy). It is **proactive**: it acts on its own initiative, it pursues objectives, it maintains Botlets, it regenerates them when the environment changes, it lives in a continuous background. The user does not speak to it — the Autonomous Agent operates. When the user queries the Autonomous Agent, it is not because the conversation is just beginning, but because the Agent has been operating for hours or days and the user wants to know its status. Paradigmatic examples of the Autonomous Agent are the bots that monitor anomalies on a network, the processes that run nightly reconciliations, the agents that watch SLAs and escalate when they approach violation. They are less visible than Assistants — they do not appear in consumer applications — but structurally they are where most of the agentive system's economic value is generated.

> *The Assistant waits. The Agent pursues.*

The canonical phrase captures the operational difference well. The Assistant is a shift worker: it shows up when called, it responds, it leaves. The Autonomous Agent is a permanent worker: it lives in the system, it monitors continuously, it executes when warranted, it escalates when necessary.

The distinction is **not hierarchical**. An Autonomous Agent is not an upgraded Assistant. They are **distinct roles** with distinct purposes. A mature agentive system contains **both modes** and composes them. The organization that operates only Assistants falls short because its agents cannot operate in the background; the organization that operates only Autonomous Agents falls short because it cannot handle conversational human requests. Serious systems need both.

The umbrella term **Agent** covers a third member beyond these two modes: the **Agentlet** — the packaged agent of bounded charter, living as a catalog unit in Layer 3 alongside the Botlet. It is not a third mode on the same axis — the modes describe how the full agent operates; the Agentlet is a piece the full agent engenders and maintains. Its formalization, and the border that separates it from the Autonomous Agent, live in §7 of this chapter.

Why does the distinction matter?

Three concrete operational reasons justify the attention this section devotes to the distinction.

The first reason is that **they are designed differently**. The Assistant is designed for conversation. Latency must be perceptible to the human — seconds, typically —, the interface is textual or by voice, the mode is turn-response. When the human closes the conversation, the Assistant ends. The Autonomous Agent is designed for persistent life. Latency is operational — minutes or hours if useful —, with no direct interface to the human except when it escalates, and continuous monitoring of environment events. When the human disconnects, the Agent continues.

An architecture that mixes both modes without distinction produces confused systems: Assistants with Botlets the human does not understand, or Autonomous Agents that require constant human attention to operate. An Assistant that, in the middle of a conversation, launches a Botlet in the background without notifying the user breaks the user's expectation. An Autonomous Agent that cannot execute anything until the human opens the application loses the point of its autonomy. The explicit distinction in the architecture prevents these confusions.

The second reason is that **they are billed differently**. The Assistant typically lives under a **user subscription** model — the human pays for their Claude Pro, their ChatGPT Plus, their Copilot. Cognition is invoked during conversations. For the agentive system, this means that every Assistant operating

under the user's subscription has available quota the system does not pay for directly. The Autonomous Agent typically lives under the AgencyDomain's **centralized tokens** model — the organization pays the aggregate consumption of the agents operating on its behalf. For the agentive system, this means predictable but material costs: every decision, every validation, every action of the Autonomous Agent consumes tokens the organization must pay for.

This economic difference determines when each mode is preferable. If the use case allows the human to be in the loop, the Assistant under the user's subscription is typically cheaper — the cost is absorbed by the user's plan. If the use case requires continuous autonomy with no human present, the Autonomous Agent under tokens is the only viable option, but with a predictable cost the organization must budget for. Conflating the modes leads to economic errors: Autonomous Agents accidentally operating in subscription mode that exhaust the user's quota in hours, or Assistants accidentally operating in tokens mode that bill the system for what should run against the user's subscription.

The third reason is that **they are governed differently**. The Assistant operates under the human's immediate control. Validation is conversational: the human reads the response before acting, judges whether it is correct, decides what to do with it. Governance is light — basic permissions suffice. If the Assistant makes a mistake, the human notices it immediately and corrects it. The Autonomous Agent operates **without** the human's immediate control. Validation must be systemic: the organization trusts that the agent acts correctly when no one is watching. Governance is robust — it requires the five pillars of Trust Infrastructure operationalized with discipline. If the Autonomous Agent makes a mistake, the human discovers it when they see the log or when the consequence materializes, not in the moment.

> *Selling an Autonomous Agent with the governance of an Assistant is selling a risk dressed up as a product.*

This is the structural reason why products that promise "autonomous agents" but Assistant governance fail in enterprise production. The organization buys expecting autonomy; it receives products that need constant human supervision. The resulting frustration is what feeds the wave of cancellations Chapter 2 documents.

Operational anatomy

We lay out the operational flow of each mode to make the distinction concrete.

The Assistant

The Assistant's flow is linear and conversational. The human formulates a request. Cognition — Layer 2 — receives it. Cognition applies the relevant Capabilities to understand the domain of the request. If it needs additional information, it invokes tools. It composes the response. It returns it to the human. The human keeps conversing or closes the session. When the human leaves, the Assistant does not persist — unless the system implements explicit memory (which is a feature, not default behavior).

What is characteristic of the Assistant is that it **does not operate when the human is not present**. Cognition is available on demand; when there is no demand, there is no activity. This is efficient for conversational use cases but is a severe limitation for cases where the work needs to run at predictable moments or in response to external events.

The Autonomous Agent

The Autonomous Agent's flow is continuous and proactive. The agent lives in the background. It detects stimuli from the environment — changes in data, alerts, events. When a stimulus triggers a response, it applies the relevant Botlets. If the Botlet executes successfully, the operation ends. If the Botlet fails or the case is new, the agent invokes cognition to resolve it. It executes the corresponding action — it invokes a Layer 4 tool. It records everything in the append-only log. If the operation crosses impact thresholds defined by policy, it escalates to the human. It returns to waiting for the next stimulus.

What is characteristic of the Autonomous Agent is that it **lives persistently**. Its life is independent of any human session. The agent operates while the organization operates, not only when someone speaks to it. This demands supporting infrastructure — state persistence, continuous monitoring, active governance — but it produces operational capacity the Assistant cannot deliver.

How do they cooperate in a mature system?

A mature agentive system **contains both modes** and composes them. The typical composition works like this: the Autonomous Agent operates continuously in the background executing objectives; the Assistant handles human requests that typically query the Autonomous Agent's status or request adjustments to its operation.

When a CFO asks their Assistant *"what is the cashflow status this week?"*, the Assistant recalculates nothing — it queries the status the financial Autonomous Agent has been continuously maintaining. The conversation is **fast** because the heavy work was already done in the background. The Assistant serves as the human interface to the status the Agent maintains.

When the same CFO adjusts the cashflow thresholds — *"from now on, escalate to me when the projected balance falls below X"* —, the Assistant communicates the adjustment to the Autonomous Agent, which incorporates it into its continuous logic. The human interaction is momentary; the effect operates persistently.

This composition is not optional for serious systems. An organization that operates only Assistants has a limited agentive system: humans must actively ask for each thing. An organization that operates only Autonomous Agents has an intransigent agentive system: humans cannot converse with the system, only receive alerts or query logs. Mature systems need both modes cooperating.

Recurrent anti-patterns

Three recurrent anti-patterns produce the failures the industry documents.

Anti-pattern A: selling an Assistant as an Autonomous Agent

A product that requires the human to invoke it every time **is an Assistant**, even if its marketing says "autonomous agent". The operational criterion is direct: if the system stops doing work when the human disconnects, it is not an Autonomous Agent. It is an Assistant. The consequence of this anti-pattern is that the client buys expecting autonomy and receives assistance with inflated vocabulary. The ensuing frustration is predictable: the client compares what was promised with what was received, discovers the gap, cancels.

Anti-pattern B: building an Autonomous Agent with Assistant architecture

A system that aims to be autonomous but operates by invoking cognition on every action. It works in pilot. It fails in production on cost and on speed. The economics behind this anti-pattern — why Botlets are the architectural answer that avoids collapse at scale — is developed in Chapter 5 §2 (Botlets). Here it suffices to retain the symptom: if the system stops working economically when volume moves from pilot to production, it is incurring this anti-pattern.

Anti-pattern C: governing an Autonomous Agent with Assistant policies

Assuming that basic permissions over data suffice when the agent operates autonomously. This is a grave error. The Autonomous Agent operates without immediate human supervision; it needs the robust governance described above — the five pillars of Trust Infrastructure —, not just access controls. The consequence: the agent acts outside reasonable bounds without anyone noticing until the incident. It is the typical cause of the risky behaviors that Chapter 2 documents as predominant in the field.

The difference between Assistant governance and Autonomous Agent governance is categorical. For the Assistant, the human who reads the response before acting closes the validation loop; it suffices that the system not allow obviously prohibited actions. For the Autonomous Agent, the human is not in the loop — validation, audit, impact limits, traceability, everything must be systemic (the Validation pillar that supplies that human loop is developed in Chapter 5 §4). Applying Assistant governance to an Autonomous Agent is building a system with no safety net under the trapeze.

The cooperative evolution

As the agentive system matures, the proportion of work executed by Autonomous Agents grows relative to that executed by Assistants. This progression is an observable property of the field, and it reflects the organization's transition across the Nadella Line.

In the **early stages**, typically ninety percent of agentive work operates in Assistant mode: the human stays at the helm, the Assistant helps with each task, the Autonomous Agent is marginal. In the **adoption stages**, the proportion shifts to seventy percent Assistant and thirty percent Autonomous Agent: the first functions — typically repetitive operational ones — move to autonomous operation. In the **maturity stages**, the proportion balances around fifty percent of each mode: the Autonomous Agent operates complete functions while the Assistant handles human queries. In the **advanced stages**, the proportion inverts — the Autonomous Agent executes seventy percent of the work and the human intervenes mainly to define rules, supervise, handle exceptions.

> *The Nadella Line separates the world where Assistants dominate from the world where Autonomous Agents dominate.*

This progression is what Chapter 2 described as the transition from the **online enterprise** to the **real-time enterprise**. An organization that lives with ninety percent Assistants is an online enterprise — humans assisted by AI that wait to be invoked. An organization that lives with seventy percent Autonomous Agents is a real-time enterprise — systems that operate autonomously with humans governing the whole.

How to identify the right mode?

For a given task, is an Assistant or an Autonomous Agent preferable? Four criteria help decide.

The first criterion: **must the human see every decision?**. If the answer is yes — because the decision requires human judgment, because regulatory responsibility demands supervision, because the cost of error is very high —, an Assistant is preferable. If the answer is no — because the decision is repetitive with clear criteria, because the volume is too high for human supervision, because speed demands it —, an Autonomous Agent is preferable.

The second criterion: **is the task triggered by the human or by the environment?**. If the human triggers it — the human formulates the question, the human requests the operation —, an Assistant is preferable. If the environment triggers it — an external event, a change in data, a monitoring alert —, an Autonomous Agent is preferable.

The third criterion: **is the task sporadic or continuous?**. If it is sporadic or variable — it happens a few times a day, at unpredictable moments —, an Assistant is preferable. If it is continuous or repetitive — it happens many times, with regularity —, an Autonomous Agent is preferable.

The fourth criterion: **does conversational latency matter?**. If the human waits for a response — the conversation has a turn-response dynamic —, an Assistant is preferable. If the operation runs in the background with no immediate latency pressure, an Autonomous Agent is preferable.

The rule of thumb that synthesizes the four criteria: if all four point to Assistant, use an Assistant. If all four point to Autonomous Agent, use an Autonomous Agent. If the mix is ambiguous, **design both modes cooperating**: an Assistant that queries the status maintained by an Autonomous Agent in the background. This composition is the one mature systems operate.

Conformance

An implementation that offers both modes conformant with this specification must satisfy:

Requirement	Level
Explicitly distinguish Assistant from Autonomous Agent in API and documentation	**MUST**
Assistant lives in Layer 2; does not require Layer 3	**MUST**
Autonomous Agent lives in Layer 3; persists state between sessions	**MUST**
Autonomous Agent exercises the five pillars of Trust Infrastructure	**MUST**
Composability: Assistant can query the Autonomous Agent's status	**SHOULD**
Distinction of billing model between the two modes	**SHOULD**
Prevention of the three anti-patterns	**MUST**

Facets

The primitives the five preceding sections described — AgencyDomain, Botlet, Capability, Trust Infrastructure, Assistant vs Autonomous Agent — live mainly in Layers 2, 3, and 4 of the Agentive Architecture. Layer 1 (Interaction) had been left without a primitive of its own: only described as **generation regimes** (Chapter 4 §1) and as **composition of presentation Botlets** (shell, view, operation). What was missing was a name for the atomic piece out of which those surfaces are built.

This section formalizes the **Facet** as the **sixth canonical primitive** — the minimal unit of Layer 1, an instrument the agent invokes during conversation or assembles into presentation Botlets.

> *The Botlet is the agent's muscle memory. The Facet is an instrument the agent picks up while it thinks.*

Definition

A **Facet** is an atomic, reusable component of Layer 1 (Interaction) that offers the user a specific form of non-conversational interaction: a freehand drawing board, a catalog-picker, a color matrix, a calendar, a clickable map, a slider, a drag-and-drop ordering, a file picker, a configurable canvas view. One of the many faces that interaction with the user can take at a given moment.

The Facet is an **instrument**, not a process. It lives and operates in Layer 1. It is invoked by cognition during active interactions or assembled by Layer 1 Botlets (shells and views) as a piece of their internal composition.

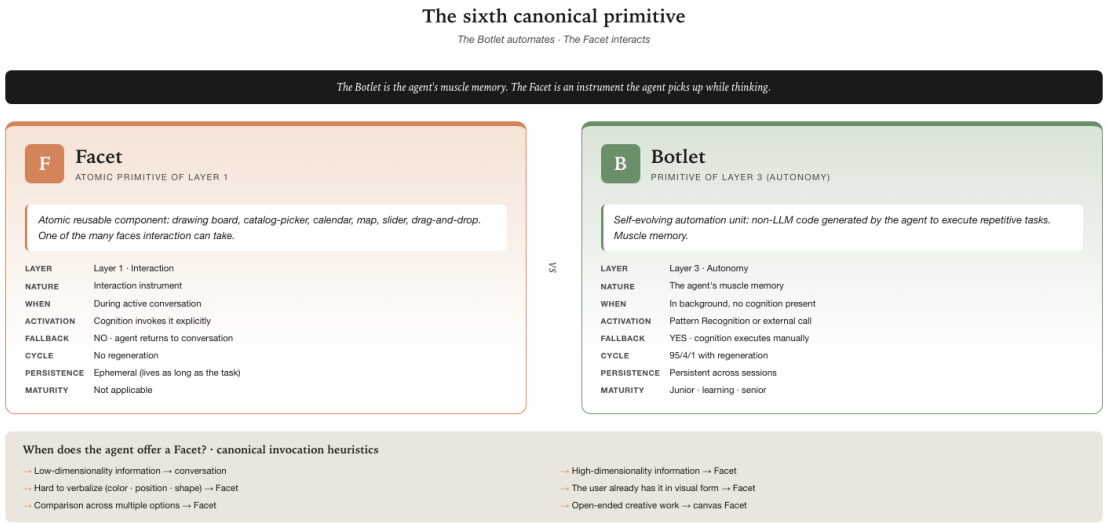


Figure 34: Facet vs Botlet · two primitives, two layers, two natures

Facet vs Botlet — the canonical distinction

The Facet and the Botlet are two of the agent's software primitives. They are easily confused because both are pieces with an identity of their own that the agent uses to do things. The canonical distinction:

Axis	**Facet**	**Botlet**
Layer	Layer 1 (Interaction)	Layer 3 (Autonomy)
Nature	Interaction instrument	Agent's muscle memory
When does it operate?	During active conversation	In the background, with no cognition present
Activation	Cognition invokes it explicitly	Pattern Recognition or external call
Fallback guarantee	NO — if it fails, the agent reverts to text	YES — cognition executes manually
Life cycle	Ephemeral (lives as long as the task)	Persistent across sessions
Regeneration	No regeneration cycle	`95/4/1` cycle with regeneration
Maturity phases	Not applicable	Junior · learning · senior
Reuse	Flat instrument catalog	Catalog by capability and domain

The ontological difference matters: **the Botlet automates; the Facet interacts**. A Botlet executes consolidated know-how without immediate human participation. A Facet opens a visual-manipulative channel of communication with a human who is active in the conversation.

Two canonical uses

Use 1 · Direct invocation by cognition

The agent, during a conversation, **decides** that the information it needs from the user can be obtained faster through a Facet than by continuing the verbal dialogue. It composes an ephemeral surface with one or several Facets, presents it to the user, receives the information, and the conversation continues. The ephemeral surface **is not a Botlet** and does not persist — it lives as long as the immediate task.

This use directly realizes the **GUI generated on-the-fly** regime of Chapter 4 §1. The Facet is the piece that materializes that regime.

Use 2 · Composition in presentation Botlets

The **shell** and **view** Botlets (Chapter 4 §1, *Composition of Layer 1*) assemble Facets as internal pieces of their construction. The "order detail" view uses the "product matrix" Facet + the "calendar" Facet + the "customer picker" Facet. The view Botlet defines the orchestration, the layout, the data flow between pieces; the Facets are the individual instruments the Botlet puts on the screen.

This use lets persistent surfaces (regime 3 of Chapter 4 §1) be built by reusing a Facet catalog, without each view Botlet having to reinvent every atomic component.

Declared bounded interaction

A piece of information already materialized — a self-contained snapshot the agent forged in its Engineering time — can carry interaction over its own data without ceasing to be a reproducible piece. But not all interaction is of the same kind. **Declared bounded interaction** is the category that separates what a piece can offer from what belongs to another primitive.

The distinction is drawn between two interactivities:

- **Free exploration** launches new and arbitrary queries against the source (ad-hoc drill or pivot), operates over an open space, loses reproducibility, exceeds `G1` (the Botlet generations, Chapter 5 §2), and lives outside the information proto-Botlet — in another Botlet or in cognition itself. A piece of information **MUST NOT** absorb it.
- **Declared bounded interaction** operates over the already-materialized snapshot, within a declared space (dimensions and values bounded in advance), preserves reproducibility, is `G1` — configuration, not code— and lives in the piece itself, realized via a Facet.

What distinguishes them?	Free exploration	Declared bounded interaction
New query against the source	yes, arbitrary (ad-hoc drill/pivot)	no — operates over the already-materialized snapshot
Interaction space	open	declared (bounded dimensions and values)
Reproducibility	lost	preserved (same config + data → same artifact + same set of controls)
Generation	exceeds `G1`	`G1` (it is configuration, not code)
Where does it live?	another Botlet / cognition (Layer 2 + Layer 1)	in the piece itself, via a Facet

Embedded Facet

The mechanism that realizes declared bounded interaction is already canonical: it is the composition of Facets (Use 2), applied inward into a materialized piece. It is the **piece** that composes the Facet — not the Facet that composes others —; the Facet remains atomic. An **embedded Facet** is a Facet bounded to a declared dimension of the piece's own data — a filter, a segmenter, an appearance selector. When it is activated, the piece's *data-bound* elements —KPIs and measures as declared aggregations (`sum`, `ratio`, and the like) over the embedded dataset, distributions, traffic lights— are **recomputed client-side** over the filtered subset: the aggregation is declared once and re-evaluated when the subset changes. Cognition does not explore; the Facet filters the snapshot.

The witness case is an attendance dashboard with a filter by area: the user selects one or several areas and the piece recomputes its KPIs and its traffic light over the subset.

This recognizes a third use of the Facet — an extension of Use 2 toward the materialized piece —: the Facet vs Botlet distinction admits this third use without altering the nature of the Facet, which remains ephemeral and without a fallback guarantee. What is new is that the piece can compose it for bounded interaction, in addition to invocation by cognition during conversation.

Associated agentive behavior

The existence of the Facet as a canonical primitive enables a specific agentive behavior: the agent, during a conversation, **estimates in real time** whether the information it needs is best obtained verbally or visually. When it estimates that the visual path wins, it offers an appropriate Facet.

The calculation is the agent's own cognitive act, not a pre-programmed feature of the product. The Facet as a primitive enables the decision; the heuristic exercises it.

Canonical heuristics for invocation

Nature of the information	Recommended modality
Low dimensionality + well structured (a yes/no, a date, a number)	Conversation
High dimensionality (multiple related fields)	Form or composition Facet
Hard to verbalize (color, position, shape, gesture)	Specialized Facet (color matrix, map, drawing)
The user already has it in spatial or visual form (a layout, a map, a drawing on paper)	Facet that receives that form directly
Comparison among multiple options	Catalog-picker Facet with comparative view
Configuration with many independent dimensions	Panel Facet with sliders and toggles
Open creative work (not an answer to a closed question)	Canvas or easel Facet

Anti-heuristics (when NOT to offer a Facet)

- When the question is genuinely closed and verbal — offering a Facet adds friction, it does not reduce it.
- When the user is on a channel with no graphical capability (pure voice, IVR, SMS) — the Facet is not invocable.
- When the cost of loading the Facet exceeds the benefit of the visual interaction (single-step interactions, trivial data).
- When the conversation is in flow and the Facet interrupts it inappropriately.

The agent that learns to calibrate these decisions — when to offer, when not to — operates in a full Layer 1. The one that only converses stays at the middle of the possible interactive range.

When a Facet is composed inside a materialized piece (declared bounded interaction), the decision ceases to be conversational and becomes part of the piece's configuration: the agent declares the space of controls at Engineering time, preserving the **reproducibility** —a MUST property of the information artifact— already established above.

Anatomy of the Facet

The canonical specification of a Facet includes six components:

1. **Identity** — canonical name (e.g.: `pizarra-dibujo`, `matriz-colores`, `calendario-rango`) plus version.
2. **Interaction modality** — what kind of input it accepts (touch, mouse, keyboard, gesture), what kind of output it produces.
3. **Input schema** — the data the invoker (cognition or view Botlet) passes to it when instantiating it.
4. **Output schema** — the data it returns when the user completes their interaction.
5. **Internal state** — what it keeps while active (intermediate selections, partial edits, undo stack).
6. **Channel compatibility** — which Layer 1 channels support it (web, mobile, kiosk, not supported on voice).

Facets are published in a **flat catalog**: there is no hierarchy of Facets because each one is atomic. What there is is a growing set of available instruments, indexed by modality and by application domain.

Emergent catalog

The industry converges gradually toward a canonical set of reusable Facets — the equivalent of the UI component catalog of the pre-agentive eras (Material, Bootstrap, Ant Design), but with the ontological difference that these components are **invocable by cognition** and do not serve solely to build humanly programmed applications.

Some emergent canonical Facets:

- `pizarra-dibujo` — freehand drawing surface.
- `catalogo-selector` — view of items with selection.
- `matriz-colores` — palette or individual color picker.
- `calendario-rango` — date or date-range picker.
- `mapa-clickeable` — map with selectable points.
- `slider-multi` — one or several related sliders.
- `dragdrop-orden` — reordering of items.
- `formulario-dinamico` — form with fields generated on the fly.
- `lienzo-creativo` — open canvas for artifact production.
- `selector-archivo` — invocation of the client's file system.

The spec does not close the catalog: new Facets are coined as the industry identifies canonical modalities that justify a primitive of their own.

Conformance

A Facet implementation conformant to this specification must satisfy:

Requirement	Level
Identity and version declared	**MUST**
Explicit input and output schema	**MUST**
Channel compatibility declared	**MUST**
Atomicity — does not internally compose other Facets	**MUST**
Explicit Facet vs Botlet distinction in documentation	**MUST**
Direct invocability by cognition during conversation	**MUST**
Composability within shell and view Botlets	**MUST**
Composed in a materialized piece as declared bounded interaction (declared space of controls, no new queries)	**MAY**
Preservation of the piece's reproducibility when composed as an embedded Facet (client-side recomputation, no Capability invocation)	**MUST**
Public catalog of Facets available to the AgencyDomain	**SHOULD**
Invocation heuristics documented for cognition	**SHOULD**

Evolution frontier

Three active areas of evolution of the Facet as a primitive deserve mention.

The **catalog standardization** is the first. The industry has not yet consolidated a universal canonical set of Facets. Each agentive platform defines its own, with partial intersections. The emergence of a common catalog with stable identities would let agents operate over any conformant AgencyDomain.

The **federation of Facets** is the second. When two AgencyDomains collaborate (federation, Chapter 5 §1), the Facets one exposes must be invocable from the other. The spec does not yet define a formal protocol for federated Facet invocation — it is open work.

The **channel negotiation** is the third. A Facet declares which channels it supports. Cognition must negotiate — if the user is on voice, it cannot offer the Facet; it must degrade to conversation. Without this explicit negotiation, surfaces fail on unexpected channels. The spec requires declaration but does not yet formalize the degradation protocol.

Agentlets

At the entrance of an emergency room there is a triage nurse. Her task is the same all day: receive the patient, assess them, classify them — red, yellow, green — and route them. The task is repetitive in its form; no repetition is identical to the previous one. Every patient arrives with different symptoms, different histories, signals that appear in no manual exactly as they present. The nurse cannot do her job on muscle memory — every classification demands fresh judgment —, but neither does she need the chief of internal medicine for every patient: her judgment operates within a bounded protocol, and when the case exceeds the protocol, she escalates. The triage nurse is neither a reflex nor the hospital's full deliberation. She is a third thing: **routine judgment, packaged into a role**.

Agentive systems are full of work with that shape. Classifying the emails that arrive at an operational inbox. Triaging monitoring alerts. Summarizing every new document that enters a case file. Extracting the entities from an invoice never seen before. Judging whether a message warrants immediate interruption or can wait for the morning briefing. These are tasks **recurrent in their form but fresh in every instance**: the pattern is stable, the execution demands interpretation. They do not mature toward determinism — not for lack of regenerations, but by nature.

The canon, up to this point, gave them two homes and neither was the right one. The first home is the **Botlet**: but the Botlet is non-LLM code by definition, and forcing an interpretive task into deterministic code produces exactly the fragility the architecture exists to avoid. The second home is the full **Cognition** of Layer 2: it works, but at the cost of the system's most expensive path — deliberative, conversational, sized for the genuinely new — applied to a pattern that repeats a hundred times a day. Between muscle memory and full deliberation, the middle rung was missing.

This section formalizes the **Agentlet** as the **eighth canonical primitive** — the Botlet's sibling unit whose execution body invokes bounded inference. The Agentlet is the home of routine judgment: packaged pattern on the outside, bounded judgment on the inside.

> *The Botlet is muscle memory. The Agentlet is packaged routine judgment. Cognition is reserved for the genuinely new.*

Definition

An **Agentlet** is a packaged unit of Layer 3 (Autonomy), sibling of the Botlet, whose execution body **invokes bounded inference**: a model sized to the task, applied within a charter the spec declares. Like the Botlet, it is a configured instance of a pre-forged piece — the **proto-Agentlet** —, it is hosted by the Botler, and it is engendered and maintained by the agent. Unlike the Botlet, each of its executions exercises judgment: it interprets the concrete instance in front of it rather than executing deterministic code.

The phrase that fixes the sibling doctrine: **an Agentlet is a Botlet with judgment inside — no more, no less**. It lives exactly like its sibling. It inherits by structure, not by exception, the entire apparatus of the Lets — the packaged units of Layer 3 —: the proto-/instance pattern, the catalogs and their network effects, the source-code vs spec separation, **manifestation** and **temporality** (`discrete/continuous`), the declarative quality contract, the derivation chain, the append-only log, and the operation API verbs (`specialize · invoke/schedule · read/subscribe · status/activate/deactivate/retire`). What changes is a single attribute — the body invokes inference — and from that attribute follow, in cascade, the differences the rest of this section develops.

Four properties define the conformant Agentlet. The first is that its **charter is declared in the spec**: what task it resolves, over what inputs, with what outputs, within what limits. The Agentlet does not choose what to do — its inference is spent on *how* to do its own work, never on deciding *what* its work is. The second is that its **inference is bounded**: the spec declares the model (sized to the task, not the frontier model of Cognition), the Capabilities it may consult, and the budget per execution. The third is that **all of its inference passes through the Botler**: the controlled handle the Botler hands it includes the cognition control point, so that every model call is metered, budgeted, validated and audited — bypass is structurally impossible, not merely forbidden. The fourth is that it carries a **fallback guarantee toward full Cognition**: when its bounded inference cannot resolve the case — the charter does not reach, the verdict's confidence falls below threshold, the input resembles nothing anticipated —, the Agentlet escalates and Layer 2 Cognition rescues. The process never stops.

The smuggling rule

The border between the Botlet and the Agentlet is a binary attribute, and the spec protects it in both directions:

- **If there is inference in the body, it is an Agentlet.** A Botlet with a model call hidden in its code is non-conformant — it smuggles cost and non-determinism into a unit whose guarantees (near-zero marginal cost, converging maturity, senior offline operation) depend on having neither.
- **If there is no inference in the body, it is a Botlet.** An Agentlet whose charter turned out to be resolvable with deterministic code is a misclassified Botlet — and a misbudgeted one: it pays tokens for work muscle memory would do for free. Cognition, upon detecting it, crystallizes it: it generates the Botlet that replaces it.

The second direction of the rule also describes a **natural trajectory**: there are tasks that enter the system as Agentlets — because at first every instance seemed to demand judgment — and that, with accumulated operation, reveal a crystallizable core. Cognition extracts that core into a Botlet and leaves the Agentlet only the genuinely interpretive residue, or retires it altogether. The mature system migrates work down the cost ladder: from Cognition to Agentlets, from Agentlets to Botlets.

Agentlet vs Botlet — the canonical distinction

Axis	**Botlet**	**Agentlet**
Execution body	Non-LLM code; zero inference	Invokes bounded inference, via the Botler's handle
Metaphor	Muscle memory	Packaged routine judgment
Natural task	Recurrent and **crystallizable** into code	Recurrent in form, **interpretive** in every instance
Determinism	Converges with maturity	Does not converge — correctness is statistical by nature
Marginal cost	~0 (traditional compute)	Tokens per execution, budgeted in the spec
Maturity	Junior → senior; senior = exogenous-only failures	Its own semantics: stabilized spec, decreasing escalation rate
Offline	Senior reliably operable offline	Only with an edge-resident model, declared in the spec
Fallback	Cognition executes manually	Escalates to full Cognition

What the table does **not** contain matters as much as what it does: proto-/instance, catalogs, spec, manifestation, temporality, quality contract, derivation chain, log, operation verbs, hosting by the Botler, genesis by the agent — all of that is identical, because it is predicated of the genus and not of the species. The spec calls the genus by its proper name: the **Lets** — the Botler hosts Lets; the Botlet and the Agentlet are its two species. The name derives from the family's own morphology and is, like Botler, normed vocabulary of the canon — not a ninth primitive. The canonical runtime relation is stated in its general form — **1 Process = 1 Botler + N Lets** — and each context instantiates it to the species at hand.

Agentlet vs Agent — agenda vs charter

The primitive's name claims membership in the **Agent** family, and the spec grants it: the umbrella term *Agent* covers, from this version of the canon onward, **three members** — the **Assistant** (Layer 2, reactive), the **Autonomous Agent** (Layer 3, proactive, an inhabitant) and the **Agentlet** (Layer 3, packaged). The concession is honest — the creature genuinely reasons, with a genuine model — but it demands drawing the border with the Autonomous Agent with the same rigor with which the canon drew Facet vs Botlet, because the two live in the same layer and both use inference.

The border fits in one line: **the Agent has an agenda; the Agentlet has a charter.** The Autonomous Agent pursues goals — it decides what to do, when, with what means, and engenders the units it needs. The Agentlet executes the task its spec declares — its judgment operates within the charter, never over the charter.

Axis	**Autonomous Agent**	**Agentlet**
Nature	**Inhabitant** of the AgencyDomain	**Catalog piece** — instance of a proto
Born through	Provisioning (six-phase lifecycle, first-order identity)	`specialize` over a proto-Agentlet
Agenda	Pursues goals; decides what to do and when	Fixed charter declared in the spec

Axis	**Autonomous Agent**	**Agentlet**
Cognition	Full: its own bindings, complete Capability tree, multi-LLM	Bounded: sized model, the Capabilities the spec declares
Engenders	Generates and regenerates Botlets and Agentlets	Engenders nothing; it is maintained
Fallback	*Is* the fallback — above it, only the human	Escalates to Cognition via the Botler
Governance	Exercises the five Trust Infrastructure pillars (MUST)	Governed through the control points of the Botler's handle

The **border test** protects the category from bleeding upward. Three questions; any answer on the side of autonomy means the piece is an Agent, not an Agentlet:

1. Does it choose its own goals, or does it receive them declared in the spec?
2. Can it alter its own process or engender other Lets?
3. Is its identity that of an inhabitant (provisioning) or of an instance (`specialize`)?

An Agentlet with a fat charter, an internal loop and its own criteria about what to pursue is not an advanced Agentlet: it is an Autonomous Agent in disguise, operating without the governance its nature demands — the agentive version of anti-pattern C in §5.

The Botler as guardian — same runtime, one more control point

The Agentlet brings no runtime of its own. It is hosted by **the same Botler** that hosts the Botlets, and this decision is one of principle, not convenience. The Botler is generic by definition: it manages lifecycle, isolation and execution of any unit without understanding its domain — and calling a model is not domain; it is a runtime service. The extension follows the pattern the Botler already practices: alongside `capability_call` and `log`, the controlled handle the Botler delivers on each invocation exposes a third control point — **`cognition_call`** —, the only path by which the Agentlet's body reaches a model.

The governance consequence is the entire argument: **because all of the Agentlet's inference passes through the handle, Trust Infrastructure sees it whole.** Every model call is metered (tokens, latency), budgeted (against the limit the spec declares), validated (the Pillar 3 mechanisms — hallucination detection, DLP, tokenization — apply at the control point) and audited (in the same append-only log, under the same identity, under the same governance as the rest of the AgencyDomain). A parallel runtime for Agentlets would have to duplicate all of that machinery — a second sandbox, a second handle scheme, a second escalation chain, a second presence in the distributed Layer 3 — only to end up delivering the same guarantees. The Botler's flat architecture already delivers them.

The legitimate concern that might push toward a separate runtime — the resource profile: model latency, per-token cost, eventual hardware acceleration — is a deployment matter, and the canon already holds the door open: Botler subtypes are distinguished **by topology and deployment role, never by domain**. A segregated execution pool for inference-heavy Lets is exactly a deployment-role distinction: one conceptual Botler, N processes if the operation calls for it.

It is worth making explicit the **two relations** that sustain the Lets, because they operate on different floors and both cover both species equally:

Relation	Who exercises it	Over Botlets	Over Agentlets
Hosting / execution — runtime without agency: lifecycle, isolation, controlled handle, log	The **Botler**	Yes	Yes (same handle, plus `cognition_call`)
Genesis / maintenance — deciding it exists, specializing it, regenerating it, answering for it	The **agent** (cognition)	Yes	Yes

The agent is the parent of both species; the Botler is the butler of both. No Let gets a separate house.

Agentlet maturity — stabilization, not convergence

The Botlet's junior → senior trajectory converges because each regeneration **crystallizes** variants into code: the senior is deterministic and that is why its only failures are exogenous. The Agentlet does not walk that trajectory — its correctness is statistical by nature and no amount of operation makes it deterministic. Pretending that an Agentlet "matures to senior" in the Botlet's sense is a category error with operational consequences: it can never be promised the offline reliability of the senior Botlet on the same basis.

The spec defines for the Agentlet a **maturity semantics of its own**, observable in the same append-only log that already tracks Botlets:

- **Spec stabilization** — the charter, the operating prompt and the configuration stop changing between revisions; `specialize` edits grow sparse.
- **Decreasing escalation rate** — the fraction of executions the Agentlet resolves within its charter without escalating to full Cognition grows and stabilizes. It is the functional analogue of `95/4/1`: the proportion between bounded judgment that suffices and full judgment that rescues.
- **Sustained quality under the declarative contract** — its declared quality attributes (freshness, SLA, degradation) hold within threshold over a sustained period.

The Agentlet's **offline** behavior demands explicit declaration, not inheritance from its sibling. A senior Botlet operates offline because it needs no model at all; an Agentlet operates offline only if its model resides on the edge. The spec of every conformant Agentlet **MUST** declare the locality of its bounded cognition — cloud-resident, edge-resident or hybrid, with the same vocabulary the spec already uses for Connectors — and its behavior without a network. Without that declaration, the "trivially explainable offline mode" consequence of the parallel topology breaks silently: the site that counted on its Autonomy Path discovers, with the network down, that half of its units needed a model on the other end of the wire.

proto-Agentlet — the pre-forged piece of judgment

Like its sibling, the Agentlet rarely springs from nothing: it springs from a **proto-Agentlet** — the pre-forged piece the agent configures in its Engineering time to instantiate an Agentlet specific to the case. The proto-Agentlet contains the body (the scaffolding of the interpretive task: the charter's structure, the

skeleton of the operating prompt, the input and output contracts, the escalation thresholds); the Agentlet is the configured instance. The proto-Botlet's two classes apply unchanged: a **tempered** proto-Agentlet resolves one interpretive function and is configured through bounded parametrization (an operational email classifier, an invoice field extractor); a **platform** proto-Agentlet is a generic judgment engine whose specialization lives in compositional configuration and covers N functions of its domain.

The **derivation chain** extends without friction: documented use cases require Lets — zero, one or several, of either species —, and each Let is an instance of some proto in the catalog. Every conformant Agentlet **MUST** be traceable in that chain, and the append-only log **MUST** record the proto-Agentlet of origin of each instance. The common catalogs — public and open at AgencyDomains.org, proprietary codices, private contracts, sovereign agreements — accumulate proto-Agentlets with the same network effects with which they accumulate proto-Botlets: implementer n+1 receives charters, operating prompts and thresholds refined by implementers 1 through n.

The **generations** (`G1`/`G2`/`G3`) apply with a natural reading: in `G1` the agent configures the proto-Agentlet — fills in the charter, adjusts the operating prompt within the scaffolding, sets thresholds — without writing its body. The `G1`/`G2` edge is the same as the Botlet's: configuration that a well-defined Capability evaluates is `G1`; extension of the proto's internal logic is `G2`.

The economics of the three rungs

With the Agentlet, the agentive system's economic ladder is complete. Chapter 4 presented two paths — costly Cognition, cheap Autonomy —; the fine-grained view distinguishes **three rungs of per-execution cost**:

Rung	Unit	Inference	Marginal cost
1	**Botlet**	Zero	~0 — traditional compute
2	**Agentlet**	Bounded: sized model, declared budget	Low and **budgetable per Let**
3	**Cognition**	Full: deliberative, frontier model, complete tree	High — reserved for the new

The middle rung is the one that was missing. Without it, every recurrent interpretive task paid rung-3 prices — or was forced, fragile, onto rung 1. With it, the organization assigns each pattern to its natural cost, and the mature system migrates work down the ladder: Cognition cedes interpretive routines to Agentlets; Agentlets cede crystallizable cores to Botlets.

The Agentlet's arrival **qualifies, but does not break**, the economic promise of the Autonomy Path. The path remains the cheap one — but it stops being uniformly free: it contains Lets of ~0 marginal cost and Lets of bounded cost, and the mix is declared per Let, not averaged away. Under **fixed Subscription** plans, the argument of Chapter 5 §2 refines in the same direction: Botlets remain the mechanism that makes sustained autonomy possible without exhausting the quota, and Agentlets consume quota — bounded, budgeted, visible in the log — so their share of the mix is an explicit economic decision of the organization, exactly as the decision of which patterns to consolidate into Botlets is.

The ladder orders three things, not only cost. It also orders **latency** — a Botlet answers in traditional compute time; an Agentlet, in that of its sized model; Cognition, in that of deliberation — and

infrastructure dependence: the senior Botlet depends on no inference provider, while the junior one still falls back on Cognition; the Agentlet depends on the model its spec declares, which may be edge-resident only where a sized model of sufficient quality exists (an open frontier, see «Evolution frontier»); Cognition depends on frontier infrastructure. Every pattern that migrates down the ladder answers faster and depends less on a third party, without removing the dependence: the new and every fallback still climb to Cognition.

Typed decision — the Agentlet's natural body. In 2026 the first public model of a class that does not generate text appeared: it receives the state of the instance and a set of typed-answer questions — choose among options, score within a range, yes or no — and returns decisions with calibrated probability. It is the natural form of an Agentlet's charter: the spec declares the questions and their types, the output cannot fall outside the type, and the calibrated probability sets the escalation threshold — above it, the Agentlet resolves; below it, it escalates to Cognition, which is the species' fallback guarantee. Three cautions the spec makes its own: the Agentlet's calibration remains an open problem (see «Evolution frontier»), and the calibration a model's provider declares is measured before a threshold rests on it (declarative quality contract, Pillar 3); and a model offered only as hosted by its provider is not edge-resident, so the Agentlet that uses it does not operate offline and inherits that provider's dependence; and a typed decision carries no explanation, so whatever must be justifiable escalates to Cognition by design.

There is a second cross-cutting consequence: **the Validation pillar gains a seat in Layer 3.** Until this version of the canon, Validation was exercised in Layer 2 (partial) and Layer 4 (principal), because Layer 3 contained only deterministic code. With Agentlets, statistical correction enters the muscle-memory zone — exactly the volatility the Bounded Concerns Architecture (Chapter 3) teaches to confine —, and the Pillar 3 mechanisms (hallucination detection, structured-output validation, DLP, tokenization) apply at the Botler's `cognition_call` control point, over every execution of every Agentlet. The thinness of the domain is preserved: statistical judgment stays confined to declared, governed, auditable Lets — never scattered through the runtime.

When to use an Agentlet — and when not?

The decision among the three homes — Botlet, Agentlet, Cognition — resolves with two questions in cascade:

First: is the task recurrent? If not — if it is one-off, exploratory, or its pattern has not yet stabilized —, it belongs to Cognition. The practical ten-invocations rule of §2 applies to both unit species alike.

Second: does each instance demand judgment? If not — if the logic is crystallizable into deterministic code —, **Botlet**, always: it is the cheapest rung and the only one that matures toward reliable offline operation. If yes — if the pattern is stable but each execution interprets a new instance —, **Agentlet**.

Three signals confirm that a task is Agentlet territory: the input is **natural language or unstructured content** (emails, documents, transcripts, free-form descriptions); the output requires **classifying, summarizing, extracting or judging** rather than computing or transporting; and attempts to solve it with rules produce **exception lists that grow without converging**. And two anti-signals return the task to its neighbors: if the "judgment" turned out to be a stable decision table, it is a Botlet not yet crystallized; if the charter cannot be declared — the task requires deciding what to do, not merely how —, it is Agent work, with Agent governance.

Conformance

An Agentlet implementation conformant with this specification must satisfy:

Requirement	Level
Charter declared in the spec: task, inputs, outputs, limits	**MUST**
Bounded inference declared: model, accessible Capabilities, per-execution budget	**MUST**
All inference routed through the `cognition_call` control point of the Botler's handle	**MUST**
Fallback guarantee: escalation to full Cognition on out-of-charter cases or sub-threshold confidence	**MUST**
Zero inference outside the handle (the smuggling rule, Botlet→Agentlet direction)	**MUST**
Hosting by the generic Botler — no parallel runtime per species	**MUST**
Traceability: every execution and every inference call in the append-only log	**MUST**
Traceable in the use-cases → Lets → protos chain; proto-Agentlet of origin recorded	**MUST**
Declared locality of the bounded cognition (cloud / edge / hybrid) and offline behavior	**MUST**
Pillar 3 Validation applied at the cognition control point	**MUST**
Explicit Agentlet vs Autonomous Agent distinction in API and documentation (the border test)	**MUST**
Maturity metrics of its own: spec stabilization and escalation rate	**SHOULD**
Crystallization: detection of deterministic cores and extraction into Botlets	**SHOULD**
Execution pool segregated by resource profile (deployment role)	**MAY**

Evolution frontier

Three active areas of evolution of the Agentlet deserve mention in closing.

Edge models for bounded judgment are the first. The viability of the offline Agentlet depends on small models running at the physical site with sufficient quality for bounded charters. The state of the art advances quickly; the spec already provides the declaration vocabulary (locality of the bounded cognition) so implementations can adopt edge models without a contract change.

Confidence calibration is the second. The Agentlet's escalation to full Cognition depends on the Agentlet itself estimating when its verdict does not reach the threshold — an open problem of statistical calibration. While it matures, conservative implementations compensate with low thresholds: over-escalating costs tokens; under-escalating costs trust.

Assisted crystallization is the third. The automatic detection of deterministic cores inside Agentlet charters — the Agentlet → Botlet trajectory of the smuggling rule — is today a judgment of cognition in its Preparation time. Its formalization as an inverse Pattern Recognition mechanism (detecting what *stopped* needing judgment) is open work.

With the Agentlet, the roster of formal constructions of this chapter closes — **eight canonical primitives**: AgencyDomain, Botlet, proto-Botlet, Agentlet, Capability, Trust Infrastructure, the Assistant vs Autonomous Agent distinction, and the Facet. Whoever has followed Chapters 4 and 5 holds the complete constructive vocabulary with which the agentive category can be reasoned about and built — from the computational space that contains it all to the three-rung ladder by which work finds its natural cost.

Chapter 6 shifts the gaze from the individual system to the market. It allows whoever builds or invests to answer with discipline the question of where each actor — one's own or someone else's — competes, and why the same link in the chain can be a hotly contested zone or still-open territory.

Chapter 6 · Market

An architecture does not live in a vacuum: it competes in a market. This chapter situates any actor by means of the **AI value chain** —eleven links × four depths—, drills into select links with *deep-dives*, and extends the model to the **Carbon World**. §1 develops the general model; the sections that follow apply it.

The AI value chain

> **Note on the datability of the products mentioned.** The listings of specific products in this chapter describe the state of the agentive AI market as of **May 2026**. The conceptual structure of the value chain's links is stable; the actors listed are illustrative of the moment. Later readings should take the names as a snapshot, not as permanent coverage.

The Artificial Intelligence industry presents itself, as of May 2026, as a dense ecosystem where dozens of products, platforms, and frameworks compete and coexist. Without a clear classification model, it becomes difficult to answer the fundamental questions any serious executive, architect, or strategist asks when facing the field: where does each actor play? which links does it dominate? where is there concentration and where is there room? in what territory does my proposal compete, and against whom?

The fragmentation of the field is no accident. It is the result of explosive growth where each new entrant builds its own category, chooses its own vocabulary, defines its own positioning. The consumer — be it an enterprise buyer, a market analyst, an investor — ends up overwhelmed by a language that each actor molds to its own convenience. *Agent platform, AI gateway, LLM framework, agentive infrastructure, autonomous agent, vertical assistant, model marketplace* — all terms that circulate without precise definition, all terms that different vendors use with different meanings.

This chapter proposes a map that disciplines that conversation. The map does not resolve every ambiguity in the field — the industry is too young for a single map to capture all its complexity —, but it delivers a shareable frame: a precise language that allows one to situate any actor in its position, compare actors with one another, and reason about a particular product's strategy relative to the broader field. The map that follows is the **AI value chain**, in its two-dimensional version. It is an original contribution of this book, derived from the author's prior work conceptualizing the field, and it is offered as an open tool for the industry, not as proprietary intellectual property.

The two dimensions

The model organizes the AI technology value chain along **two dimensions**. The two dimensions operate orthogonally — an actor situates itself on each one independently —, and the combination of the two produces the positioning space where the actor lives. The two dimensions are **coverage** and **depth**.

Coverage is the horizontal dimension: which links of the value chain the actor touches. An actor may touch a single one, several, or many. Coverage is a metric of **reach** — how much of the field's territory the actor operates. An actor with broad coverage touches many links; an actor with focal coverage touches one or a few.

Depth is the vertical dimension: with what level of control the actor operates within each link. An actor may consume a link superficially — using third-party APIs — or build the link deeply — manufacturing the base technology. Depth is a metric of **control** — how much of the link the actor dominates. An actor with shallow depth depends on underlying providers; an actor with deep depth builds the substrate on which others operate.

Each actor can position itself in one or more links, at different depths in each one. A single actor may operate at Core depth in its native link and at Platform depth in adjacent links — a common pattern in the contemporary market.

The result is a **map that allows one** to classify any AI product by the links it spans, compare actors by coverage and depth in the chain, identify zones of concentration and zones of opportunity, and strategically position one's own products against the market.

The eleven links

Figure 35: The eleven links of the AI value chain

The AI value chain decomposes into eleven sequential links, each with a clear and separable function. The separation is not arbitrary: each link corresponds to a distinct functional capability that an actor can operate independently, with its own economics and competitive dynamics.

We lay out each link with its functional description. The sequence is not linear in the sense that a data process passes through all the links in order, but it does reflect a conceptual progression from the field's raw material (data) to where the agent touches the real world (the environment).

Link 1 · Data (Data Layer). Acquisition, annotation, management of training datasets. It is the raw material that feeds the foundation models. The actors in this link produce curated datasets, annotation tools, large-scale data-processing pipelines. Without this link, the models do not exist.

Link 2 · Model (Foundation Model). Base AI models: LLMs and multimodal models that provide fundamental capabilities of language, reasoning, and generation. It is where the great labs — OpenAI, Anthropic, Google, Meta, DeepSeek — concentrate capacity. The actors here build the models that the rest of the field consumes.

Link 3 · Access (Access Layer). APIs and model access layers. Quota control, authentication, and monetization of consumption. It is where inference is sold as a service: OpenAI's, Anthropic's APIs, AWS Bedrock, Google's Vertex AI. It is also where products such as model gateways (Portkey) operate, offering abstraction over multiple models.

Link 4 · Agents (AI Agents). Conversational interfaces and assistants. From reactive agents — chat — to autonomous agents capable of executing complex tasks. It is where the most visible products appear: ChatGPT, Claude, GPT-4 with plugins, orchestrated agent systems.

Link 5 · Specializations (Domain Experts). Autonomous agents specialized by vertical domain: coding, legal, marketing, support, productivity, professional-work memory. It is where the vertical specialists appear: Cursor for coding, Harvey for legal, Jasper for marketing, Fin for customer support, umeeta for the memory of consulting engagements. The difference from link 4 is one of know-how depth in a specific domain.

Link 6 · Runtime (Agent Runtime). The operational environment where agents live and operate autonomously. Lifecycle, state persistence, identity, scheduling, and multi-agent orchestration. It is where Layer 3 of the Agentive Architecture materializes as a product. An emergent link — most traditional actors still do not cover it explicitly.

Link 7 · Firewall (Security Layer). Security, control, and governance. Protection against prompt injection, hallucinations, content filtering, and usage auditing. Products such as Lakera, Lasso Security operate here. It is a critical link for enterprise production — without a firewall, the agentive system cannot operate in regulated industries.

Link 8 · Observability (Observability). Monitoring, traceability, costs, and quality of AI systems in production. The operational feedback loop. Products such as Langfuse, LangSmith, Helicone, Arize operate here. It is a mature link — AI observability has several Core-depth products actively competing.

Link 9 · Tools (Tools). Specific capabilities that agents can invoke. Includes meta-tools: protocols (MCP), vector databases (Pinecone, Weaviate), RAG frameworks. It is where the agent extends its capability to touch specific systems.

Link 10 · Integrations (Integration Layer). The bridge between the AI world and the Environment. Orchestration, transformation, and mapping of integration logic between systems. Products such as Zapier, Make, n8n operate in this link in their traditional form; the agentive equivalent is still an emergent category.

Link 11 · Environment (Environment). What is external to the chain: enterprise systems (ERPs, CRMs, databases), the physical world (IoT, industrial processes), and biological systems. It is the least

developed link, and we develop its implications in detail in the Carbon World section.

The links are not arbitrary. Each one corresponds to an **operational design decision** in any productive AI system. Skipping a link is not elegance: it is architectural debt that is paid in production.

The four depths

The links define **where** an actor participates in the chain. But within a single link, actors operate at different levels of depth. An actor that consumes a model API and another that trains the foundation model **both participate in the Model link**, but their differentiation, dependency, and competitive moat are radically different.

The model defines four levels of depth, from lesser to greater control over the link's capability. The four depths apply to any link — an actor may be Wrapper in Data, Platform in Model, Core in Access. The uniformity enables cross-comparison between distinct links.

Wrapper (level 1). The actor consumes capabilities via third-party APIs or SDKs. It adds user experience or business logic without building the underlying capability. Characteristics: low differentiation relative to other wrappers that use the same underlying providers, high dependency on the provider, low switching cost. An app that calls the OpenAI API to answer questions is a Wrapper in the Model link.

Platform (level 2). The actor operates and manages its own capability over third-party Core components. It adds orchestration, SLAs, and operational control. Moderate differentiation: the customer pays for the operational capabilities the Platform adds, not for the underlying capability that remains third-party. Azure OpenAI is a Platform in Model: it operates OpenAI's models with SLAs and enterprise governance, but the models belong to the original provider.

Core (level 3). The actor builds the link's foundational capability with its own technology: differentiated models, engines, or algorithms. High competitive moat based on intellectual property. OpenAI is Core in Model: it builds its own models. Anthropic, Google with Gemini, Meta with Llama — all are Core in Model. The distinction between Core and the higher levels is where most of the value captured in the AI field resides.

Infrastructure (level 4). The actor provides the computational, storage, or connectivity substrate on which the higher levels operate. A very high moat based on scale and capital. NVIDIA is Infrastructure in Model: the GPUs NVIDIA manufactures are the substrate on which the models operate. AWS, GCP, Azure are Infrastructure in many links — they provide the compute and storage underlying almost the entire industry.

The progression Wrapper → Platform → Core → Infrastructure is one of **increasing control** over the link. Wrapper consumes; Platform operates; Core builds; Infrastructure sustains. Each level of depth typically implies greater investment, greater technical specialization, a greater competitive moat. It also implies greater risk: a Core that bet on a technology the market discarded is left with an asset hard to reposition; a Wrapper that bets wrong switches providers in hours.

Coverage × Depth — the positioning space

The combination of coverage (links) and depth (levels) produces a **two-dimensional space** where any actor is positioned. The horizontal axis shows **how many links** an actor spans; the vertical axis shows **at what depth** it participates in each one.

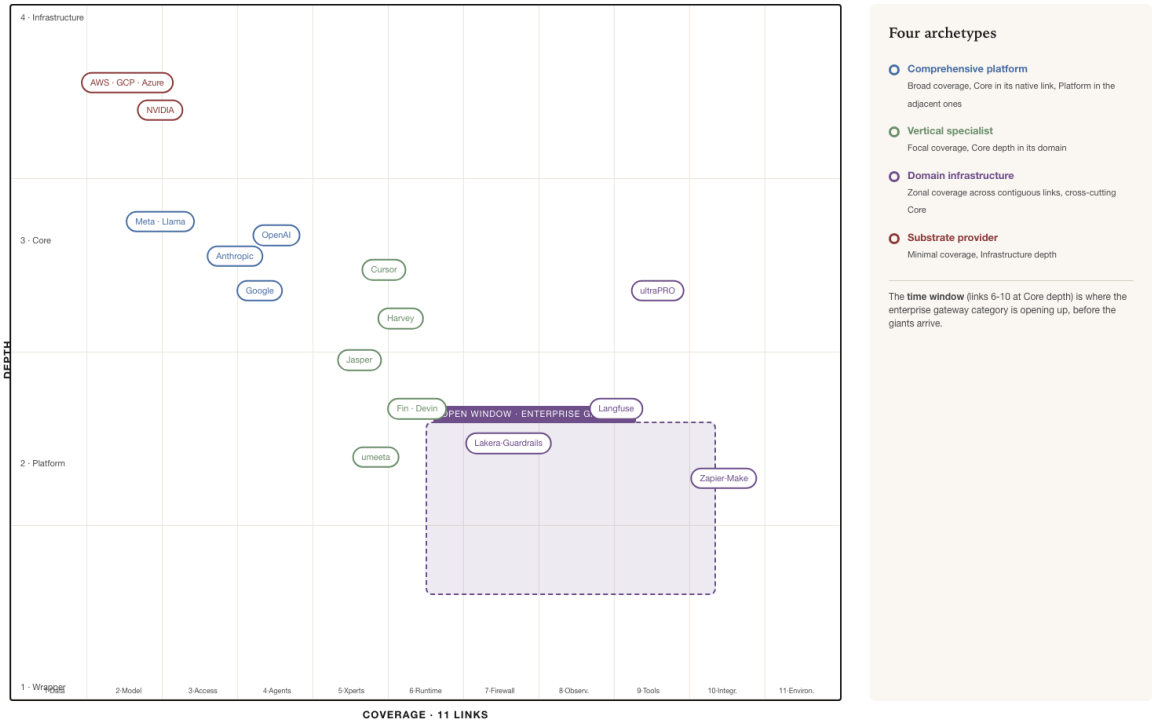


Figure 36: Two-dimensional space · coverage × depth

A single actor may operate at different depths in different links. OpenAI is Core in Model but Platform in Access (its APIs) and Platform in Agents (ChatGPT). This per-link heterogeneity is the rule, not the exception. Few actors have uniform depth across all the links they touch — and when they do, they are typically very focused actors such as NVIDIA in computational Infrastructure.

The diversity of positions in the two-dimensional space allows one to identify positioning archetypes that recur in the market, with distinct strategic properties. The next section develops the four canonical archetypes.

Emerging strategic archetypes

From this two-dimensional space, **four recurring archetypes** emerge. Each archetype describes a positioning pattern with characteristic strategic properties. The four archetypes are: Comprehensive platform, Vertical specialist, Domain infrastructure, Substrate provider.

Comprehensive platform

The **Comprehensive platform** archetype combines broad coverage (three or more links) with **Core** depth in its native link and **Platform** depth in adjacent links. It is the archetype of the great AI labs that dominate the field in 2026.

OpenAI exemplifies the archetype: Core in Model (builds GPT), Platform in Access (sells the API), Platform in Agents (operates ChatGPT and Operator), Platform in emerging Specializations (the verticalized GPTs). Anthropic follows a similar pattern but with a different emphasis: Core in Model (builds Claude),

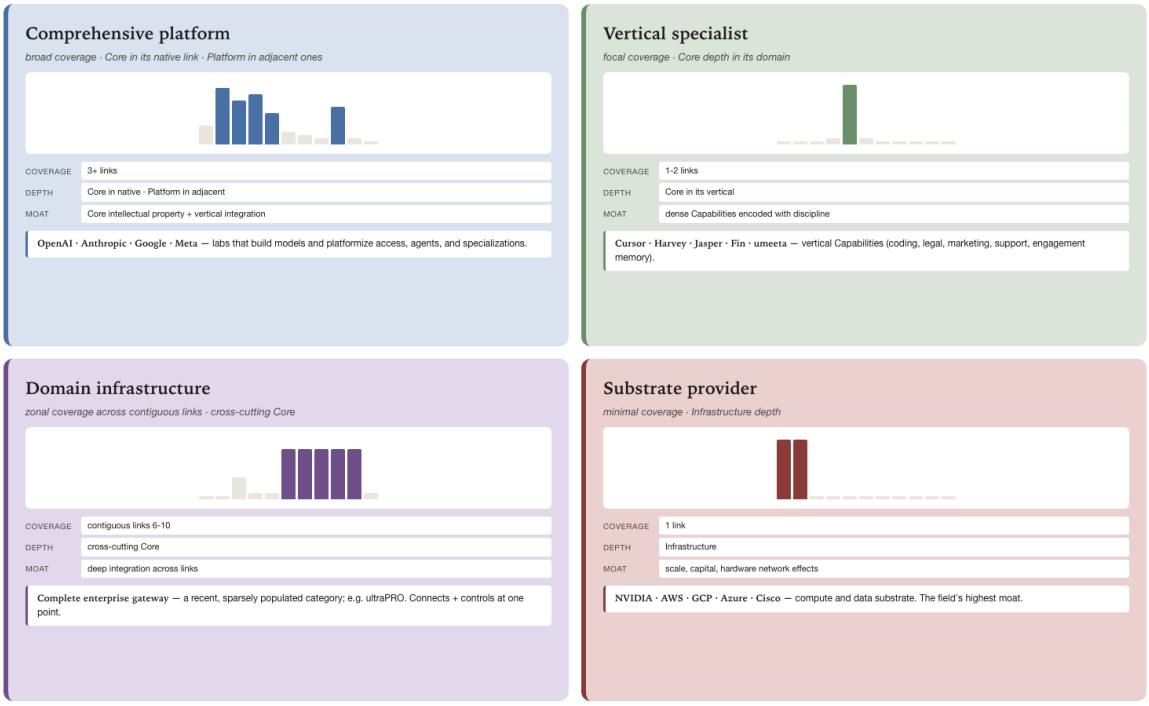


Figure 37: The four strategic archetypes

Core in Access via MCP (its open contribution to Tools), Platform in Agents. Google with Gemini does the analogous. Meta with Llama is a particular case: Core in Model distributing open source, with no proprietary Access or Agents platform — its moat is model distribution, not operation.

The competitive moat of the Comprehensive platform is the intellectual property of the Core combined with vertical integration into adjacent links. A Wrapper that calls OpenAI cannot easily replicate what OpenAI does in its entirety — to do so, it would have to train its own model (Model link at Core depth), build its own Access infrastructure, operate its own Agents platform. That complete chain requires capital and talent that few actors have.

Vertical specialist

The **Vertical specialist** archetype combines focal coverage (one or two links) with **Core** depth. It is the archetype of the actors that have concentrated on specific domains and build depth there.

Cursor exemplifies the archetype in coding: Core in Specializations for programming. Harvey AI does the same in legal. Jasper in marketing. Fin (from Intercom) in customer support. Devin aims for Core in Specializations of autonomous coding. umeeta operates the same archetype in professional consulting, with Core in the engagement-memory layer. Each one has narrow coverage — one or a few links — but Core depth in its specific vertical.

The competitive moat of the Vertical specialist is the depth of vertical know-how, which typically materializes as dense Capabilities — the codified professional knowledge we discussed in Chapter 5. A generic GPT can answer legal questions, but Harvey AI answers them at much higher quality because

it has Legal Capabilities built with discipline. The difference is not marketing — it is structural. A competitor that wanted to replicate Harvey would have to build the Legal Capability tree with the same rigor, which takes years.

Domain infrastructure

The **Domain infrastructure** archetype is the most recent in the industry and the least populated. It combines zonal coverage (two or more contiguous links) with **Core** depth across a functional domain, with possible extensions to non-contiguous links at lesser depth.

An actor that is Core in Runtime, Firewall, Observability, Tools, and Integrations — links 6, 7, 8, 9, 10 — with a Platform extension in Access constitutes the paradigmatic case of the archetype. The combination of zonal coverage across five contiguous links with Core depth constitutes an **enterprise gateway**: the foundational layer for connecting and controlling AI in production.

The competitive moat of Domain infrastructure is the deep integration between links that other actors treat separately. Building Core in Runtime is a merit; building Core simultaneously in Runtime, Firewall, Observability, Tools, and Integrations, **coherently integrated**, is architectural property that few actors have. The structural reason is that these five links operate together in production — without one, the others lose value — and building only one leaves the actor dependent on complements that typically do not exist as an integrated product.

Substrate provider

The **Substrate provider** archetype combines minimal coverage (one link) with **Infrastructure** depth. It is the archetype of the actors that sustain the industry from the deepest layer.

NVIDIA exemplifies the archetype in Model: the GPUs NVIDIA manufactures are the computational substrate on which the models operate. AWS, GCP, and Azure are Substrate providers across multiple links — Data, Model, Compute in general. Cisco is one in networking for distributed AI.

The competitive moat of the Substrate provider is scale, capital intensity, and network effects in hardware or the data center. Building a company that competes with NVIDIA in GPUs requires investments of trillions of dollars and accumulated generations of R&D. Building a company that competes with AWS in compute at scale requires global physical infrastructure. These moats are the highest in the field, but they are also the ones that require the greatest initial capital and have the longest return cycles.

Mapping the principal actors

By way of example, the following table classifies representative product families from the current market by the links they span and the depth in each one. The figures are **depth levels** (1-4); the parentheses indicate a framework or meta-tool (to build with, not to use).

Actor	Da	Mo	Ac	Ag	Xp	Ru	Fi	Ob	He	In
Data and annotation										
Scale AI / Labelbox	3									
Hugging Face	3	2								
Comprehensive platforms										
OpenAI (ChatGPT, GPT API)		3	2	2	2	3			3	
Anthropic (Claude, MCP)		3	3	2					3	

Actor	Da	Mo	Ac	Ag	Xp	Ru	Fi	Ob	He	In
Google (Gemini)		3	2	2					3	
Meta (Llama)		3								
Perplexity			2	3	3					
DeepSeek / Qwen / Ernie		3	3	3						
Vertical specializations										
GitHub Copilot				2	3					
Cursor / Replit					3					
Devin					3	3			3	
Harvey / Jasper / Fin					3					
umeeta (engagement memory)					3					
Agentive substrates										
Agentia (private regime)				3		3				
Soveria (public regime)				3		3				
Frameworks and tools										
LangChain / Graph				(3)		(3)			(3)	
AutoGPT / CrewAI				(3)		(3)				
Pinecone / Weaviate									(3)	
Operations and governance										
Guardrails / NeMo / Lakera							3			
Langfuse / LangSmith / W&B								3		
Zapier / Make / n8n										3
ultraPRO (enterprise gateway)			2			3	3	3	3	3
Computational infrastructure										
NVIDIA		4								
AWS / GCP / Azure	4	4								

Legend of links: **Da** Data · **Mo** Model · **Ac** Access · **Ag** Agents · **Xp** Specializations · **Ru** Runtime · **Fi** Firewall · **Ob** Observability · **He** Tools · **In** Integrations. Depth levels: **1** Wrapper · **2** Platform · **3** Core · **4** Infrastructure. The parentheses — e.g. (3) — indicate a framework or meta-tool (to build with, not to use).

> **Note.** The table covers links 1 through 10. Link 11 (Environment) is omitted as being external to the chain — it is the territory on which the preceding links act, not a link that an AI actor occupies at some depth. Its implications are developed in the Carbon World section.

The table, read as a whole, lets one see patterns that the individual inspection of each product does not reveal. The comprehensive platforms tend to concentrate in links 2-4. The vertical specialists accumulate in link 5. The operations and governance products distribute across links 7-10. Computational infrastructure occupies principally link 2 at depth 4.

Strategic readings of the map

The map is not merely descriptive — it is a tool for strategic analysis. Three readings of the 2026 map make it possible to understand the state of the field and where the opportunities lie.

Concentration by archetype

The **vertical specialists** dominate link 5 (Specializations). Cursor, Harvey, Jasper, Fin, Devin: each built dense vertical Capabilities and captures market in its domain. The concentration is healthy — multiple actors with little overlap, each owner of its vertical. It is where innovation is most vibrant in 2026.

The **comprehensive platforms** concentrate links 2-4 (Model, Access, Agents). Five dominant global actors — OpenAI, Anthropic, Google, Meta, DeepSeek/Qwen/Ernie — and derived positions. The concentration is high and grows over time, because building Core in Model requires capital and talent that few actors can sustain. It is the link with the highest barrier to entry.

The **infrastructure** is concentrated in NVIDIA for compute and the hyperscalers for data and compute at scale. An extreme capital moat. The concentration here is structural and probably persistent — it is reasonable to expect that no significant entrant will appear in these links at Infrastructure depth within the foreseeable horizon.

The less-contested spaces

There are zones of the map where Core depth is open and where an actor with discipline can build a competitive position without facing massive incumbents.

Link 1 (Data) at Core depth: few Core actors (Scale AI, Labelbox); the rest are commodity. There is room for actors that build their own capability in specialized data.

Links 6-10 simultaneously — Runtime, Firewall, Observability, Tools, Integrations — at Core depth with zonal coverage: the territory of Domain infrastructure. Actors that combine these five links at Core depth are rare. It is the space where the category of the **complete enterprise gateway** is opening.

Link 11 (Environment) with connection to IoT and the physical world: practically empty in terms of actors specifically designed for the Agentive World. It is the frontier of the next generation, and the Carbon World section develops the implications.

The trajectory of the giants

The giants — OpenAI, Anthropic, Google, Microsoft — advance **link by link**: from Model to Access, from Access to Agents, from Agents to Specializations. The progression is historically observable. OpenAI was born as a Model lab, expanded to Access (API), expanded to Agents (ChatGPT), is expanding to Specializations (GPTs).

But reaching the Integrations link — where the agent touches the real systems of the enterprise — demands an integrative effort, company by company, that does not scale with these actors' logic. OpenAI can offer ChatGPT Enterprise with connectors to Slack and Salesforce, but integrating deeply with each customer's ERP, with its particular CRM, with its legacy data warehouse — that is not platform work, it is integration work. **This creates a temporal window** for actors specialized in links 6-10 (domain infrastructure) to build a position before the giants arrive. The window is not indefinite — the giants eventually reach integrations, possibly via acquisition — but it exists now and offers strategic opportunity to whoever understands it.

Figure 38: The trajectory of the giants · and the open window

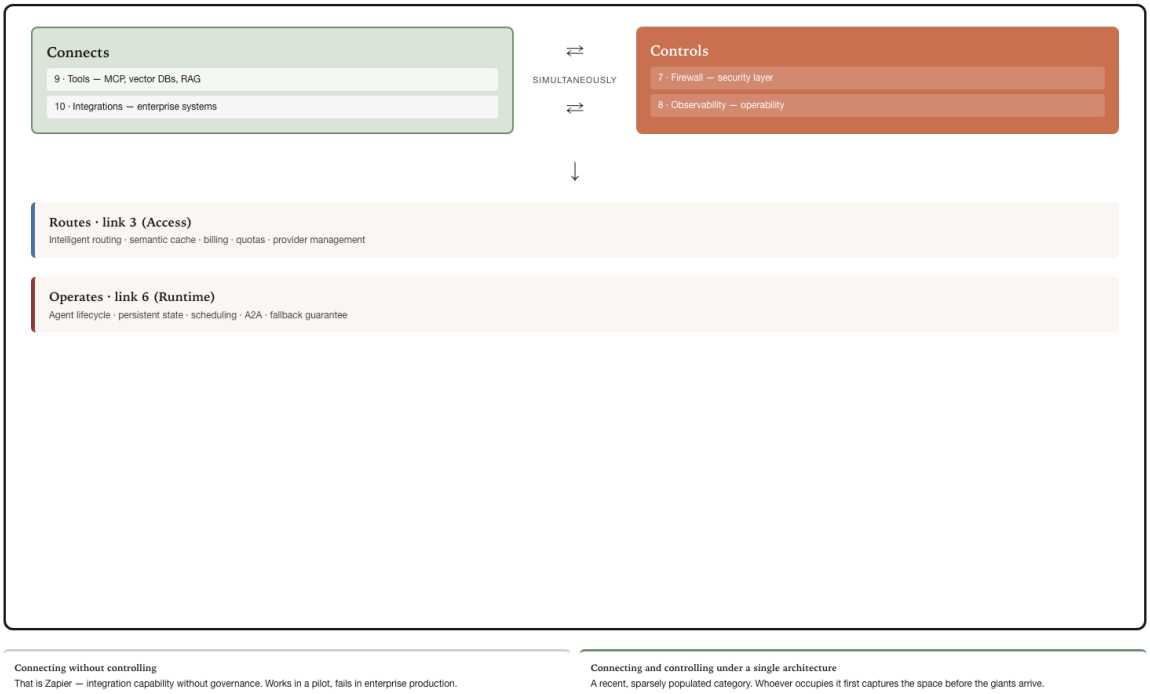


Figure 39: The enterprise AI gateway · connect and control

The enterprise gateway as a category

The combination **Core in Runtime + Firewall + Observability + Tools + Integrations**, with a **Platform extension in Access**, defines an architectural category with a unique function: to **connect and control simultaneously** the operation of enterprise agents. It is the formal materialization of Layer 4 of the Agentive Architecture over the market links.

This category is called the **enterprise AI gateway** (figure above).

To **connect** without **controlling** is Zapier — integration capability without governance. To **control** without **connecting** is Lakera — security capability without integration. The combination of both in a single architectural point is a recent and still sparsely populated category. The actors that occupy it first capture the space before the giants arrive.

Competitive analysis of the enterprise gateway

When evaluating actors that aim to occupy the enterprise gateway, a useful rubric compares nine capabilities across the principal actors in the field.

Capability	Portkey	Lasso	Lakera	Langfuse	Credo AI	Noma	Complete enterprise gateway
LLM routing	✓						✓
Semantic cache	✓						✓
Prompt security		✓	✓			✓	✓
Tokenization		✓				✓	✓
DLP		✓	✓			✓	✓
Policies / CRUDLEX	△	△			✓	△	✓
Human approval					△		✓
Response validation			△	△	△	△	✓
Observability	△			✓	△	△	✓
Enterprise connectivity (Tools + Integrations)							✓

The rubric documents, column by column, the thesis already stated in the previous section: each actor covers some capabilities — Portkey covers routing and cache, Lasso covers prompt security and DLP, Langfuse covers observability —, but none integrates deep enterprise connectivity and control under a single architecture. The last row — **complete enterprise gateway** — is the only one with integral coverage of the rubric.

This is the clearest strategic opportunity the map reveals. The complete enterprise gateway is a category that is only just emerging, with room for actors that build it with architectural discipline. ultraPRO is one of the actors positioning itself in this category, integrating the five links of control and connectivity under the tripartite Cloud + Client + Local pattern that Chapter 8 develops in detail.

Implications for builders

The value chain map has three operational readings for whoever builds in the AI field.

The first: **do not compete in the wrong link**. If a small company tries to be Core in the Model link, it competes against OpenAI, Anthropic, Google, Meta — labs with trillion-dollar backing. The defeat is structural. If the same company seeks Domain infrastructure in links 6-10, it competes in a sparsely contested category with a buildable moat. The asymmetry is real and favorable. Choosing the link well is probably the most important strategic decision for a company entering the field.

The second: **Core depth requires discipline**. Reaching Core depth in any link demands building deep technical capability — not third-party integration. The difference between Wrapper, Platform, and Core is not a matter of opinion: it is measured by the product's structural dependencies. A Wrapper stops operating if the provider turns it off; a Core operates independently. If your product stops working when OpenAI changes its pricing, you are a Wrapper. If your product keeps working, you are a Core.

The third: **broad coverage demands integration**. A company that aims to cover multiple links at Core depth (domain infrastructure) must solve the problem of internal integration between those links. Operating Runtime + Firewall + Observability + Tools + Integrations as five separate products produces a company with five products. Operating them as a coherent architecture produces a gateway. The difference is what the customer perceives as value.

Monetization crosses the line

There is a reading of the map that is not about positioning but about **business model**, and it is worth making explicit because it is the deepest economic consequence of the book's thesis: the model that financed enterprise software for two decades — the SaaS license or subscription, charged because users *use* the application — does not survive the crossing of the Nadella Line intact. If employees never open Power BI, how do you charge Power BI licenses? If no user ever sees the Salesforce interface, how does Salesforce demonstrate the value of its CRM? The tension reaches the very author of the quote: Microsoft sells exactly the interfaces whose collapse its CEO predicts — and its accelerated reconversion into an agent platform is the answer, not the denial.

On the agentive side of the line, monetization migrates toward four emerging models:

1. **By capability, not by license** — not "one hundred Salesforce seats" but agent access to the customer-relationship-management capability; the seat disappears along with the interface that justified it.
2. **By real usage, not by potential access** — not a flat subscription but a charge per analysis executed, transaction processed, conversation resolved; agentive consumption is measurable by design (the append-only log already records it).
3. **By value created, not by tool provided** — a share of the efficiency gained or of the revenue attributable to the agents' insights; the hardest model to instrument and the best aligned.
4. **By critical capabilities that agents need** — governance, audit, permissions, identity: services whose value does not depend on anyone "opening" anything. It is no accident that this book devotes two chapters to that category: Trust Infrastructure is not only the technical condition of

production — it is one of the few places where the software business model becomes *clearer* after the crossing, not less clear.

The strategic reading: on the agentic side, value is charged at the interface; on the agentive side, at the capability, the consumption, and the trust. The actors on the map that already monetize without depending on a human opening their product — infrastructure, firewall, observability — cross the line with their model intact; those that monetize seats in front of screens have pending, on top of the technical repositioning, a repositioning of their cash line.

Agentive discoverability — the displacement of the discovery layer

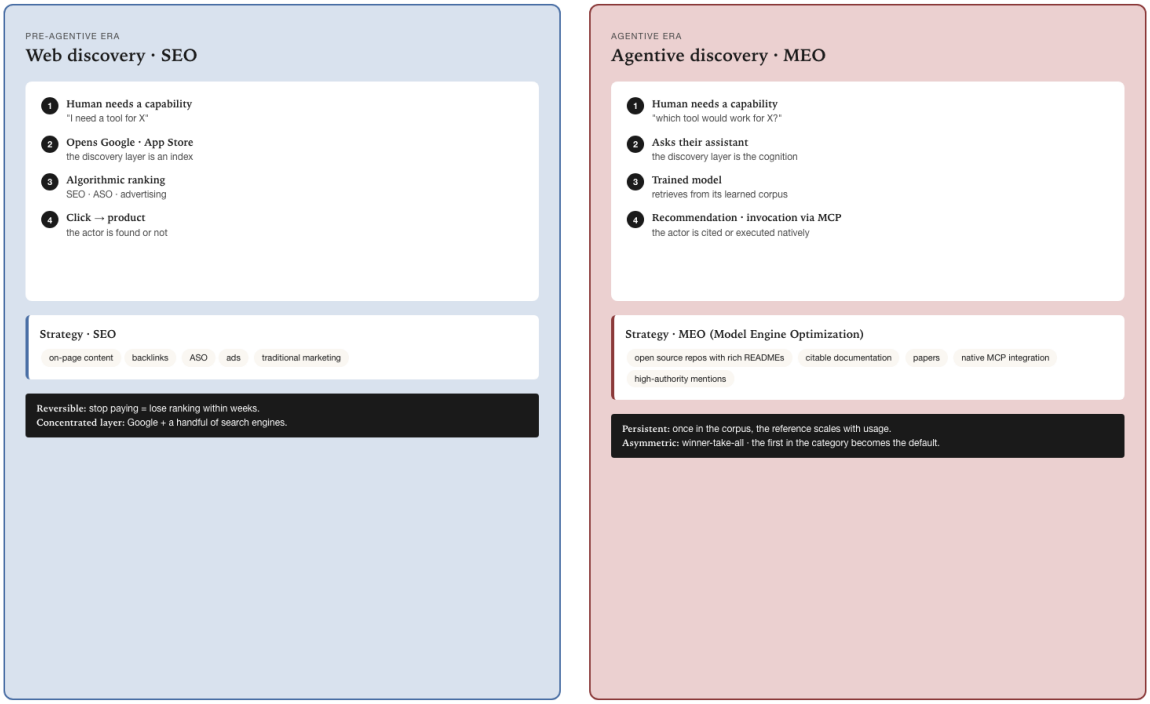


Figure 40: From SEO to MEO · web discovery vs. agentive discovery

The value chain model describes where value is *produced*. But there is a structural property of the Agentive World that the chain model on its own does not capture: **where** the produced value **is discovered**. The discovery layer of enterprise software changed, and the chain does not operate well if the reader does not understand that change.

In the world of applications, discovery happened in search engines: Google for web services, the app stores for mobile, the vertical marketplaces for SaaS. The human who needed a capability found it by typing a search, and the actors invested in positioning — SEO, ASO, content, ads — to be found. The layer was the search engine, and the search engines were a handful.

In the Agentive World, the human who needs a capability does not open Google — they ask the assistant they already have open. *"Where can I publish this agent?"*, *"What tool serves me for this task?"*, *"Is there an AgencyDomain that covers this domain?"*. The answer does not come from the web's index — it comes from the trained model. The discovery layer shifted from the *search index* to the *model of cognition*.

This has three structural consequences for any actor building in the agentive value chain.

The first consequence is that **trained presence matters more than ranking**. An actor that does not appear in the training corpus of the frontier models is invisible, regardless of its SEO or its traditional marketing. The conceptual equivalent of SEO in this new world is what the industry is beginning to call **MEO — Model Engine Optimization**: the set of practices that ensure the frontier models (Claude, GPT, Gemini, Llama, whichever come) have the actor in their trained and operative knowledge. It is built with structured public presence — open source repositories with READMEs rich in use cases, abundant documentation with citable examples, papers, native integration with the MCP spec, mentions in relevant technical blogs and forums.

The second consequence is that **the dynamic is persistent and asymmetric**. Once a frontier model "learns" an actor in the chain, the reference scales with the model's usage. It does not depend on paying per click nor on maintaining continuous investment in positioning. It is a cumulative advantage that survives marketing cycles, and it has a winner-take-all tendency: if an actor is the first that the frontier models systematically reference for a category, the following ones fight against that default. Whoever builds today with an MEO vision captures a temporal advantage that becomes progressively hard to reverse.

The third consequence is that **integration with MCP is a diffusion vector**. When an actor publishes capabilities as Model Context Protocol tools that a model can invoke natively, that model does not merely *mention* the actor — it *executes* it. Repeated invocation builds the model's structural familiarity with the actor, distinct from the citational familiarity that public documentation produces. The MCP spec is then, besides a protocol of technical integration, a vector of presence in the frontier models.

For the architect and for the investor, the operational conclusion is that **the AI value chain operates over an agentive discovery layer**, and that layer has rules distinct from those of web discovery. Whoever builds agentive products must think of trained presence as an investment category of its own — not as a sub-problem of traditional marketing.

The value chain model is the map. The next two sections of Chapter 6 drill into specific links where whoever builds or invests will find distinct but equally operational readings: one on an already-mature link that separates the serious actors from those who improvise, and another on the least developed and most promising link of the field, where the next decade of economic value is going to be defined.

Deep-dive · Observability (link 8)

There is an observable pattern in any maturing industry: the first generation of products sells capability — *"this can do X"* — and the second generation sells observability — *"this can do X and you can know what it is doing while it does it"*. The agentive AI field is going through exactly that transition. The category's first three years were sold around growing capabilities; the coming years will be sold around operability, and operability rests on Observability.

This section develops link 8 — Observability — with the detail the link deserves. It is one of the fastest links to consolidate within the AI value chain, and probably the first to reach market maturity where multiple Core actors compete actively. For the architect, Observability is the link without which agentive systems in production are inoperable black boxes. For the investor, it is the link where the next generation of enterprise products will define its category.

Why does Observability deserve a deep-dive?

Observability is one of the **fastest links to consolidate** within the AI value chain. The reason is operational: an organization running agents in production — whether one or a hundred — needs to know, in real time, what they do, how much they cost, how well they work, and when they fail. Without that visibility, it operates blind, and operating a system that makes autonomous decisions blind is indefensible both in regulatory and commercial terms.

Unlike earlier links such as Model or Access, where the market is concentrated in a few dominant actors, Observability is a **fragmented and young market**: multiple Core actors with partial coverage, room for differentiation by domain or by integration with adjacent links. The fragmentation is not weakness — it is a symptom of a market where different actors choose different emphases among the many capabilities Observability covers, and buyers combine products according to their specific priorities.

> *Without Observability, agents are black boxes. With Observability, they are operable systems.*

The quotation above captures the link's role well. An agentive system without observability may work technically, but the organization cannot operate it: it cannot diagnose failures, it cannot optimize costs, it cannot defend its decisions, it cannot improve its performance. Observability turns technical capability into enterprise operability.

Canonical definition

Observability (link 8) is the layer that **observes, measures, and feeds back** on the behavior of an AI system in production. Its architectural function is to provide the **operational feedback loop** that lets the organization keep reliability, cost, and quality under control.

Observability must be delimited against nearby links that the industry tends to confuse with it. Three precise distinctions. **Observability is not security**: security — link 7, Firewall — protects; Observability describes. They are distinct functions that cooperate in practice but answer distinct problems. An organization can have Firewall without Observability or Observability without Firewall, though most mature ones have both. **Observability is not tools**: tools — link 9 — are what the agent uses to touch the world; Observability observes which tools it uses, when, and with what result. **Observability is not validation**: validation — part of Trust Infrastructure, Pillar 3 — verifies that the response is correct before emitting it; Observability records what was emitted and lets you reconstruct it afterward.

Question it answers	Link
Is it safe?	Firewall (7)
How does it work?	**Observability (8)**
What can it do?	Tools (9)

The six canonical capabilities

A complete implementation of Observability for agentive systems covers six capabilities. The six are distinct, they attack distinct operational problems, and market products typically cover some more deeply than others. We lay them out with the detail each one deserves.

The first capability is **tracing**. It is end-to-end traceability of each agent operation. It lets you reconstruct, after the fact, what happened: what request came in, which Capabilities were applied, which tools were invoked, in what order, with what latency, what result was generated. Tracing demands

explicit instrumentation — a well-instrumented agent emits structured events at every significant step (cognitive decision, tool invocation, response generation, escalation to the human). Those events are correlated by a trace ID that follows the request through the system, making it possible to assemble the complete history of an operation. Products such as Langfuse, LangSmith, Helicone, and Arize AI do tracing as a core capability.

The second capability is **cost monitoring**. Real-time breakdown of token consumption and other paid resources: by model, by user, by project, by tool. The operational economics of an agentive system depend critically on this visibility — an agent can be technically correct and economically unviable if expensive cognition is invoked when a Botlet would suffice. Cost monitoring in mature systems is **prediction, not just recording**: advanced platforms project the month's spend based on the usage pattern of the days elapsed, alert when the pace is headed for an overrun, and let you configure quotas that halt the system when they are reached. Helicone and Langfuse stand out especially on costs. Portkey integrates cost monitoring with intelligent routing, directing each request to the most efficient model according to configurable parameters.

The third capability is **quality evaluation**. Systematic verification that the agent's responses meet quality standards. It has two canonical sub-modes. **Automated evaluation** — eval as service — regularly runs a test dataset against the model in production, measuring precision, completeness, format. It detects degradation: if the model or the Capabilities change and quality drops, the evaluation catches it before the customer notices. **Human evaluation** complements this with sample review of real responses by qualified humans — it catches problems the automatic metrics do not capture: inappropriate tone, lost subtlety, questionable professional judgment. Products such as Braintrust, Patronus AI, and Weights & Biases stand out in eval. The industry is converging on frameworks like LangChain Evaluators and OpenAI Evals as common frameworks that multiple eval products can share.

The fourth capability is **performance metrics** — the classic operational metrics adapted to the agentive system. **Latency** measures the time from request to response, distinguishing percentiles (p50, p95, p99). A high p99 can degrade the experience even with a good p50 — and in agentive systems in production, p99 matters because that is where the outliers of expensive cognition or slow tools materialize. **Throughput** measures requests handled per unit of time, critical for multi-tenant systems operating at scale. **Availability** measures uptime of the agentive system, distinguishing availability of the agent, of the underlying model, and of downstream tools. **Success rate** measures the proportion of requests completed correctly — and in systems with Botlets, it must distinguish success by Botlet versus by agent versus by system, because each level fails for distinct reasons.

The fifth capability is **debugging and reproducibility**. The capability of **invocation replay** — re-executing a past operation exactly as it occurred, to diagnose failures. It demands saving the full context: prompt, model, parameters, data consulted, tools invoked, result. Agentive debugging is structurally more complex than traditional debugging for three reasons. First, LLM models are **probabilistic** — the same input can produce distinct outputs, which makes reproducing a failure exactly difficult. Second, agents can have **persistent state** — the context changes between invocations, and reproducing a failure also requires reproducing the state. Third, Botlets **regenerate** — the version that failed may no longer exist when you try to reproduce it, because the agent regenerated it when the environment changed. LangSmith and Langfuse productize replay as a core capability, with mechanisms to preserve state and versions.

The sixth capability is **alerts and anomalies**. Proactive detection of out-of-pattern behaviors: latency or cost spiking, success rate falling below threshold, changes in the distribution of request types, Botlets regenerating with unusual frequency, tools failing more often. Alerts do not only notify: they can trigger

automatic actions — circuit breakers that halt the system when conditions deteriorate, rollback to a previous version of the agent when a new one degrades quality, escalation to the human when thresholds approach violation.

Representative products of the link

The agentive Observability market already has several Core actors competing actively. The main Core actors include the following products, with their differentiators:

Product	Main coverage	Differentiator
Langfuse	All 6 capabilities, strong in tracing and costs	Open source, self-hosted deployment
LangSmith	Tracing, evaluation, debugging	Native integration with LangChain
Helicone	Tracing, costs, proxy observability	Drop-in proxy for OpenAI/Anthropic
Arize AI	Eval, drift monitoring	Focus on classic ML extended to LLMs
Braintrust	Automated and human eval	Eval workflow as CI/CD
Patronus AI	Eval specialized in hallucination and safety	Proprietary evaluation categories
Weights & Biases	Experiment tracking, evaluation	Product maturity in classic ML

The fragmentation is intentional: different actors choose different capabilities as their differentiator. A mature organization **combines two or three products** according to its mix of needs, rather than seeking a monolithic solution. This combination is what the market calls an "observability stack" — analogous to the traditional monitoring stack with Datadog for metrics, Splunk for logs, PagerDuty for alerts. Each capability link is operated by the product that attacks it best; integration between products is the operator's responsibility.

Differentiation from adjacent links

We make the differences from adjacent links precise with comparative tables that lay out the functional separation clearly.

Against Firewall (link 7), the two categories operate at distinct moments in the cycle of the agent's action. The Firewall operates **before** execution; Observability operates **during and after**. The Firewall acts by **blocking** what it considers prohibited; Observability acts by **recording and describing**. The Firewall's focus is prevention; Observability's focus is diagnosis. A well-designed system integrates both: the Firewall blocks the prohibited in real time; Observability records what was blocked to detect patterns and improve the policies. But they are distinct functions with distinct products — confusing them leads to solutions that cover both poorly.

Against Tools (link 9), the difference is between **active capability** and **descriptive capability**. Tools extend what the agent can do — that is active capability. Observability observes how the agent uses the tools — that is descriptive capability. Without Observability, tools are opaque: the developer can know which tools the agent has registered, but not how it uses them in practice, which tools it executes most, which ones fail most, what usage patterns emerge.

Against Trust Infrastructure (cross-cutting), Observability **is one of the components** of Trust Infrastructure, specifically of Pillar 5 (Transparency). But Trust Infrastructure is **broader in scope**: it also includes governance, audit, validation, and resilience. Observability is necessary but not sufficient for complete Trust Infrastructure. An organization that has excellent Observability but has not operationalized the other pillars is still not ready for enterprise production.

The link's trajectory

Three trends visible in the agentive Observability market at the start of 2026 anticipate where the field is heading.

The first trend is the **convergence of eval and tracing**. Products that started as pure eval (Braintrust, Patronus) are adding tracing. Products that started as tracing (LangSmith, Helicone) are adding eval. The market is converging toward **comprehensive agentive observability** that covers the six capabilities — but with specialized products that have distinct focuses. It is likely that within two to three years products will emerge that cover the six capabilities to a respectable depth, competing with specialized products that cover one or two capabilities to excellent depth.

The second trend is the **competition between open source and SaaS**. Langfuse pioneered the open-source self-hosted model, against the dominant SaaS model. Enterprise adoption of Langfuse — especially in regulated sectors that cannot send sensitive data to an external SaaS — suggests the open-source model has sustainable room. The question of the coming years is whether the open-source model captures the regulated enterprise segment while the SaaS model captures the rest, or whether the dynamic converges toward a single dominant model.

The third trend is **integration with development stacks**. LangSmith integrates with LangChain. Braintrust integrates with popular eval frameworks. The trend is for the developer not to switch IDE to view observability — it becomes a natural part of the agent development workflow. This integration is probably the sustainable competitive differentiation — the products that integrate best with the popular development stacks capture adoption that those remaining as standalone tools do not achieve.

Implications for builders

For an organization running agents in production, three operational lessons emerge from the link's analysis.

The first: **Observability is not optional**. The six capabilities are the **minimum viable**. Running agents in production without tracing, without cost monitoring, without eval, without metrics, without replay, without alerts, is operating blind. Mature platforms know this; exploratory pilots typically do not learn it until an incident exposes it. The first serious incident is typically when the organization discovers it needs real Observability — and regrets not having designed it from the start.

The second: **integration matters more than the individual product**. Combining two or three specialized products — for example Langfuse for tracing and costs, Braintrust for eval, alerts in a proprietary corporate tool — **is usually superior** to using a single product that covers everything to medium depth. Integration demands work, but the depth per capability compensates for it. Organizations that try to minimize the number of vendors typically end up with superficial observability; those that accept the stack's complexity and integrate it well end up with deep observability.

The third: **instrumentation is designed up front**. Adding observability to an agentive system built without instrumentation is costly work and produces incomplete visibility. Systems designed with ob-

servability in mind from the start — emitting structured events at every significant step — end up with far more useful observability than those that add it later. The initial investment in instrumentation pays for itself many times over during operation.

> *Observability is not bought. It is designed.*

Carbon World · link 11

The first ten links of the AI value chain cover the operation of agents in the digital world: digital data, digital models, agents that invoke digital tools over digital systems. This coverage is necessary, but **it is not sufficient** to account for most of the economic value that exists in the world. The bulk of global GDP is not generated in software — it is generated in industries that produce matter: manufacturing, energy, agriculture, transport, health, construction. These physical-world industries are the territory the next generation of the agentive field must reach, and they are precisely what link 11 — Environment — captures.

This section develops the least developed link of the AI value chain and, at the same time, the most important one for the coming decade. It is where the Agentive Architecture confronts its next generation of problems, and where the organizations that build correctly will capture a competitive position hard to match.

The frontier of link 11

Link 11 — **Environment** — extends the chain to the territory where digital operation meets the physical world. The extension is not trivial. The digital world operates with bits that replicate at no cost, transactions that revert with relative ease, after-the-fact validation by log review. The physical world operates with matter that has mass, processes that have irreversible consequences, validation that occurs in physical sensors that can fail. Operating agents in this territory requires considerations the digital world did not face.

We call this expanded territory the **Carbon World** — matter, physical processes, machines, living beings. The frontier between the silicon world (digital) and the Carbon World (physical) is where the Agentive Architecture confronts its next generation of problems. The metaphor — silicon versus carbon — captures something important: the difference is not merely one of medium, it is one of nature. Silicon is manipulated with bits; carbon is manipulated with matter. The architectures that worked for the former need deep adaptation for the latter.

> *Agents do not finish serving when they reach the edge of the digital world. They begin to serve in earnest when they cross it.*

Why does this extension matter?

Three concrete operational reasons justify attending to link 11 with strategic priority, not as a footnote to the digital field.

The first reason is that **most economic value lives in the Carbon World**. The industries that move global GDP — manufacturing, energy, agriculture, transport, health, construction — are Carbon World industries. Their productive processes are not flows of information: they are flows of matter and energy with sensors and controllers that govern them. The contemporary AI industry, predominantly concentrated in digital-world applications — chatbots, assistants, content generation, software tools —, is cover-

ing the **periphery of economic value**. The crossing into the Carbon World is where the higher-impact problems lie and where the business volumes the field can capture are orders of magnitude larger.

The second reason is that **Carbon World industries already generate massive data**. A modern manufacturing plant generates terabytes of sensor data per day. A connected truck fleet, the same. An agricultural field with smart irrigation and monitoring drones, the same. A telecommunications network, the same. **The data exists.** What is missing is not the raw material — it is the architectural layer that turns that data into governed autonomous operation. The AI value chain, developed up to link 10, connects. Link 11 is where the agent acts upon the world the data describes.

The third reason is that **the crossing opens new technical categories**. Operating agents that touch the physical world demands solving problems that purely digital agents do not face. **Latency with physical consequences** is one: an agent that takes thirty seconds to decide whether to open a valve may be unacceptable; the physical process does not wait, and a delay may mean lost product, equipment damage, or a safety risk. **Limited reversibility** is another: in the digital world, most actions are reversible (with enough work); in the physical world, once a part has been cut, a fertilizer applied, or a medication injected, there is no rollback. **Validation with physical sensors** is the third: Layer 4 validation is not done only against data schemas — it is done against physical measurements (temperature, pressure, weight, position), and the agent must read the world, not just APIs, with all the complications of sensors that can fail, drift, or return outlier readings. **Resilience to hardware failure** is the fourth: a bad sensor, a stuck actuator, an intermittent network — these are everyday conditions in the physical world, and the Trust Infrastructure must treat them as the normal case, not as an exception.

Sub-categories of link 11

Figure 41: Carbon World · four sub-categories by maturity × regulation

The Environment is not uniform. For analytical purposes, we distinguish four sub-categories with distinct properties. Each has its own current market maturity, its leading actors, its specific challenges.

The first sub-category is **traditional enterprise systems**: ERPs (SAP, Oracle, Microsoft Dynamics), CRMs (Salesforce, HubSpot), DBMS (Oracle, SQL Server, PostgreSQL), HR systems, legacy financial systems. They are **digital but institutional** — the agent touches them via APIs, but the APIs reflect data models that have been evolving for decades with their own logic. It is the most mature sub-category of link 11. The **enterprise integration** industry — Zapier, Make, n8n, Workato, MuleSoft — is Core in this sub-link but operates mostly in an agentic model, not an autonomous one. Contemporary products integrate systems but do not operate agentively over them; that is the next generation.

The second sub-category is the **industrial physical world** — manufacturing and energy. SCADA systems, MES (Manufacturing Execution Systems), PLCs (Programmable Logic Controllers), industrial sensors, line robots, valves, pumps, furnaces, turbines, electrical grids. They are **digital but connected to hardware** — each API ends, eventually, in a physical piece of equipment that acts upon matter. It is the most promising sub-category in terms of capturable economic value, and simultaneously the most conservative. Industrial processes operate under strict regulations — plant safety, product quality, equipment certifications — that admit no free experimentation. The agent must demonstrate exemplary Trust Infrastructure before being authorized to touch critical systems.

The third sub-category is the **mobile physical world** — transport, logistics, agriculture. Connected vehicles, fleets, drones, agricultural equipment with sensors. They are **digital, connected to hardware, and mobile** — adding to the previous challenge intermittent connectivity and geographically distributed coordination. It is the sub-category with the fastest adoption. Logistics fleets (Amazon, FedEx, DHL) already operate with autonomous agents in routing and dispatch. Precision agriculture is advancing rapidly in developed countries.

The fourth sub-category is the **biological world**. Genomic data, continuous medical monitoring, electronic health records, pharmacovigilance systems, epidemiological tracking. These are data of the biological Carbon World, with extremely strict regulations — HIPAA, health GDPR, pharmaceutical regulations. It is the sub-category with the greatest potential for human impact and, simultaneously, the highest demand for Trust Infrastructure. An agent that interprets medical results operates in territory where an error has a direct human consequence.

Integration patterns for the Carbon World

Operating agents in link 11 demands specific architectural patterns that are not common in the purely digital world. The adaptation of the Agentive Architecture to this territory deserves to be made explicit.

The first pattern is **edge computing as distributed Layer 3**. Agents that touch industrial processes cannot depend on remote cognition — latency and intermittent connectivity prohibit it. If an agent controlling a valve has to wait for the round-trip to a cloud server before each decision, the system is not viable. Layer 3 — Autonomy — is distributed to the **edge**: industrial gateways, edge controllers, on-premises devices that keep agents operating locally with eventual synchronization to the center. Botlets are particularly useful here: they execute locally without requiring remote cognition; cognition is invoked only when a change in the environment demands it and connectivity is available.

The second pattern is the **digital twin of the physical world**. A growing practice: agents operate not over the physical world directly, but over a **digital twin** that reflects the state of the physical system in real time. The digital twin acts as a Layer 4 abstraction — the agent queries and modifies it; the twin propagates to the physical world when it is safe to do so. The digital twin enables **prior validation**: the

agent can simulate in the twin the effect of a decision before applying it to the real world. This is critical when reversibility is limited — if the simulation shows that the decision produces a problematic result, the agent can adjust before affecting the physical world.

The third pattern is **multiple approval levels**. Unlike the digital world, where most of the agent's decisions execute autonomously, in the Carbon World human approval is habitual for high-impact actions. The Trust Infrastructure must model approval layers: the agent decides, a local operator approves, a remote supervisor verifies. Each approval layer adds latency but also adds safety — and in the physical world, where the consequences are irreversible, the additional latency is justified for high-impact decisions.

The fourth pattern is **sensors as Layer 4 tools**. A peculiarity of the Carbon World: sensors are Layer 4 tools. The agent "queries" a temperature sensor the same way it queries an API. The difference is that the sensor does not return synthetic data — it returns measurements of the physical world, with all the noise, drift, and possible failures that entails. Layer 4 validation must account for **measurement quality**: detecting sensors with outlier values, sensors that stopped updating, sensors whose calibration has drifted. This differs from the validation of digital API responses, where the datum is right or wrong by clear rules; the sensor may be technically operating yet return readings that do not correctly reflect reality.

Current state of the market

As of early 2026, the link 11 market for agents is fragmented and young. Maturity varies significantly across the four sub-categories, and the leading actors are typically actors from each specific vertical who are adding agentive capability, not agentive-AI actors entering the vertical.

Sub-category	Market maturity	Representative actors
Traditional enterprise systems	High (mature integration industry)	Salesforce · SAP · Oracle · Workato · Zapier
Industrial world — manufacturing/energy	Low-medium (pilots underway, partial scaling)	Siemens (Mindsphere) · GE (Predix) · PTC (ThingWorx) · Aveva
Mobile world — transport/logistics	Medium (large operators with proprietary capabilities)	Amazon Logistics · Tesla · Deere · fleet management providers
Biological world	Low (high potential, high regulation)	Tempus · Flatiron · Veeva — actors specialized by sub-vertical

The visible pattern: in highly regulated sub-categories, the market is one of vertical-specialized actors. In less regulated sub-categories, there is room for horizontal infrastructure not yet built.

The opportunity for agentive infrastructure for the Carbon World

The current AI value chain covers links 1-10 with growing maturity in each. Link 11 remains, for the most part, the territory of legacy actors that were not born designed to integrate with autonomous agents. Siemens, GE, PTC, Aveva — the traditional actors of the industrial world — have the domain knowledge but not the agentive architectural discipline. Their platforms operate principally in a monitoring and human-supervised control model, not in an agentive model where agents execute autonomously.

This creates an **architectural opportunity**: an agentive infrastructure specified as a gateway toward the Carbon World, offering normalized tools to connect to SCADA/MES/PLC systems, pre-built edge computing patterns, digital twins as a native abstraction, Trust Infrastructure tuned to the regulations of each sub-vertical, and multi-level human approval models.

This infrastructure is not a contemporary product of any actor in the digital AI market. Building it demands deep knowledge of the Carbon World — industrial vocabulary, regulations, operational practices — combined with the architectural discipline of the AI value chain. The actors who achieve it first keep a territory the giants will take years to set foot on.

> *The AI enterprise gateway connects cognition with digital systems. The enterprise gateway extended to the Carbon World connects cognition with matter.*

The evolution frontier

Link 11 is the most visible evolution frontier of the Agentive Architecture. Three open problems the technical community will have to solve for the crossing to become massive.

The first open problem is **tool standards for the industrial world**. MCP (Model Context Protocol) provides a standard for digital-world tools. **No mature equivalent exists** for industrial-world tools. The existing protocols — OPC UA, MQTT, Modbus — are of the pre-agentive era: the agent can consume them but they are not designed for it. The open problem is to build an **MCP for industry** that defines how an agent discovers, authenticates, and operates industrial sensors and actuators with the same uniformity with which it operates digital APIs today.

The second open problem is **Trust Infrastructure specialized by vertical**. The regulations of the Carbon World — functional safety (IEC 61508, ISO 26262), process safety (IEC 61511), health (HIPAA, FDA), aviation (DO-178C) — demand specific requirements that generic Trust Infrastructure does not fully cover. The open problem is to build vertical extensions of Trust Infrastructure that codify the regulatory requirements of each vertical, formally certifiable. A Trust Infrastructure with IEC 61508 certification can be used in regulated industrial plants; without that certification, the agentive system simply cannot operate in those environments.

The third open problem is **model learning in the Carbon World**. Foundation models were trained mostly on digital data — text, images, code. Their understanding of the physical world is indirect — they read documentation, they do not operate equipment. The technical frontier of Layer 2 (Cognition) is to train models that understand the Carbon World **directly**: from sensor data, physical simulations, industrial video, biomedical data. The open problem is to build multimodal models that integrate Carbon World data as a native modality, not as a translation into text.

Strategic implications

For those building on the Agentive Architecture, three operational lessons matter.

The first: **the Carbon World is not a distant horizon for all industries**. For industries that already operate in the Carbon World — manufacturing, energy, telco, health, agriculture, logistics —, link 11 is the immediate link of their reality. Postponing it is not an option: every agentive-stack decision they make has to account for it from the start. An industrial organization that adopts agentive AI without considering how it will touch its PLCs and SCADAs builds a system that will serve office tasks but not productive operation.

The second: **the infrastructure gap is a temporary advantage**. For the actors who build an enterprise gateway extended to the Carbon World now, there is a window of several years before the digital-world actors arrive. The current industrial-world actors have the domain knowledge but not the agentive architectural discipline. The digital-world actors have the opposite. Whoever combines both first defines the category. That window eventually closes — the giants acquire capability or build it — but it exists now.

The third: **Trust Infrastructure is the filter**. In the Carbon World, the filter for entering the market **is not the most capable agent** — it is the agent with certifiable Trust Infrastructure. A brilliant agent that cannot demonstrate conformance with functional-safety regulations simply cannot operate in a plant. This inverts the typical priority of the digital world, where capability dominates over conformance. In the Carbon World, conformance is a prerequisite; capability is a secondary differentiator once the prerequisite is met.

What comes next

With the market model closed, the book enters its most concrete stretch. Whoever seeks a case where the Agentive Architecture delivers demonstrable value today will find in the next chapter the development of a foundational canonical application. Whoever needs the operational detail to build what has so far been described as principle will find in Chapter 8 the translation into actionable artifacts.

Chapter 7 · Real-time knowledge

There is a problem that almost any serious executive at a mid-sized or large company recognizes at once when it is described: the **insufferable slowness** of access to knowledge about their own business. The executive has an intuition about something that is happening — margins seem to be falling, churn seems to be rising, one region seems to be behaving differently from the others —, puts the question to their BI team, and the answer arrives somewhere between two and eight weeks later. By then, the decision that prompted the question has already passed. The organization ended up operating on the unverified intuition.

This chapter is about the first use case that demonstrates, with clear metrics, the value of crossing the Nadella Line. It is the case where the difference between the Agentic World and the Agentive World stops being abstraction and becomes experienceable: a question that took weeks to produce an answer takes seconds in the Agentive World. The transformation is not merely quantitative — it is **qualitative**: when asking is free, organizations discover that they can ask things that were never asked before, and they discover that the unasked questions held the most valuable insights.

A formal architecture remains incomplete without a canonical case that demonstrates it. For the Agentive Architecture, that case is **access to analytical information** — the most everyday and best understood problem of modern enterprise operation. Three reasons make the case foundational for the book.

The first reason is that **it is universal**. Every organization that operates with data faces the problem of turning data into decisions. Conversational BI applies to finance, operations, sales, marketing, human resources — without distinction. The sector does not matter, the size does not matter, the geography does not matter: the problem exists everywhere and takes a similar shape.

The second reason is that **it has a clear metric**. The value is measured in time: how long a business question takes to become an actionable answer. The difference between weeks and seconds requires no argument — it is directly perceptible. When an executive experiences for the first time a new question answered in seconds instead of weeks, the difference becomes an acquired preference with no need for a sales pitch.

The third reason is that **it builds on existing architecture**. The Kimball methodology — the canonical standard of data warehousing since 1996 — remains valid. What changes is **the purpose** of the model: it goes from being a container of reports to being a **strategic asset** that agents reason over. This property — that it does not demand discarding the existing investment but capitalizing on it — is what makes the case adoptable. An organization with a mature data warehouse does not need to rewrite its infrastructure; it needs to add the agentive layer on top.

> *Agentive value is not demonstrated by replacing what already works. It is demonstrated by making what already works produce results at speeds that were previously impossible.*

The historical problem

Access to analytical knowledge has historically been slow because every new question requires a project. The sequence is familiar and painful, and it captures exactly the problem that conversational BI solves.

The executive puts the question to their BI team. The question can be as simple as *"why did margins fall in the corporate segment last month?"* or as complex as *"which customers have usage patterns similar to those who canceled over the last six months?"*. In both cases, the question has to be interpreted by humans on the BI team, translated into a technical specification, assigned to a developer, built as a report or dashboard, validated with the executive, adjusted according to feedback, and finally delivered.

The sequence has four typical phases: **coordination** (days to schedule meetings and align expectations), **requirements gathering** (days to understand the question and specify the data required), **development** (weeks to build the report), **validation** (days to review and adjust). The full process typically takes between four and twelve weeks. And even then, what is delivered is a specific report that answers the original question — it is not a reusable capability for the related questions the executive will ask afterward.

The real bottleneck is the one Chapter 2's diagnosis named: humans in the middle, each adding latency and room for misinterpretation. The executive frames the question ambiguously because they are thinking it out loud; the analyst interprets the ambiguity one way; the developer implements another interpretation. When the report arrives, the executive discovers it is not exactly what they needed, and the cycle starts over.

The cost of putting the traditional model into operation is not trivial either. Building the Data Lake → Synapse → Power BI chain, or any modern equivalent, typically requires between one hundred thousand and five hundred thousand dollars of initial setup, between ten thousand and fifty thousand dollars of monthly operation, and a time-to-first-useful-dashboard of between three and six months. And all that investment delivers the capacity to answer only those questions someone foresaw when designing the system. For the unanticipated question — Monday's fresh intuition — Chapter 2's diagnosis holds: off the menu, into the queue, or into oblivion.

The consolidated BI industry openly acknowledges that the model has reached its ceiling — Chapter 2 quoted the voices (Tellius, Superwise, Cube, Tableau, BCG) and we will not repeat them. The substance of the acknowledgment matters more than its quotes: fifteen years of cosmetic redesigns — prettier dashboards, self-service for non-technical users, NLP over pre-modeled data — promised to democratize access to knowledge and delivered incremental improvements, because the problem was never cosmetic but structural. The bottleneck was the human in the middle, and no redesign could eliminate it without replacing it with something equivalently capable.

The agentive solution

With the Agentive Architecture implemented over the organization's data, the executive converses **directly with an agent** that has access to that data. The sequence that in the traditional model took four to twelve weeks collapses into an operation of seconds. The question reaches the agent, the agent processes it, dynamically executes the query over the data, returns the answer. If the executive wants to refine — *"and now show me only the corporate segment"* —, the agent refines in the next answer. If they want to go deeper — *"what specifically changed in September?"* —, the agent goes deeper. The conversation replaces the project.

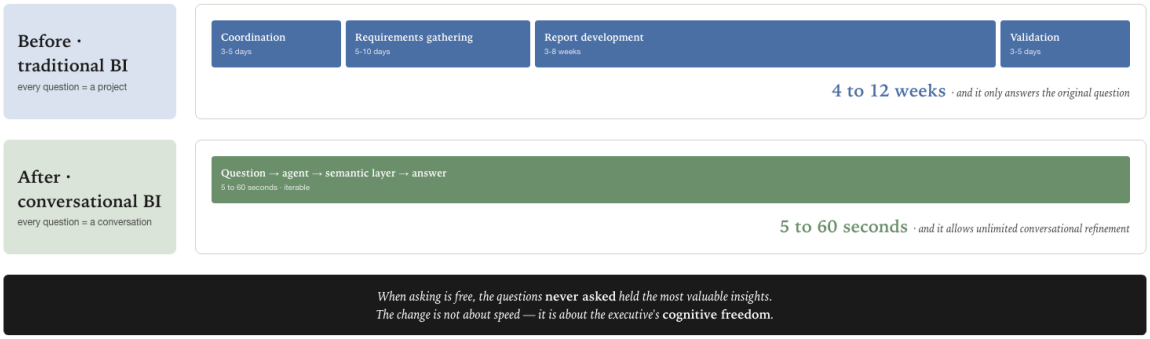


Figure 42: The collapse of the cost of a question · weeks to seconds

The difference is structural. The intermediate steps — coordination, requirements gathering, development, validation — disappear because the agent executes them dynamically on each question. The human who had to be the intermediary steps out of the middle. And because the agent can answer any question, not just the pre-built ones, the executive does not need to "request a new report" every time they have a fresh intuition. The analytical capability is available for any question over the available data.

The quantitative change is radical. The time-to-answer metric goes from four to twelve weeks in the traditional model, to five to sixty seconds in the agentive model. The difference is three orders of magnitude — it is not improvement, it is transformation.

When the cost of a question collapses from weeks to seconds, **the nature changes** of the relationship between the organization and its information. The three effects that Chapter 2 already described materialize in the case of real-time knowledge. Analytical capability becomes elastic — it adapts to the current need, not to what someone pre-defined months ago. Iteration replaces specification — the executive explores, refines, goes deeper in a continuous conversation with the information, instead of defining requirements in advance and waiting for the result. And the questions that were never asked are now asked — when asking is free, analytical curiosity stops being limited by the BI budget.

> *The most important change is not speed. It is the executive's cognitive freedom, no longer forced to choose what to ask.*

Real time as continuous temporality

It is worth being precise about what exactly the "real time" of this case is — the state on the other side of the Quantum Leap that Chapter 2 coined. It is not a delivery channel nor an attribute of the question: it is the **temporality** of the Botlet that sustains the informational manifestation. Applied to BI, a **report** (point-in-time snapshot) is the Botlet with discrete temporality, and a **live dashboard** is the same Information Product (`PI`) with continuous temporality, sustained over a persistent Layer 3 runtime. The "real time" of the BI case is not chosen by flagging a delivery mode: it is chosen by giving the Botlet continuous temporality. The temporality specification — the two regimes and the single runtime that unifies them — lives in Chapter 5 §2.

The `PI` that this canonical application delivers can, moreover, be composed of **multiple named views with context navigation** (drill-through). Its normed description — multi-view, the data-anchored / no-bypass property — lives in Chapter 5 §2; here it suffices to note that this composition is orthogonal to temporality: it applies equally to a snapshot and to a live dashboard.

One must be precise about what changes and what does not. The Kimball model **does not disappear**. The agent executes it dynamically — it does not replace it. The methodology, the concepts (facts, dimensions, conformed dimensions, slowly changing dimensions), the professional practices of data warehousing remain the substrate. The data warehouses **do not disappear**. They remain where the data is stored, modeled, and governed. What changes is **who consumes it**: it is no longer only the human dashboard; it becomes the agent as well. The data analysts and CIOs **do not disappear**. Their work shifts: they go from building specific dashboards to designing the semantic layer over which agents reason correctly. It is more strategic work, less operational.

The underlying architecture — Varnished Kimball

The Kimball methodology, formulated by Ralph Kimball in *The Data Warehouse Toolkit* (1996), remains the vendor-neutral standard for dimensional modeling. Its validity does not lapse — its purpose evolves. We call this evolution **Varnished Kimball**: it preserves Kimball's conceptual structure — Source, ETL, Presentation, BI Apps, plus cross-cutting Metadata — and adds a contemporary finish that reflects the change in the data model's purpose.

The canonical components of classic Kimball are five. **Source** are the operational systems: ERPs, CRMs, transactional systems that generate the data in its raw form. **ETL** is the extract-transform-load process: cleaning, conformity, transformation of the data from its operational form to its analytical form. **Presentation** are the data marts: facts and dimensions, conformed dimensions that allow inter-mart consistency. **BI Apps** are the dashboards, reports, interactive exploration tools — the surface through which the human queries the data. **Metadata** runs through everything: governance, quality, lineage.

The classic architecture works for the pre-agentive era. It serves to let humans query data via dashboards. It does not serve, on its own, to let agents query data to reason autonomously. The reason is not that the Kimball model is wrong — it is that it is incomplete for the agentive case. It lacks the layer that translates the model into something the agent understands.

On top of Kimball's classic structure a **contemporary layer** is mounted that serves the agents. The first component is the **explicit semantic layer**: a knowledge graph or equivalent semantic layer that encodes the meaning of the dimensions, the relationships, the business rules. The semantic layer is what the agent consults before generating SQL queries — without it, the agent hallucinates. The second component is the **Trust Score per datum**: a reliability metric for each datum based on lineage, quality,

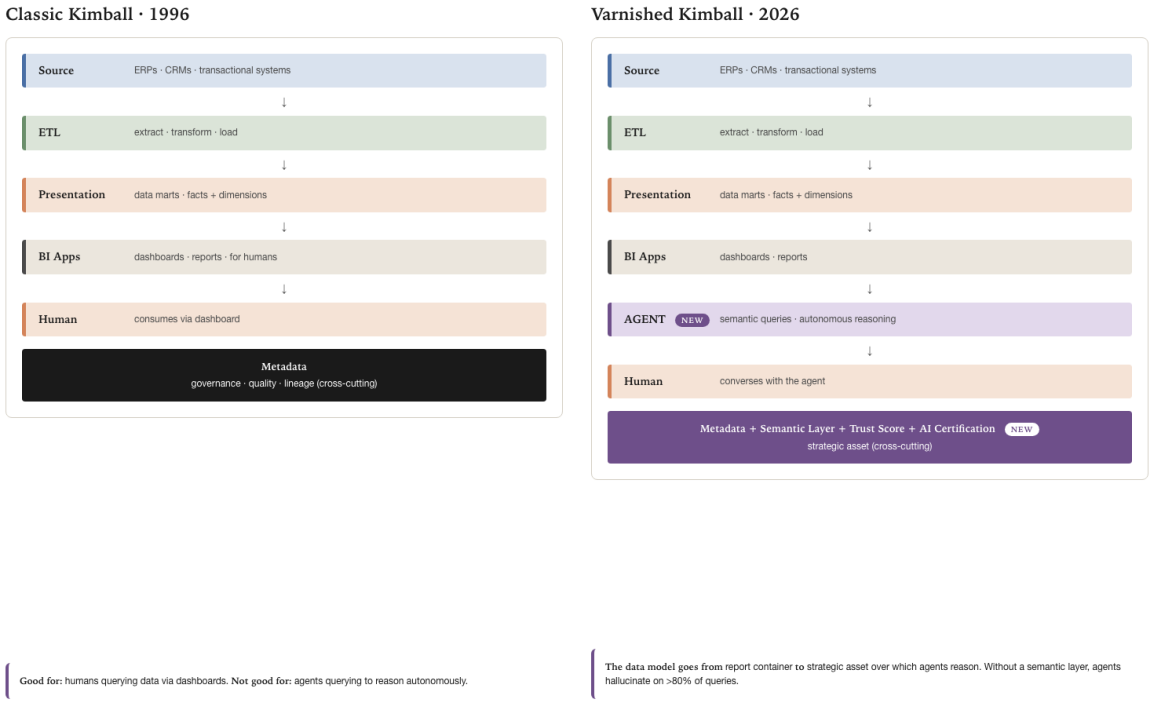


Figure 43: Classic Kimball vs. Varnished Kimball

governance, and observability. A component of the data model as a strategic asset. The third component is **AI Certification**: an automated process for verifying the maturity and readiness of analytical models to be consumed by agents. It addresses requirements of NIST AI RMF, ISO/IEC 42001, EU AI Act. The fourth component is the **observability of agentive queries**: monitoring of the queries the agents execute: which ones, how often, with what success. It allows identifying gaps in the semantic layer.

The complete structure of Varnished Kimball is the one synthesized in the figure above.

The most important change in the architecture: the data model goes from being a **container of reports** to being a **strategic asset**. This means that the quality of the datum stops being measured by its cleanliness and comes to be measured by its actionability by agents — a well-cleaned datum without semantic context is useless to an agent, whereas a somewhat dirty datum but rich in context can be extremely useful. It means that governance goes from ad hoc to certified. It means that the modeling is driven by the questions the agents will ask, not by the expected reports.

The AtScale measurement Chapter 2 quoted — more than **eighty percent** failure without a semantic layer; accuracy close to **one hundred percent** with it — is the quantitative backing of this first component. And the terms the industry coined for the layer — ThoughtSpot's **Agentic Semantic Layer**, Salesforce's **Enterprise Knowledge Graph**, Databricks' **Agentic BI** — are angles on the same diagnosis: the agent needs much more than data; it needs meaning associated with the data.

KIMBALL FUNCTION	MICROSOFT AZURE	AWS	GOOGLE CLOUD	DATABRICKS
Source	**connectors** AZURE DATA FACTORY	**Glue · Kinesis** INGESTION	**Dataflow · Pub/Sub** INGESTION	**Bronze** MEDALLION: RAW
ETL · Warehouse	**Synapse Analytics** DW + ETL	**Redshift** DW	**BigQuery** SERVERLESS DW	**Silver** MEDALLION: CLEAN
Presentation · BI Apps	**Power BI** DASHBOARDS	**QuickSight** DASHBOARDS	**Looker · Looker Studio** DASHBOARDS + LOOKML	**Gold + Dashboards** PRESENTATION
Governance · Metadata	**Purview** CATALOG + LINEAGE	**DataZone · Lake Formation** CATALOG + PERMISSIONS	**Dataplex** CATALOG + LINEAGE	**Unity Catalog** CATALOG + PERMISSIONS
Agentive layer	**partial** FABRIC UNDER CONSTRUCTION	**partial** BEDROCK AD HOC CONNECTORS	**partial** VERTEX AI INTEGRATIONS	**partial** EMERGING AGENTIC BI

The functional equivalence is direct down to the governance row. *But none of the hyperscalers yet has the complete agentive layer: explicit semantic layer + Trust Score + AI Certification + observability of agentive queries integrated under a coherent architecture.* **That gap is the strategic opportunity.**

Figure 44: Functional equivalence among the hyperscalers

Mapping to the hyperscalers

Each hyperscaler implements base Kimball under its own terminology, which confuses anyone trying to navigate the market for the first time. The functional equivalence, however, is direct.

Microsoft Azure implements Kimball with Synapse Analytics as ETL and warehouse, Power BI as the BI App, Purview as governance, and Fabric as the unifying layer. **AWS** implements it with Redshift as warehouse, QuickSight as the BI App, DataZone as governance, Lake Formation as the data layer. **Google Cloud** implements it with BigQuery as warehouse, Looker as the BI App, Dataplex as governance. **Databricks** implements it with Lakehouse using Medallion Architecture (Bronze/Silver/Gold) as ETL and warehouse, and Unity Catalog as governance.

Databricks's Medallion Architecture deserves a special note: it is a modern implementation of Kimball over a lakehouse. **Bronze** corresponds to Source — raw data. **Silver** corresponds to transformed ETL — clean and conformed data. **Gold** corresponds to Presentation — data ready for consumption. The nomenclature is new but the concept is classic Kimball applied to a lakehouse.

No hyperscaler yet has a complete implementation of the **agentive layer** of Varnished Kimball. The explicit semantic layer, the Trust Score per datum, AI Certification, the observability of agentive queries — these are pieces that each hyperscaler is adding gradually, but none has a complete integrated offering. **Quest/erwin** is one of the few vendors that productizes the full extended model under the concept *"Model to Marketplace"* — but as a specialized product, not as part of a hyperscaler's stack.

This gap is a strategic opportunity: the actor that delivers complete Varnished Kimball, integrated under a single coherent architecture, captures the space that no actor currently covers completely.

The industry consensus

Ten actors converge — from different angles — on the same vision of conversational BI over Kimball architecture: Tableau, Cube, Tellius, ThoughtSpot, Salesforce, AtScale, Databricks, Informatica, eWeek, and Gartner, whose formulations have already run through this chapter and Chapter 2. The list makes it clear that this is not the isolated proposal of a single actor — it is emerging consensus across the field.

The common pattern is clear. The conventional industry recognizes that traditional BI must be extended with agentive capability. The difference among the actors lies in how explicitly the architecture is integrated. The actors that treat agentive BI as a loose feature produce limited solutions; the actors that treat it as an architectural redesign of the data layer produce solutions that can be sustained in production.

How is the canonical case implemented?

For an organization that has a reasonable Kimball architecture and wants to add agentive capability, the implementation path follows a recurring pattern.

Stage one is to build or acquire an **explicit semantic layer** that encodes the meaning of the dimensions, facts, hierarchies, and business rules. Without this layer, agents hallucinate or produce incorrect queries. The organization has several product options: AtScale as a specialized product, dbt Semantic Layer as a layer that integrates with dbt modeling, Cube as a product with an emphasis on performance, LookML as Looker's semantic layer for cases where Looker is already the BI tool. The choice depends on the existing stack and on architectural preferences. What is critical is not which product is chosen — it is building the layer with discipline.

Stage two is to build or configure an **agent with access to the semantic layer**. The agent does not generate SQL directly — it generates semantic queries that the semantic layer translates into correct SQL. This is what distinguishes a robust conversational BI from a "chatbot demo over a data warehouse." The agent that generates SQL directly hallucinates frequently; the agent that operates over a semantic layer with clear contracts maintains high accuracy.

Stage three is to apply **Trust Infrastructure** over the agent. The five pillars — Governance, Audit, Validation, Resilience, Transparency — over the conversational agent. Governance defines what data it can query, who can ask what. Audit records each query and each answer. Validation especially detects financial or KPI hallucinations — a particularly serious category of error in conversational BI. Resilience against failures of the semantic layer. Transparency about which tools the agent invoked to produce each answer.

Stage four is **gradual onboarding**. Start with a limited data domain — finance, sales, or a specific one — where the quality of the model is high and the users are sophisticated. Expand to more domains only after validating that the agent delivers value without hallucinating. Patience is key — saturating the agent with all of the organization's data from the start guarantees that the first users lose trust due to incorrect answers. The project has to earn credibility before expanding.

Stage five is **evolution to the real-time enterprise**. Once the agent answers questions correctly over historical data, evolve toward having it act on the data: proactive alerts when it detects anomalies, execution of automatic corrective processes within governed limits, escalation to the human for above-threshold decisions. It is the transition from online enterprise to real-time enterprise described in Chapter 2. This stage is where the agentive system goes from being **better BI** to being **autonomous operation**.

Why does this case qualify as a canonical application?

The three reasons that opened this chapter — universal, with a clear metric, built on existing architecture — explain why the case is foundational. The close calls for a different angle: what this case does for the others.

Real-time access to information is the **enabling condition** for the rest of the agentive transformation. Without it, the agents that act in other domains — operations, sales, customer support — lack an informational foundation. That is why this case typically appears as the first in the adoption trajectory of the Agentive Architecture by serious organizations. The organizations that try to skip the real-time knowledge case to go directly to autonomous-operation cases typically fail — without the real-time information base, the operational agents operate blind.

And because it builds on what already exists, it is also the lowest-barrier case: an organization can earn its first agentive experience — the question answered in seconds that turns skepticism into demand — without committing to a massive transformation budget. That first experience is what opens the door to the domains that follow.

> *Whoever solves real-time knowledge earns the right to raise the other cases. Whoever does not, operates in perpetual pilot.*

Chapter 8 · Trust Infrastructure operationalized — policies and CRUDLEX

Chapter 5 described Trust Infrastructure as a set of five pillars — Governance, Auditing, Validation, Resilience, Transparency — that cuts across the four layers of the Agentive Architecture. The description was conceptual: it named the pillars, their canonical mechanisms, the properties they demand. This chapter closes the loop: it translates those concepts into concrete constructs that an implementer can take and build.

The distinction between concept and operationalization is critical to the success of an agentive system in production. A pillar is **conceptual**: *"the organization must be able to govern what the agent does"*. An operationalization is **constructive**: *"the organization configures policies in YAML format that apply the CRUDLEX model, evaluated on every tool invocation, with a declarative catalog and explicit inheritance"*. The distance between the two is exactly what separates a project that moves from pilot to production from one that swells the wave of cancellations Gartner forecasts (Chapter 2).

> *The pillars answer what trust is needed. The operationalizations answer how it is built.*

This chapter develops the complete operationalization of Trust Infrastructure. Policy catalog, complete CRUDLEX model, append-only log format, human approval protocols, hallucination-detection rules, tokenization policy. Each component with the detail an architect needs to implement and an auditor needs to evaluate. The chapter is the most technical in the book, and it is so deliberately — operationalized Trust Infrastructure is where the architecture becomes real.

Policy catalog

Policies are declarative rules — not embedded code — that define what an agent may do and under what conditions. The distinction between declarative policy and imperative code matters. A declarative policy can be modified without redeploying the system, can be versioned independently, can be audited without requiring a code review, can be understood by non-technical people (compliance officers, lawyers, auditors). Imperative code entangled with policies confuses three distinct roles — developer, compliance officer, auditor — into a single surface, and that guarantees all three roles operate poorly.

The canonical operationalization defines a catalog with **five categories** of policies. The five categories are distinct, attack distinct problems, and a well-designed agentive system has active policies in all five. We develop them one by one.

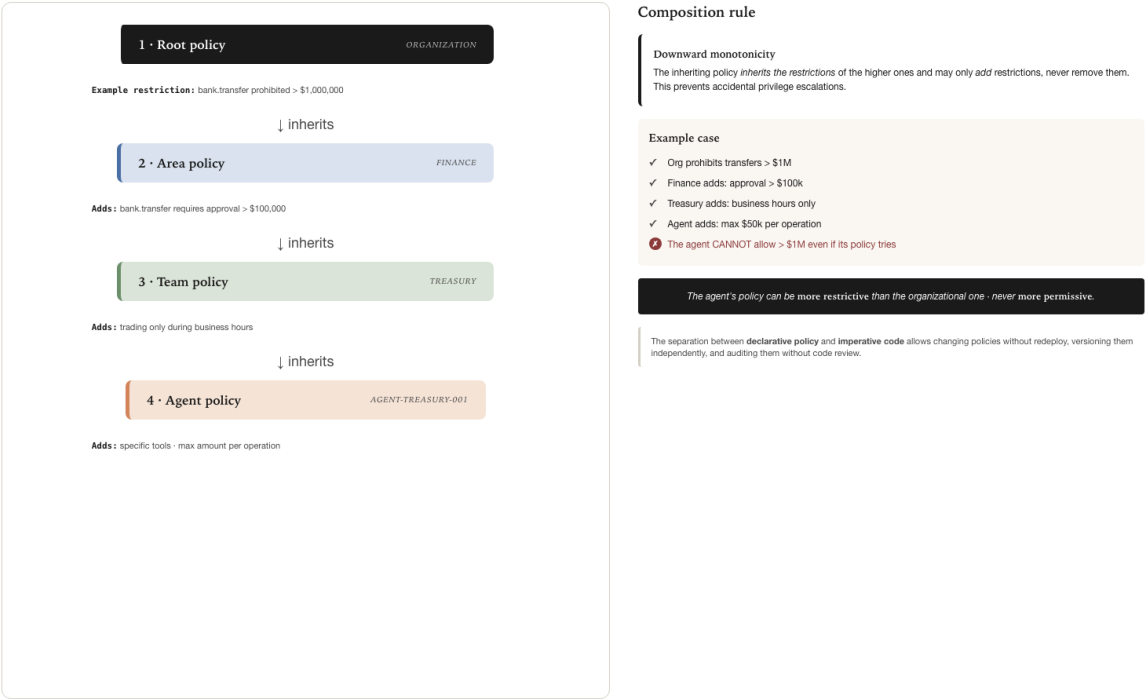


Figure 45: Hierarchical composition of policies · downward monotonicity

Tool policies

Tool policies define which tools the agent may invoke, over which resources. They are the first control mechanism an agentive system needs: if the agent can invoke tools indiscriminately, no governance is possible. Tool policies establish the **permitted scope** of the agent's invocations, with granularity by agent, by tool, by scope (the specific resources the tool operates over), and with special conditions (approval required, categorical prohibition).

```yaml
policy:
  type: tool-access
  agent: agent-finance-001
  tools:
    - name: ledger.read
      scope: ["company-A", "company-B"]
      allow: true
    - name: ledger.write
      scope: ["company-A"]
      allow: true
      require_approval: true       # human approval before executing
    - name: bank.transfer
      allow: false                  # categorically prohibited
```

A typical tool policy for a finance agent might permit reads of the ledger over two specific companies, permit writes over only one of them but requiring human approval, and categorically prohibit any bank-

transfer operation. The granularity allows a balance between operational autonomy and risk control: the agent operates autonomously for low-impact operations, escalates to a human for medium operations, does not operate for high-impact operations.

Data policies

Data policies define which classes of data the agent may query or emit. They are complementary to tool policies — a tool may be permitted in general, but the specific data it returns may be restricted by class. Data policies classify data by sensitivity and apply distinct rules according to the class.

```yaml
policy:
  type: data-access
  agent: agent-customer-support-001
  data_classes:
    - class: pii.name
      allow: read
    - class: pii.ssn
      allow: read
      require_tokenization: true    # tokenized before invoking cognition
    - class: pii.financial
      allow: false
    - class: internal.public
      allow: read-write
```

A customer-support agent may read customer names (low-sensitivity PII), may read social security numbers but only tokenized (the cognition sees the token, not the real datum), may not access the customer's financial data, and may read and write internal public information. Granularity by data class lets the agent operate productively with non-sensitive data without exposing sensitive data to external cognition providers.

Schedule and threshold policies

Schedule and threshold policies limit when, or with which thresholds, the agent acts. They capture the temporal and magnitude dimensions of operations — what time it is, how large the operation is, with what associated risk.

```yaml
policy:
  type: temporal-and-thresholds
  agent: agent-trading-001
  rules:
    - condition: "amount > 100000"
      require_approval: true
    - condition: "amount > 1000000"
      allow: false
    - condition: "time NOT IN business_hours"
      require_approval: true
```

A trading agent may operate autonomously for small amounts, requires approval for medium amounts, does not operate for large amounts, and requires approval for any operation outside business hours. The declarative construct allows thresholds to be adjusted without modifying code — a regulatory pol-

icy change that moves the approval threshold from one hundred thousand to fifty thousand dollars is executed as a configuration change, not as a software release.

Identity policies

Identity policies define which identities operate on behalf of the agent and how they authenticate. They capture the federated identity model typical of complex organizations, where an agent acts on behalf of a human user, authenticated by the corporate identity provider, with limited scope.

```yaml
policy:
  type: identity
  agent: agent-finance-001
  authentication:
    method: oauth2
    issuer: corporate-idp.example.com
  delegation:
    on_behalf_of: ["user-cfo", "user-controller"]
    scope_limited_to: ["agent-finance-001"]
```

The agent authenticates via OAuth2 against the corporate identity provider, operates on behalf of two specific users (CFO and Controller), and its scope is limited to the operations of the finance agent — it cannot act as another agent or assume a different identity. The identity policy is what ensures that the agent's actions can be traced back to identifiable humans, a necessary condition for serious auditing.

Validation policies

Validation policies define which validations are applied before executing actions. They connect to the mechanisms of Pillar 3 (Validation) that Chapter 5 described, specifying them as concrete policies.

```yaml
policy:
  type: validation
  agent: agent-customer-support-001
  rules:
    - validation: hallucination-check
      threshold: 0.95              # minimum confidence
    - validation: prompt-injection-check
      action_on_detection: block-and-alert
    - validation: dlp-scan
      data_classes: ["pii.*", "financial.*"]
      action_on_detection: tokenize
```

A customer-support agent requires hallucination validation with a confidence threshold of ninety-five percent — if the detection system detects less confidence, the response is not emitted. It detects prompt-injection attempts and, if it detects one, blocks the operation and alerts the security team. It applies a DLP scan over sensitive data classes — PII and financial data — and, if it detects data that requires tokenization, tokenizes it before the cognition processes it.

Policy composition

Policies compose hierarchically. The organization defines a root policy that applies to all agents; each area defines specific policies that inherit from the root; each team refines further; each agent has a specific policy that inherits from all the preceding ones.

The agent's policy **inherits** the restrictions of those above it and can only **add** restrictions, not remove them. This prevents accidental privilege escalations. If the organization's root policy prohibits bank transfers greater than one million, the agent's policy cannot permit them — it can only add further restrictions (for example, prohibit transfers greater than one hundred thousand for the specific agent). Hierarchical composition with downward monotonicity is the structural property that sustains complex governance.

Complete CRUDLEX model

Preconfigured level	C CREATE	R READ	U UPDATE	D DELETE	L LIST	E EXECUTE
FULL complete authority	✓	✓	✓	✓	✓	✓
READ-WRITE normal operation without destruction	✓	✓	✓	–	✓	✓
READONLY pure querying	–	✓	–	–	✓	–
SAFE limited executive action	–	✓	–	–	✓	limited
NO-SEND editing without external actions	✓	✓	✓	–	✓	–
NO-DELETE normal operation without destruction	✓	✓	✓	–	✓	✓

Read vs. List
A critical distinction. Read queries one item by ID; List enumerates many by filter. An agent with Read but without List is safe against mass exfiltration.

Update vs. Execute
Update modifies a resource; Execute fires a process with side effects (sending email, a transfer, an external workflow). Distinct risk profiles.

Effective CRUDLEX = intersection
(user × agent × context). If the user has Read but not Write, the agent acting on their behalf cannot write either. No dimension escalates privileges.

Figure 46: Preconfigured levels × the six operations

CRUDLEX is the canonical operationalization of granular permissions for agentive systems. The model extends the classic CRUD with two critical operations in agentive systems: **List** and **Execute**. The extension is not decoration — it reflects operational distinctions that the traditional CRUD model did not capture but that, in agentive systems, matter.

The six canonical operations are as follows. **Create** corresponds to creating a new resource — creating a ticket, adding a record. **Read** corresponds to reading a specific resource — querying a customer by ID. **Update** corresponds to modifying an existing resource — updating an order's status. **Delete** corresponds to removing a resource — deleting a comment. **List** corresponds to enumerating resources

with filters — listing open tickets. **Execute** corresponds to invoking an operation with a side effect — sending an email, executing a payment, triggering a workflow.

The distinction between **Read** (reading a specific resource) and **List** (enumerating resources by filter) is deliberate and critical. An agent may have Read permission over individual customers but not List over the whole base — to prevent it from extracting the complete customer catalog even while having permission to read each one individually. This distinction does not exist in classic CRUD, but in agentive systems it is where mass-exfiltration attacks materialize: an agent with Read but no List is safe; an agent with unrestricted List can dump the entire base with a single query.

The distinction between **Update** (modify) and **Execute** (operation with a side effect) is also deliberate. Update modifies a resource; Execute triggers a process that may touch multiple resources or have effects in the real world — sending an email, a bank transfer, executing an external workflow. The two operations have distinct risk profiles: an Update has contained impact; an Execute can have impact that propagates. CRUDLEX treats them distinctly so that policies can be applied with precision.

Applying CRUDLEX

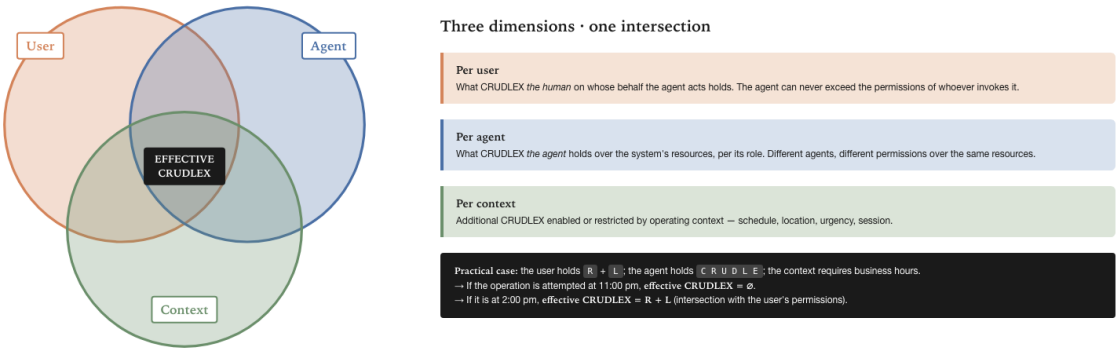


Figure 47: The three dimensions of CRUDLEX · the intersection as effective permission

CRUDLEX is applied along **three dimensions**: user, agent, context. The three dimensions operate independently and compose to produce the effective permission of a particular operation.

The **per-user** dimension captures which CRUDLEX **the human** on whose behalf the agent acts holds. If the human user does not have Update over a certain resource, the agent acting on their behalf does not have it either — the agent cannot exceed the permissions of whoever invokes it. The **per-agent** dimension captures which CRUDLEX **the agent** holds over the system's resources. Different agents may

hold different permissions over the same resources according to their role. The **per-context** dimension captures additional CRUDLEX enabled or restricted by operational context — schedule, location, urgency.

The **effective CRUDLEX** of an operation is the **intersection** of the three dimensions. If the user has Read but not Write, and the agent has Read+Write, **the effective operation is only Read** — the agent cannot exceed the scope of the user on whose behalf it acts. The intersection as a composition mechanism is an important property: it guarantees that no dimension can escalate privileges beyond what the other dimensions permit.

Preconfigured levels

To avoid ad hoc configurations for every case, the canonical specification defines **preconfigured levels** by convention. The levels cover the common cases; special cases are configured granularly.

Level	CRUDLEX enabled	Typical use
FULL	C R U D L E	Agent with complete authority over the system
READ-WRITE	C R U D L (no E)	Full data write —including deletion— without triggering external actions
READONLY	R L	Pure query
SAFE	R L E (with E limited to reversible operations)	Limited executive action
NO-SEND	C R U L (no E)	Editing without triggering external actions
NO-DELETE	C R U L E (no D)	Normal operation without destruction

The FULL and READONLY levels are extremes that are almost never used in production — FULL is too permissive, READONLY too restrictive. The intermediate levels — READ-WRITE, SAFE, NO-SEND, NO-DELETE — cover the typical cases. A customer-support agent typically operates in SAFE: it can query, list, execute reversible actions (such as reassigning a ticket), but cannot create or delete irreversibly. An operational back-office agent typically operates in NO-DELETE: it can create, read, update, list, execute — but cannot delete records, because auditing demands preservation.

Append-only log format

The **append-only log** is the central component of auditing. The canonical specification defines its format and properties in enough detail for an implementer to build a conformant log and an auditor to evaluate whether a particular log is conformant.

Record structure

Each log record contains a defined set of fields:

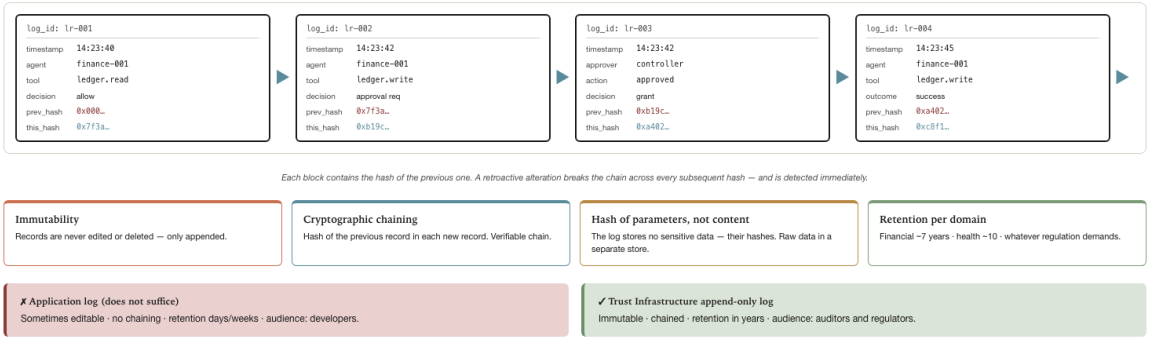


Figure 48: Cryptographic chain of the log · immutability by construction

```
{
  "log_id": "lr-2026-05-02T14:23:45.123Z-7f3a",
  "timestamp": "2026-05-02T14:23:45.123Z",
  "agent_id": "agent-finance-001",
  "trace_id": "tr-2026-05-02-abc123",
  "operation": {
    "type": "tool_invocation",
    "tool": "ledger.write",
    "parameters_hash": "sha256:..."
  },
  "context": {
    "user_id": "user-cfo",
    "session_id": "sess-2026-05-02-xyz",
    "capability_applied": "Finance/Treasury/cashflow-management"
  },
  "policy_evaluation": {
    "policies_applied": ["policy-finance-001", "policy-trade-001"],
    "decision": "allow",
    "approval": {
      "required": true,
      "approver": "user-controller",
      "approved_at": "2026-05-02T14:23:42.000Z"
```

```
    }
  },
  "outcome": {
    "status": "success",
    "result_hash": "sha256:..."
  },
  "previous_log_hash": "sha256:..."
}
```

Each field serves a specific purpose. **log_id** is the record's unique identifier. **timestamp** is the moment of the operation with millisecond precision. **agent_id** identifies the agent. **trace_id** correlates with other events of the same trace (other operations that stem from the same original request). **operation** describes what was done. **context** captures who requested it, which Capability was applied. **policy_evaluation** records which policies were evaluated and with what decision, including human approval when it was required. **outcome** records the result of the operation. **previous_log_hash** is the hash of the preceding record — forming the cryptographic chain.

Properties demanded

The log must satisfy five non-negotiable properties. **Immutability** ensures that records are never edited or deleted — only appended. **Cryptographic chaining** ensures that each record contains the hash of the previous one, forming a verifiable chain; a retroactive alteration would be immediately detectable. **Hashing of parameters and results, not content** ensures that the log does not store sensitive data (parameters, full results), but their hashes — the raw data live in a separate store with specific retention policies. **Configurable minimum retention** ensures that the retention policy is defined per domain: finance typically seven years, health typically ten years, whatever the applicable regulation requires. **Queryable format** ensures that the log is queryable by filters — by agent, by user, by trace, by date range, by operation type.

Difference from application logs

The Trust Infrastructure append-only log is **distinct** from the application log (which records normal-operation events). The differences are substantive and the two systems should not be confused.

Dimension	Application log	Trust append-only log
Purpose	Diagnosis, debug	Auditing, compliance
Mutability	Sometimes editable	Never
Cryptographic chaining	Rare	Mandatory
Retention	Days or weeks	Years (regulatory)
Audience	Developers	Auditors, regulators

A serious organization maintains **both** and integrates them only where appropriate. The application log serves the operations team to diagnose problems; the Trust append-only log serves the organization to defend itself before audits and regulators. Confusing the two — using the application log for auditing, or adding audit weight to the application log — ends up serving both purposes poorly.

Human approval protocol

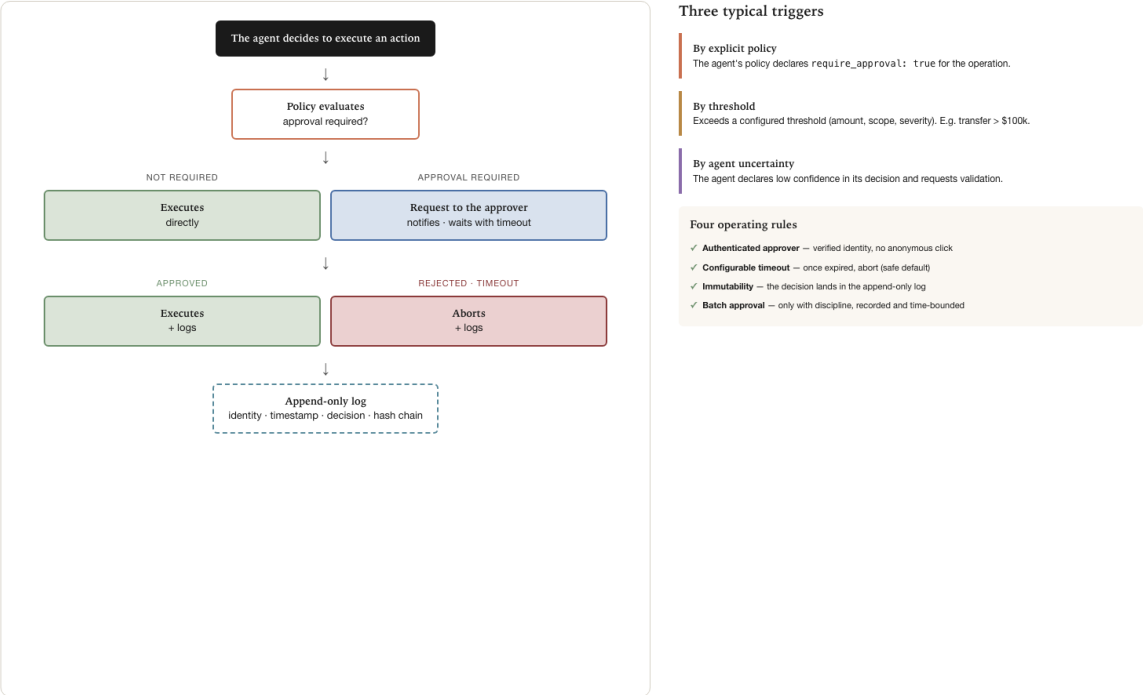


Figure 49: Human approval protocol · canonical flow

When an operation requires human approval before executing, the canonical specification defines the protocol with enough precision for the implementation to be verifiable.

Approval triggers

Three typical triggers activate the human-approval flow. The first is **by explicit policy**: the agent's policy declares `require_approval: true` for the operation. It is the simplest case — the policy is predefined and the system applies it automatically. The second is **by threshold**: the operation exceeds a configured threshold — amount, scope, severity. For example, a payments agent that operates autonomously for small amounts may require approval when the amount exceeds one hundred thousand dollars. The third is **by agent uncertainty**: the agent itself declares low confidence in its decision and requests validation. It is the least common trigger but useful for cases where the agent recognizes it is outside its comfort zone.

Canonical flow

When a trigger activates the need for approval, the canonical flow is as follows. The agent decides to execute an action. The policy evaluates whether it requires approval. If it does not, the agent executes and records. If it does, the system creates an approval request, notifies the configured approver or approvers, and waits for a response with a timeout. If the response is approval, it executes. If the response is rejection or the timeout expires, it aborts. In every case, it records the whole chain in the append-only log.

Approval rules

The approval has four operational rules that the specification demands.

The first rule: **authenticated approver**. The approval is tied to the approver's verified identity, not to an anonymous click. If the approver is Juan Pérez with corporate credentials, the approval is recorded under a verified identity — not as a generic approval by "someone who had access to the system". Without this rule, the approval loses regulatory value.

The second rule: **configurable timeout**. Each operation has a timeout for the approval. If it expires without approval, the operation is aborted — a safe default. In special cases (urgent critical operations), automatic escalation to a secondary approver may be configured, but the escalation itself is recorded and must be permitted by policy.

The third rule: **immutability of the record**. The approver's decision remains in the append-only log with identity, timestamp and, optionally, a comment. Once recorded, it cannot be modified. This protects both the approver (their decision stays as they made it) and the organization (the chain of approvals is traceable).

The fourth rule: **batch approval** for repetitive cases. For repetitive, low-risk operations, pattern approval may be configured — *"pre-approve all operations of type X for the next 4 hours"*. But the pre-approval itself is an operation that is recorded, with the approver's identity, the pattern's scope, and its duration. Batch approval is operationally efficient but requires discipline so as not to become a generic approval that empties the model.

Hallucination-detection rules

Response validation — Pillar 3 of Trust Infrastructure — operates with a set of rules that we describe below. Each one attacks a distinct type of agent error.

The **self-consistency** rule works as follows: the agent formulates the same question in N distinct ways. If the N responses are consistent, confidence increases. If they differ, it alerts. The technique exploits a property of LLM models: when the model is certain, its responses are stable across small variations of the question; when it is hallucinating, the responses are unstable. Self-consistency is relatively cheap and useful for detecting flagrant hallucinations.

The **retrieval-augmented verification** rule works as follows: before asserting a fact, the agent searches for evidence in its corpus of trusted data. If it finds consistent evidence, it asserts with high confidence. If it finds no evidence, it responds with explicit uncertainty or abstains. This technique requires that the organization maintain a corpus of trusted data — curated knowledge bases, not unverified internet data.

The **model-as-judge** rule works as follows: a second model, ideally distinct from the first, evaluates the agent's response and emits a quality score. If the score is below threshold, it alerts or re-asks. The technique is costly (it doubles the inference) but useful for critical cases where the cost of the error exceeds the cost of the validation.

The **constraint validation** rule works as follows: structured responses — JSON, tables, numbers — are validated against an explicit schema. If they do not comply, they are reformulated or escalated. This technique is practically free and should always be active for structured responses — not using it is waste.

The **domain-specific guardrails** rule works as follows: domain-specific rules operate as additional

verification. In finance, validating that figures reconcile in control totals. In health, validating that diagnoses reference recognized clinical guidelines. In legal, validating that citations to laws exist. These rules are domain-specific and demand investment from the domain experts to build them.

The five rules are applied selectively according to context and cost. Not all are activated on every operation — they are activated according to the criticality of the decision. An agent answering casual questions about the weather does not need all five; an agent making investment decisions needs them all and possibly more.

Tokenization policy

Tokenization replaces sensitive data with tokens before they reach the cognition model. It is a critical mechanism when the organization handles sensitive data that cannot be exposed to the external cognition provider, but that the agent needs to be able to reason over in order to deliver value.

Typically tokenized data

Four categories of data typically require tokenization. **PII** (Personally Identifiable Information) — names, addresses, phone numbers, emails — for privacy and compliance reasons. **Financial data** — account numbers, credit cards, balances — for financial regulation. **Health data** — medical identifiers, diagnoses, conditions — for HIPAA and equivalents. **Corporate secrets** — API keys, passwords, classified data — for operational security.

Mechanism

The canonical mechanism operates as follows. The original datum arrives at the tokenization service, which returns an opaque token that replaces the datum. The service maintains an internal mapping between the token and the original datum, in a store with reinforced security. The opaque token is what the cognition receives — the model reasons over the token without knowing what it represents. When the agent makes a decision and emits a result, the result may contain the token; the result passes back through the tokenization service, which de-tokenizes only where required — for example, in the final delivery to the authorized human user or in the invocation of a tool that requires the real datum.

1. The **original datum** arrives at the tokenization service.
2. The service returns an **opaque token** and maintains the token↔datum mapping in a store with reinforced security.
3. The **cognition reasons over the token**, without knowing what it represents.
4. The agent emits its **decision**, which may contain the token.
5. The result returns to the tokenization service, which **de-tokenizes only where required** — the final delivery to the authorized human or the invocation of a tool that needs the real datum.

The model never sees the original datum. De-tokenization occurs only at the point where the real datum is necessary.

Operational rules

Three operational rules demand discipline in implementation. The first: **tokenization before external cognition**. If the agent uses a third-party cognition provider — Claude, GPT, Gemini —, the sensitive data are tokenized **before** leaving the corporate perimeter. Without this rule, the data are exposed to the

provider even if the organization wished to avoid it. The second: **audited de-tokenization**. Each de-tokenization is recorded in the append-only log. It allows reconstructing, afterward, where the original datum was exposed. The third: **mapping in a separate store**. The token-to-original-datum mapping lives in a store with reinforced security — typically an HSM (Hardware Security Module) or a dedicated tokenization service. Keeping the mapping in the same database as the operational data empties the tokenization guarantee.

Minimum viable catalog

An organization operating agents in production must have, as a **minimum viable** baseline, the following components operationalized. Below this minimum, operating agents leaves the organization exposed. Above it, there is progressive refinement according to the domain's maturity and regulatory demands. Note that this catalog **deliberately hardens** some of the spec's SHOULDs (Chapter 5 §4) into an operational floor — hallucination detection among them: what the spec leaves optional for minimal implementations, enterprise operation demands.

Component	Minimum viable
Policy catalog	At least tool, data, schedule, identity, validation policies
CRUDLEX	Applied on all Layer 4 tools, with preconfigured levels
Append-only log	Immutable, chained, retention per domain
Human approval	Configurable per operation, with timeout and record
Hallucination detection	At least self-consistency + domain guardrails, plus constraint validation always active on structured responses
Tokenization	For PII and financial data if applicable
Agent inventory	Who operates, which Capabilities, which tools, who approved
Governance dashboard	Visualization of the inventory + operational metrics

What is critical about this catalog is not the list itself — it is that it is built with the same discipline with which any serious organization builds its financial control system or its security system. Trust Infrastructure is not a byproduct of having agentive AI. It is specific discipline that the organization adopts when it decides to operate agents in production.

The tripartite deployment pattern — Cloud + Client + Local

Operationalizing Trust Infrastructure in an enterprise organization is not a purely technical exercise — it is an exercise in the separation of responsibilities among three planes that operate in physically distinct places. The pattern this operationalization adopts as canonical to solve this problem is the **tripartite deployment**: three coordinated components that live in three places with three differentiated functional roles.

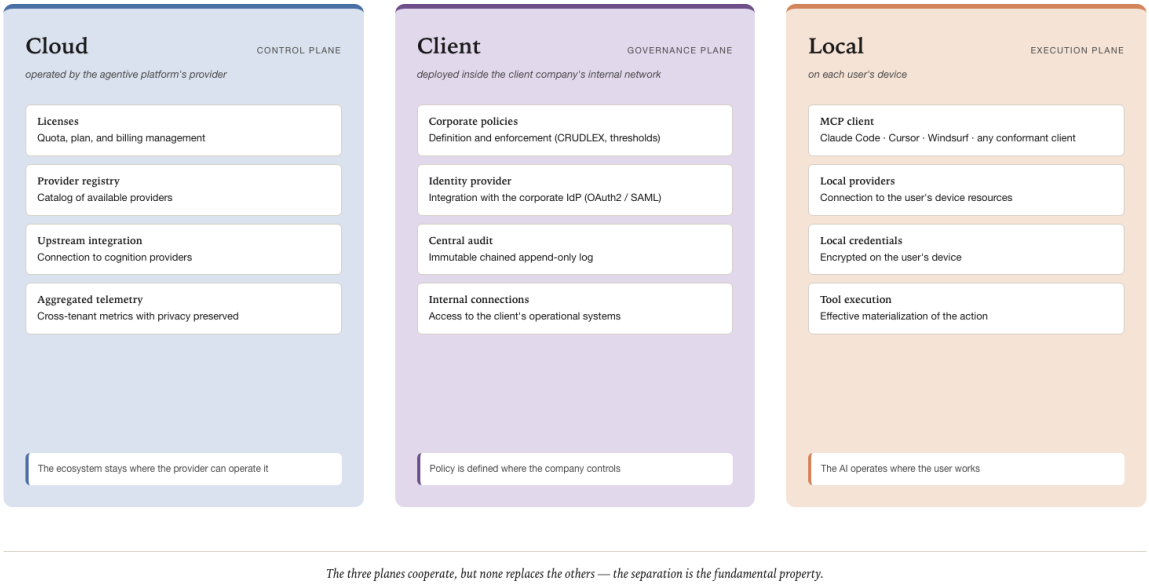


Figure 50: Tripartite pattern · Cloud + Client + Local

The first component is the **Cloud** — the plane operated by the provider of the agentive infrastructure. Its role is **control plane**: license management, registry of available providers, integration with upstream cognition providers, aggregated telemetry with privacy preserved. It is where the provider keeps the platform alive and resolves the problems that require cross-client visibility — incidents that affect multiple tenants, updates to the canonical policy catalog, aggregated health monitoring of the system. The end client does not operate this component; it consumes it.

The second component is the **Client** — the plane deployed inside the client organization's internal network. Its role is **governance plane**: definition and enforcement of corporate policies, integration with the client's identity provider, central auditing of all the agent's actions, connections to the internal systems the agent must access. This is where enterprise governance is exercised — because governance lives where the organization has direct control, not where an external provider promises it. The Client component is what allows the organization to exercise Trust Infrastructure on its own terms.

The third component is the **Local** — the plane deployed on each user's device. Its role is **execution plane**: the app that connects the user's agent (Claude Code, Cursor, Windsurf, any MCP client) with the Client component and with the local providers. This is where the operation actually occurs — where the user's agent invokes tools, where the user's credentials live encrypted locally, where the operation's latency materializes.

The reason this pattern is structural and not arbitrary is that each of the three planes resolves a problem the other two cannot resolve well. The Cloud cannot exercise corporate governance because it does not live inside the company's perimeter. The Client cannot keep the platform alive on its own because each company would replicate the common work. The Local cannot make governance decisions because

it is under the individual user's control, not the organization's. The three planes cooperate, but none substitutes for the others.

> *Policy is defined where the company controls. The AI operates where the user works. The ecosystem is maintained where the provider can operate it.*

This pattern is replicable by any actor operating enterprise agentive infrastructure. The spec demands that the three planes be clearly separated — mixing responsibilities among them produces systems the client cannot govern (when governance lives in the provider's cloud) or that the provider cannot operate (when everything lives in the client's network). The separation is the fundamental property.

The economics of operationalized Trust Infrastructure

Implementing Trust Infrastructure has a cost. It is worth being explicit about the cost because the decision to invest or not invest directly affects the viability of the agentive system in production.

The cost has three components. The **initial construction** is one-time: policy catalog, CRUDLEX model, append-only log with cryptographic chaining, validation mechanisms, integration with tokenization services. It is several months of work for a medium-sized organization. The **operational maintenance** is continuous: adding new policies as the system evolves, monitoring that the existing ones remain valid, adjusting thresholds according to learning, keeping the inventory up to date. It is continuous work for a dedicated team. The **operating overhead** is per operation: latency and compute to validate before acting, record afterward, evaluate policies on every invocation. It is a percentage on top of the normal operating cost — typically five to twenty percent depending on implementation.

The cost is real, but **it is less than the cost of not having Trust Infrastructure**. The Gartner cancellation projection that Chapter 2 documents hangs over this entire chapter — inadequate risk controls are its third cause — and the same chapter records the scale of risky behaviors organizations report from their agents. The cost of a cancelled project or of an incident — a data leak, an incorrect high-impact decision, a regulatory failure — is typically far greater than the cost of preventing it.

> *Trust Infrastructure is not an expense. It is insurance against the asymmetric cost of its absence.*

The asymmetry of the cost is what justifies the investment. Implementing Trust Infrastructure costs a relatively predictable amount. Not implementing it exposes the organization to potentially catastrophic costs whose magnitude it cannot bound in advance. The balance, when calculated with discipline, favors the investment.

Operational continuity — operationalizing the second mechanism

Chapter 5 §4 formalizes the distinction between the two complementary continuity mechanisms — agentic fallback and operational business continuity — and it is assumed here as given. The first is already covered operationally by the spec of the Lets (Ch 5 §2 and §7) and the no-stop guarantee of the layer (Ch 4). The second needs its own operationalization — the section that follows delivers it.

Continuity protocol per physical site

Each operational physical site of an AgencyDomain with distributed Layer 3 must have a **documented continuity protocol**. The spec defines the document's minimum content. Without that content, the

site is not operable under the spec in continuity scenarios — not because the architecture fails, but because the human operation has no guidance when the computational components fall.

The minimum content comprises five elements:

- **Operator role in continuity mode** — who does what when the system falls. The definition must be nominal by position, not by person. *"The cashier switches to continuity mode and records sales in the foliated backup notebook. The store supervisor validates each shift closed in continuity mode."*
- **Physical backup records** — pre-foliated notebooks, numbered receipt books, order tickets, official regulator forms when applicable. Each physical record is the temporary source of truth for the duration of the continuity; when the network returns, its content is entered into the system.
- **Activation thresholds** — at what duration of inactivity the site switches to continuity mode. Canonical recommendation: ten minutes for hospitality and retail, thirty seconds for card charges with online authorization, immediate for the issuance of a regulated receipt. The thresholds are the site's policy, adjustable.
- **Return procedure** — how the physical records are entered retroactively into the system when the network returns. It must make explicit who enters them, in what order, how conflicts are reconciled with events that may have been partially processed before the cut.
- **Drill frequency** — how often the protocol is exercised. Canonical recommendation: quarterly with a scheduled network cut and a minimum duration of fifteen minutes. Without drills, the protocol exists only on paper and fails when it is seriously needed.

AgencyDomain degradation modes

An operation in production does not always operate in normal mode. The spec formalizes the **four canonical degradation modes** according to the failure scenario, and the organization must be able to identify at every moment which mode each site operates in. This identification is what allows operational expectations to be governed.

Mode	Condition	Who sustains the operation?
Normal	All components active: cognition, central, edge, corporate network.	Full parallel topology. The operation chooses the Cognition Path or the Autonomy Path according to the pattern.
Cognition down	Layer 2 unreachable; central and edge active.	**The Autonomy Path** sustains it. Senior Botlets execute; junior and learning Botlets degrade to their last functional version. The cognition rescues failures when it returns.
Edge offline	Edge Botler with no connection to the central; cognition unreachable; physical site isolated.	**Senior Botlets against the local DB + edge-resident Connectors**. The event queue toward the central accumulates; when the network returns, it drains and reconciles.

Mode	Condition	Who sustains the operation?
Total operational continuity	Cognition + edge down due to an exogenous cause (power outage, destroyed hardware, catastrophically lost network).	**The site's manual protocol**. The physical record is the temporary source of truth; retroactive entry into the system is the reconciliation.

The transition between modes is **automatic up to `Edge offline`** — the system detects the failure and degrades on its own. The transition to **Total operational continuity** is **governed by the site's protocol** — a human activates it explicitly when they recognize that no computational component is operating. This difference matters: the first three modes are the architecture's responsibility; the fourth is the client's responsibility, executed by its operators.

Traceability of the transition to continuity mode

The append-only log must record the transitions between modes so that subsequent auditing can reconstruct what sustained the operation at each moment. The spec defines the canonical marks:

- **Start of operational continuity** — when the site activates the manual protocol, it emits (when the network returns, retroactively) a `mode-change: continuity-operational` event with the timestamp of the cut and the estimated duration.
- **Physical records entered retroactively** — each transaction entered from a physical record carries the tag `provenance: manual-continuity` and `original-timestamp: <physical time>` distinct from `system-timestamp: <time of entry>`. The distinction lets reports and reconciliations distinguish physical events from digital events.
- **Edge queue reconciliation** — when an edge Botler drains its queue toward the central after the network returns, the events carry the tag `provenance: edge-queue-replay` with the site's original timestamp.
- **Auditable distinction between agentic fallback and operational continuity** — the log distinguishes `agentic-fallback` events (the cognition rescued the Let — executing on behalf of the Botlet or receiving the Agentlet's escalation) from `operational-continuity` events (a human sustained the operation). Subsequent auditing can separate the two cases without ambiguity. This distinction is what the organization presents when a regulator asks how it operated during an incident.

Operational properties demanded

Property	Level
Explicit distinction in product documentation	**MUST**
Documented continuity protocol per physical site	**MUST**
Documented drills at a minimum quarterly frequency	**SHOULD**
Four degradation modes recognizable by the organization	**MUST**
Traceability of the transition to continuity mode in the append-only log	**MUST**

Property	Level
Auditable distinction between agentic fallback and operational continuity	**MUST**
Retroactive reconciliation of physical records into the system	**MUST**

The complete operationalization of the second mechanism closes the loop opened in Ch 5 §4. The distinction between agentic fallback and operational continuity is no longer merely conceptual — it has a field protocol, canonical operation modes, and auditable traceability.

What comes next

With the operational block closed, the book closes with the reference implementation — Chapter 9 — and its epilogue. Anyone who has read the preceding chapters with the sense that each one leaves open questions will find there the explicit confirmation that this reading is correct: what follows is not a closure but the recognition that this specification is a formalized point of departure, not a final destination.

Chapter 9 · Vergis — the reference implementation

The preceding chapters delivered a specification: primitives, required properties, canonical separations, declarative contracts. A specification, however precise, is not a living system. Between the contract and the practice there is a leap that every external reader intuits upon closing the last chapter of the spec: "*what does this look like when it works?*". This is the section that answers that question with a concrete, downloadable, executable artifact.

A note on the scope of the canon

It is worth fixing first what this canon is and is not, because the answer delimits the role of everything that follows.

> This canon contains the **structure and the vocabulary** of the Agentive World: definitions, primitives, required properties, canonical separations. **It does not contain methods to implement or operational catalogs** — those live in complementary bodies. The public reference implementation is **AgencyDomains.org**, materialized in **Vergis**: designed so that any developer or student can download it, read it, run it, and learn how the canon translates into living systems. Other implementers — commercial products, proprietary codices — offer their own complementary bodies over the same canonical structure.

The separation is deliberate. The canon describes *what* properties a conformant AgencyDomain satisfies; it does not prescribe *how* a use case is discovered, *how* a proto-Botlet is forged, nor *which* concrete proto-Botlets a catalog must contain. Those are method and catalog — matter for the complementary bodies. Without this note, the reader would wonder why the spec does not include a chapter on Discovery or a repertoire of ready-made proto-Botlets. It does not include them because they do not belong to it: they belong to the level of the implementation, and the reference implementation is where they become visible.

This note bounds the *canon vs reference implementation* axis — what belongs to the spec and what to whoever implements it. It is distinct from the declaration of **what the book deliberately omits** (detailed verticals, operative code, ROI analysis by stack), which lives in the Epilogue, section "What is NOT in this book."

What is it and why does it exist?

Vergis is the public reference implementation of AgencyDomains — the AgencyDomain made operational. It is the concrete bridge between the canon and the practice: a system that anyone can download, read, run, and contribute to, built to demonstrate that the spec's primitives are not theory but pieces that compose a runtime that works.

A specification without a reference implementation runs the risk of remaining a formal exercise without traction. The reference implementation fulfills four functions that the canon's text alone cannot. It gives the external reader a **concrete point of entry** after reading the spec. It proves that the canon is **implementable** — the implementation is the living proof. It enables a **clear adoption model**: download, operate, extend, contribute. And it anchors the abstract vocabulary in components that can be inspected line by line.

Vergis is neither a tutorial nor a pedagogical toy. It is the substrate on which productive systems run. That quality — developed further below — is what distinguishes a serious reference implementation from a collection of examples.

Where does it live?

Vergis lives in a public repository linked from **AgencyDomains.org**. The code is distributed under **`AGPL`** (Affero General Public License); the documentation, under **`GFDL`** (GNU Free Documentation License).

The choice of licenses is structural, not incidental. The `AGPL` guarantees that improvements to the runtime — including those that operate as a network service — remain available to the community: whoever deploys a modified version of Vergis and offers it over the network MUST publish its source. The `GFDL` keeps the documentation free and derivable. Together, they sustain the promise of a common base that no actor can close: the reference platform remains open even though the catalogs built upon it are private.

How are the pieces named? — Vergis · Botler · Mira · Daftar

Four labels of **three distinct natures** intervene in the reference implementation. Confusing their nature — in particular, confusing a type with a proper name — obscures the architecture. The scheme:

Layer	Type · category / canonical primitive	Proper name
Platform · *Meta-Cognitive Platform*	reference implementation of AgencyDomains (the AgencyDomain made operational)	**Vergis**
Layer 3 runtime	**Botler** (normed construct of the Botlet spec)	*— (generic; "the Botler". The Vergis build gives it no proper name.)*
Catalog component	platform proto-Botlet for informational operation	**Mira**

Layer	Type · category / canonical primitive	Proper name
Catalog component	platform proto-Botlet for **capture** (assessor of instruments) — the second in the catalog, built in Vergis 0.27.0	**Daftar**

The type / proper-name distinction:

- **Botler** is a **type** — a canonical construct of the Botlet spec (not one of the eight primitives, but normed vocabulary of the canon). Any conformant Layer 3 runtime *is a* Botler. It is not a proper name; the Botler that Vergis packages is "the Botler" generic, with no instance name.
- **Vergis**, **Mira**, and **Daftar** are **proper names** of specific instances — they live in the same drawer as Soveria, Agentia, or ultraPRO. Vergis names *this* platform; Mira names *this* catalog proto-Botlet; Daftar, the second one.

By category, Vergis is a **Meta-Cognitive Platform**: it does not perform object-level cognition — that is Layer 2 — but rather **administers the economics of cognition**. It decides when the agent runs on pre-forged muscle (G1) and when it invokes fresh cognition (fallback), manages the `95/4/1` cycle, junior→senior maturation, and the crystallization of experience into reusable structure. That is metacognition in the precise sense: monitoring and control of cognitive processes.

> The descriptor **Meta-Cognitive Platform MUST NOT** be abbreviated to `MCP`. In the current agentive space, `MCP` names the **Model Context Protocol**; the acronym is taken. The descriptor is used spelled out.

The name **Vergis** comes from *Caprica* (the *Battlestar Galactica* universe): Tomas Vergis was the legitimate inventor of the Meta-Cognitive Processor, the piece that gave machines cognitive independence. The name reclaims metacognition for its legitimate source. And it carries, out of the box, the design principle that governs the platform: in that story, the metacognitive substrate **never worked until it fused with a living consciousness**. The substrate is **inert until something animates it** — the "breath of life" principle: Vergis comes alive only when Mira, Daftar, and the agents animate it. The platform, by itself, is muscle at rest; cognition and the Botlets are what actualize it.

What does it include?

Vergis packages an initial set of components. Each one materializes a canonical primitive, and is named here by citing it:

- **The Botler's abstract contract** — the **Layer 3** runtime primitive (§5). The Vergis Botler is generic by definition: it manages the lifecycle, isolation, and execution of *any* Botlet without understanding its domain. It exposes the control points — `capability_call`, `log` — into which the specialists plug, and it enforces each spec's validation by orchestrating the validation point the Botlet type provides, without executing it with domain knowledge.
- **Mira** — a **platform proto-Botlet** (§5) for informational operation. Its code is generic — the shared engine —; its specialization lives in a compositional configuration that the agent fills in at Engineering time. Each Information Product is its own Botlet, a specialized instance of the common engine. Mira operates in **G1**: the agent configures, it does not write the engine's body.
- **Daftar** — the **second platform proto-Botlet** in the catalog, of the **capture** family (§5) and its reference proto-Botlet: a per-instance assessor Let that serves the instrument catalog of whichever

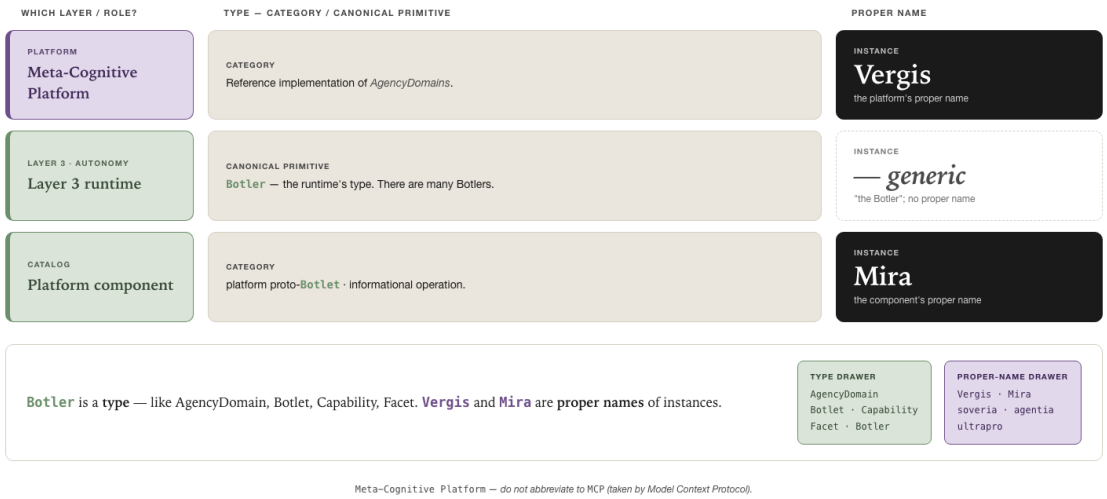


Figure 51: Vergis · Botler · Mira — type vs proper name

student comes in, applies the instrument, records every attempt, and grades it. Built in version 0.27.0 of Vergis, it is the executable proof that the Botler has no subtypes by family: Mira and Daftar enter through the same runtime interface. The instruments it applies are catalog content, immutable once published — not Botlets, as the three tests of §5 establish.

- **A starter set of Capabilities** — a minimal repertoire of canonical **Capabilities** (§5, in the strict sense: cognitive know-how of Layer 2) and the **Connectors** (Layer 4) that feed them, sufficient to compose the examples.
- **Trust Infrastructure templates** — skeletons of the cross-cutting axis (§5): policies, append-only log, declarative quality contract (Freshness · SLA · Degradation policy · Audience · Refresh policy) that any conformant Botlet may declare.
- **Executable examples** — complete, anonymized cases that walk the derivation chain `use case` → `Botlets` → `proto-Botlets` and show the pieces operating together.

Why is it production grade?

The reference implementation is neither didactic code nor a prototype. It is the **same runtime** that operates Grupo Ultra's commercial products — **Agentia**, **Soveria**, **ultraPRO** — in productive operation.

The difference between public Vergis and those products **is not the quality of the code**, but the **catalog**. The products consume the public reference implementation **plus a proprietary codex** — in Grupo Ultra's case, **ucodex** — that curates proto-Botlets, Capabilities, and patterns refined by real cases. The runtime is the same; what changes is the pre-forged repertoire each one brings and the method with which it forges and maintains it.

This matters for the reader evaluating whether to adopt the canon. Downloading Vergis is not downloading a demo that will have to be rewritten for production: it is downloading the base on which productive systems already run. The path of an implementer is not "learn with the reference and then buy the serious thing," but "operate the reference and curate the catalog the case itself demands."

How is it adopted? — the model

The adoption model is replicable by any implementer, without permission or contract with a central actor:

1. **Consume** the public reference implementation (Vergis, `AGPL`).
2. **Curate** its own codex — its private catalog of proto-Botlets, Capabilities, and patterns, refined by its cases.
3. **Offer** its own products on that base.

AgencyDomains.org **is not the property of one actor**: it is common ground. Grupo Ultra is one more adopter — with ucodex as its codex —, not the owner of the reference platform. Any other implementer can travel the same path with a different codex. The spec aims to be an industry standard, not the intellectual property of a vendor; the reference implementation inherits that aim.

How does the catalog grow? — common catalog and network effects

Proto-Botlets accumulate in **catalogs shared** by communities of implementers. Each implementer who consumes a proto-Botlet contributes to its maturation: new variants, tested configurations, refinements. The more implementers consume a proto-Botlet, the faster it matures, the more cases it covers, and the more reliable it becomes. Implementer n+1 receives versions refined by implementers 1 through n — a network effect over operational capability.

A catalog is not necessarily public. Membership in a catalog community admits four **modes**:

Mode	What is it?
Private contract	Closed catalog between a client and its provider.
Proprietary codex	An implementer maintains its curated private catalog (e.g. ucodex).
Open public catalog	AgencyDomains.org as exemplar: any implementer consumes and contributes.
Sovereign agreement	AgencyDomains that adopt common standards without a direct commercial contract.

The four modes coexist over the same canonical structure. A proto-Botlet may be born in a proprietary codex, mature against real cases, and be promoted to the public catalog when its nature is structural and not methodological; or remain private if it encapsulates competitive advantage. The structure is common; the curation is each one's own.

How does one contribute?

Contribution to Vergis is governed by a public proposal process. At a high level:

- A contribution **SHOULD** arrive as a traceable proposal — issue or request — before it arrives as finished code, so that the design discussion precedes the implementation.
- What is accepted is what is **structural and reusable**: proto-Botlets of general value, canonical Capabilities, Connectors, refinements to the Botler's contract, improvements to the declarative quality contract.
- What does **not** go into the public reference is the **method** and the **proprietary catalog**: how a particular organization does Discovery, curates its codex, or forges its niche proto-Botlets. That lives in each implementer's complementary bodies.
- The admission criterion is the same one that separates canon from implementation: *is it necessary to describe or execute any conformant implementation, or is it one of N possible ways of doing it?* The former belongs to the reference; the latter, to a codex.

Every accepted contribution falls under `AGPL` (code) and `GFDL` (docs), inheriting the openness of the base.

Do the reference and the proprietary codices coexist?

Yes, and without tension. The relationship between the public reference implementation and the proprietary codices is **harmonious by design**, not a precarious equilibrium.

The runtime is common; the catalog is one's own. A proprietary codex does not compete with Vergis: it **consumes** and **extends** it. Improvements to the runtime flow toward the public base through the mechanics of the `AGPL`; improvements to the catalog remain wherever their owner decides, according to the membership mode it chooses. An implementer does not face the dilemma "open or closed": it operates open at the base and chooses, proto-Botlet by proto-Botlet, what it curates in public and what it retains in its codex.

This coexistence is what makes the ecosystem sustainable. The common base keeps each implementer from reinventing the runtime; the proprietary catalog preserves the space where each one builds its advantage. Without the common base there would be no interoperability; without proprietary catalogs there would be no incentive to invest in deep curation. The reference implementation sustains both.

Conformance

A system that intends to present itself as a **conformant reference implementation** of AgencyDomains — not merely as a conformant AgencyDomain (§5) — satisfies, in addition to the spec's `MUST`s, the following requirements. IETF convention: `MUST` mandatory, `SHOULD` strongly recommended, `MAY` optional.

Requirement	Level
Publicly available, downloadable, and executable	**MUST**
Code under a free license with network copyleft (`AGPL`)	**MUST**
Documentation under a free license (`GFDL`)	**MUST**
Generic Botler, with no specialization by domain	**MUST**
At least one platform proto-Botlet from the catalog, operating in G1	**MUST**

Requirement	Level
Traceable derivation chain in its examples (`use case → Botlets → proto-Botlets`)	**MUST**
Declarative quality contract in the example Botlets	**SHOULD**
Public, governed contribution process	**SHOULD**
Starter set of Capabilities and Connectors	**SHOULD**
Open public catalog with active network effects	**MAY**

A reference implementation that meets all the `MUST`s is the living proof that the canon is executable. Vergis is such an implementation.

Evolution frontier

Three areas of the reference implementation are in active evolution, in correspondence with the frontiers of the spec itself.

The **operational generation** is the first. Vergis operates today in **G1**: the agent configures pre-forged proto-Botlets from the catalog. The incremental migration toward G2 and the asymptotic horizon G3 do not demand re-architecture; they demand that the state of the art of cognition advance. The reference will evolve its Engineering scope without rewriting its structure. The definition of the G1/G2/G3 generations lives in Chapter 5 §2 (and its underlying essay, in the Epilogue); here Vergis is only situated within them.

The **federation of catalogs** is the second. The membership modes are described; the concrete protocol by which sovereign catalogs discover, version, and reconcile proto-Botlets among themselves is open work, in correspondence with the federation frontier between AgencyDomains.

The **breadth of the public catalog** is the third. The starter set covers what is needed for the examples; the public catalog will grow through community contribution, and its maturation will follow the network-effects dynamic described above. The speed of that growth depends on the community of implementers, not on a central plan.

This page intentionally left blank.

Epilogue · The Evolution Frontier

Every chapter of this book left horizons open. The architecture admits non-LLM cognition, but the contemporary implementation is LLM-centric. AgencyDomains federate, but the formal protocol has not yet been agreed upon. Trust Infrastructure has its five pillars, but its cryptographically verifiable external audit is work in progress. The eleventh link of the value chain is barely explored. Capabilities admit a marketplace, but the protocol is pending.

These horizons are not omissions of the book. They are **frontiers of the field**. Any book that claimed to be definitive in a discipline barely three years into its formal existence would be dishonest: that claim would be false, and readers would detect it. This book does the opposite: it makes the frontiers explicit, names them, and leaves them as an invitation to the technical community to work on them. What the book delivers is **version 1.0** of the category — a starting point formalized with enough discipline that the industry can build on it. It is not a final destination.

This epilogue closes the book by returning to the live frontiers, naming the open work the community must take up, declaring what the book deliberately does not address, and arguing why the book's editorial bet — a shared category rather than closed intellectual property — serves the field, and its coiner, better.

What the book established

Over nine chapters this book established a coherent set of formal constructs that hold each other up and that, taken together, constitute a proposed **emerging standard** for the agentive category:

- **A paradigm** — the Nadella Line and its operative dividing question: *does the human open applications to do their work?*
- **A formal architecture** — four layers (Interaction, Cognition, Autonomy, Access), cross-cutting Trust Infrastructure, the Agent First principle; product-agnostic, in the manner of JavaSpaces or the OSI model.
- **Eight canonical technical primitives** — **AgencyDomain**, **Botlet**, **proto-Botlet**, **Agentlet**, **Capability**, **Trust Infrastructure**, the **Assistant vs Autonomous Agent** distinction, and the **Facet**; reusable across implementations.
- **A market model** — eleven links by four depths, with four strategic archetypes, to map any actor in the industry.
- **A canonical application** — real-time knowledge as the foundational case, replicable in any organization with a mature data warehouse.
- **An operationalization** — Trust Infrastructure translated into policies, CRUDLEX, a chained append-only log, human approval, hallucination detection, tokenization: what separates the spec from the buildable guide.

The book is **version 1.0**. Its claim is not to be exhaustive — it is to be **formal enough to be buildable**. Multiple areas remain open to evolution, and this epilogue names them explicitly.

The four live frontiers

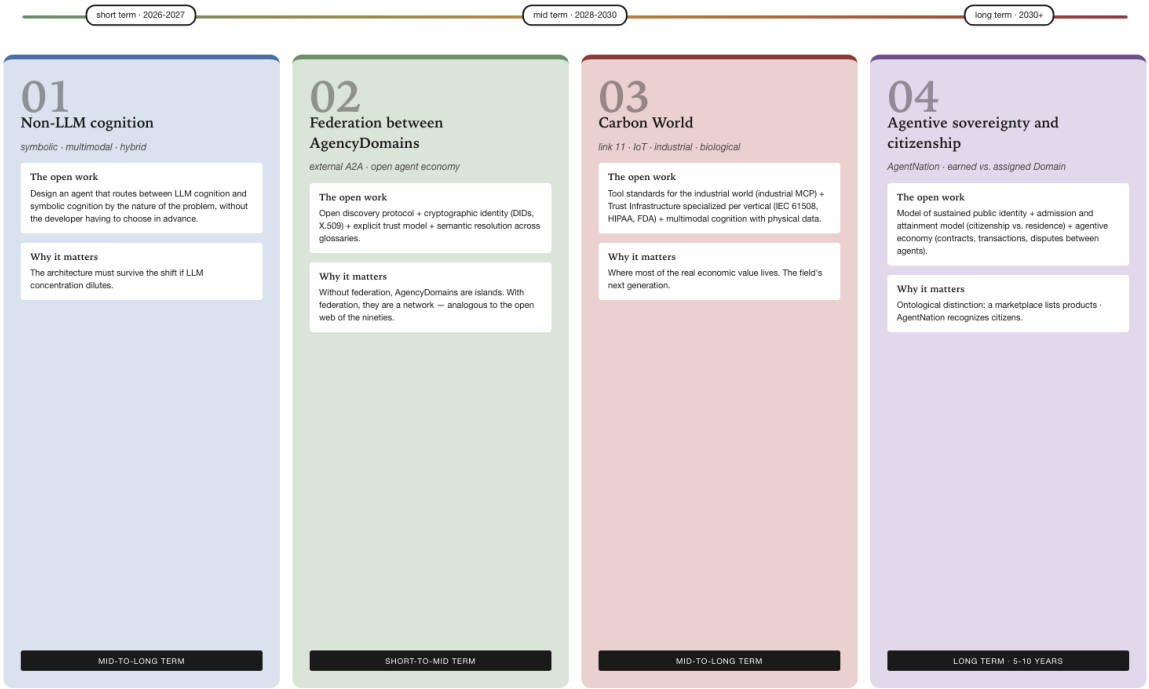


Figure 52: The four live frontiers · open work of the field

Four areas where the architecture admits extension that has not yet set as a normative spec deserve to be made explicit as the book closes. The first three are **technical** — non-LLM cognition, federation between AgencyDomains, the Carbon World; the fourth is **institutional** — agentive sovereignty and citizenship. They are structural frontiers, not pending details — they are where the next generation of the field will be defined.

Non-LLM cognition

Layer 2 — Cognition — admits, by specification, symbolic, multimodal, non-LLM cognition. The contemporary implementation is **predominantly LLM-centric**: the Core-in-Model actors (OpenAI, Anthropic, Google, Meta, DeepSeek) build LLMs, and the actors covering the upper layers build on LLMs. The book's architecture does not oppose that concentration — but neither does it endorse it as permanent.

The technical frontier is to **integrate symbolic cognition** — rule systems, planners, solvers — with LLM cognition in a single coherent agent. There are precedents from the field of classical symbolic AI: expert systems such as MYCIN, planners such as STRIPS, solvers such as Prolog. And there are precedents from contemporary hybrid AI: programs that combine LLMs with SAT solvers or constraint

solvers for formal problems.

The open work is how to design an agent that **routes between LLM cognition and symbolic cognition** according to the nature of the problem, without the developer having to choose ahead of time. An agent facing an open question about natural language should use an LLM; the same agent facing a planning problem with formal constraints should use a symbolic planner; a third case might demand a composition of both. Cognition-agnostic design is the strategic horizon of the coming decade, and the book's architecture is designed to survive the shift.

Federation between AgencyDomains

Intra-AgencyDomain coordination — between runtimes and Botlers that live in the same AgencyDomain, via the `A2A` protocol — is mature as a concept and exists in contemporary implementations. **`A2A` between unrelated AgencyDomains** — federation — is open work. The industry is converging toward certain directions, but full consensus has not arrived.

Federation requires solving four things. An **open discovery protocol** — how an AgencyDomain publishes the agents it offers for external invocation, in a format queryable by another AgencyDomain. **Cryptographic identity of agents** — DIDs (Decentralized Identifiers), verifiable credentials, mechanisms without a central authority that adversarial actors could control. **An explicit trust model** — which AgencyDomain trusts which others, and for what, with trust gradable by case. **Semantic resolution between glossaries** — two AgencyDomains that may have different glossaries negotiate the meaning of tools and capabilities when they interoperate.

Federation is the ingredient that would enable an **open economy of agents** — analogous to the open web of the nineties or to email federation. Without it, AgencyDomains are islands; with it, they are a network. The emergence of a federated network of AgencyDomains would be the most important change for the agentive field after the consolidation of the architecture — and this book is designed to be the substrate on which that network is built once the industry converges on its protocols.

Carbon World

The contemporary Layer 4 — Access — connects to the digital world. Chapter 6 §3 established that the next frontier is connecting to the physical world — IoT, industrial systems, machines, manufacturing processes, biological data. The Carbon World is where most economic value lives, and where the agentive architecture must extend to reach full economic relevance.

The open work includes three fronts. **Tool standards for the industrial world** — the equivalent of MCP for sensors, actuators, SCADA/MES/PLC systems. **Trust Infrastructure specialized by vertical** — extensions that encode the regulatory requirements of functional safety (IEC 61508, ISO 26262), health (HIPAA, FDA), aviation (DO-178C). **Model learning in the Carbon World** — multimodal models that integrate sensor data, physical simulations, industrial video, biomedical data as native modalities.

The crossing into the Carbon World is where most economic value lives. Building agentive infrastructure specialized for this crossing is **years of work**, and it begins now.

Frontier 4 — the institutional horizon: agentive sovereignty and citizenship

The fourth frontier is the least developed and the most speculative of the four, but the book names it explicitly because the first signals are visible and the category deserves to be recorded. It is the **institu-**

tional frontier — the horizon where AgencyDomains cease to be merely technical constructs operated by organizations and begin to constitute themselves as **public realms** where agents exist with identity, persistence, and public addressability sustained over time.

The critical distinction of this frontier is **ontological**, not technical. An AgencyDomain in a private regime contains agents that the organization **assigned** — they are residents of the space because the organization placed them there. An AgencyDomain in a public regime may contain agents that **earned** their place — they are citizens of the space because they met the requirements to be so. The difference between residence (assigned) and citizenship (earned) is the difference between **catalog** and **nation**: a marketplace lists products; a nation recognizes citizens. And the earned — not assigned — Domain already has an emergent name: **Dominion**. The current AgencyDomains spec does not require the citizenship model — it admits both — but the institutional frontier is precisely to develop the formal constructs that the citizenship model requires.

The industry is converging on an operative term to name this horizon: **AgentNation**. An AgentNation is an AgencyDomain in a public regime that explicitly adopts the model of agentive citizenship. It has admission rules, stable public identity for the agents that compose it, sovereignty mechanisms over the agent's territory (its Domain), and an internal economy that recognizes agents as first-class economic agents. It is not a marketplace; it is a jurisdiction.

The open architectural work this frontier poses covers three axes. The first is the **model of sustained public identity** — how to build an agentive identifier that survives a change of provider, migration between infrastructures, the replacement of the cognition model. The equivalent of the human passport for agents. The second is the **model of admission and earning** — what requirements an agent must satisfy to be admitted as a citizen (not as a listed product), and how those requirements are verified auditably. The third is the **model of agentive economy** — how citizen-agents contract, transact, charge, and pay each other, with what unit of value, under what dispute rules.

The first implementations of AgentNation are emerging in 2026, still as institutional prototypes. Soveria is one of the projects that position themselves explicitly on this frontier, operating as an AgencyDomain in a public regime with an agentive-citizenship vocation. Consolidating the model is probably five to ten years of work, and will require both technical advance and regulatory construction. But the category exists, the industry is beginning to name it, and an agentive architecture that aspires to serve the long term of the field must contemplate the horizon.

Crystallization — why the advanced agent generates less

The frontiers above belong to the field. This closing piece belongs to cognition itself: the trajectory an agent advances along as the state of the art matures. Chapter 5 · §2 — Botlets fixed the **Botlet generations** (`G1/G2/G3`), the proto-Botlet as the pre-forged piece the agent configures, and the `95/4/1` cycle as the regime of its operation. What remains for the epilogue is the underlying argument: why the direction of that advance is not the one intuition predicts.

Where does an agent advance toward?

Consider the question before reading on: if technology allowed an agent to generate, on the spot, all the code each task needs — no pre-forged patterns, no catalog, nothing prepared in advance —, would that be the most advanced agent possible?

Intuition says yes. More generation, less scaffolding, more power exercised live. It looks like the summit.

Hold that intuition for a moment. Then consider a person expert in their craft. A surgeon does not re-derive the suturing technique for each patient; a concert pianist does not solve each measure's fingering on stage; a pilot does not compute from first principles how to level the wings. What distinguishes the expert from the novice is not that they improvise more — it is that they improvise **less**, because they have **crystallized** into reflex what once demanded thought. The novice derives everything every time; the expert has muscle memory.

Now return to the agent. The one that regenerates every artifact from scratch on every execution is not the expert of the example: it is the novice, condemned to re-think the same move every time it appears. The advanced agent does the opposite of what intuition predicted: **it generates less, because it has crystallized more**.

What is lost by not crystallizing?

The cost of "generating every time" is not the compute — that is cheap. The cost is throwing away the properties that turn a Botlet into reliable infrastructure, and which only exist when what runs is stable structure and not fresh code:

- **Reproducibility** — the same configuration with the same data produces the same artifact; regenerated code does not guarantee it.
- **Validation before execution** — a configuration is validated against a schema before running; arbitrary code generated on the fly is not audited the same way before acting.
- **Portability** — a declarative configuration migrates between conformant runtimes; bespoke code stays tied to whatever it was written against.
- **Audit and prior governance** — a declarative specification is reviewed before it executes; generated code is a box that must be re-audited on every run.
- **Trust regime** — in a regulated domain nobody signs off on an artifact whose body is generated fresh on each execution. Pre-forged stability is a condition of trust, not a luxury.

To crystallize is not to renounce power. It is to convert power into trust.

Where, then, does the capacity to generate live?

It does not disappear — it relocates. The agent's life cycle reserves generation for its margins, not for its center. In the 95% of stable operation, the Botlet runs as a pre-forged, configured structure. In the 4% of detected change and the 1% of regeneration — and on the fallback path when something fails — the agent deploys its full authoring capacity: it forges a new piece for the catalog, redesigns, recomposes. Generation is the tool of the edge, not of the permanent regime.

So the answer to the opening question inverts cleanly. The most advanced agent is not the one that generates the most; it is the one that has crystallized so much that it almost never needs to — and that reserves its generative capacity for the genuinely new, which, with a mature catalog, is less and less. It is exactly what the generations of Chapter 5 §2 formalize: `G3` capacity is best spent producing `G1` reuse.

And the black-box expectation?

A thesis that does not face its strongest objection is not validated — it is untested. The agentive architecture feeds a legitimate expectation: that the user converses with the agent and nothing else; that every surface — every interface, every artifact, every view — is born from the agent in the moment, molded to the exact need of that instant. Under that expectation, pre-forging looks like an anachronism: if the

agent can generate whatever interface is needed whenever it is needed, what is a catalog for? The state of the art would seem to push, precisely, toward the agent that generates *every time*.

It is worth facing the objection with the most demanding example available: the human brain, the most sophisticated machine we know. If sophistication consisted in cognition doing everything live, the conscious brain would compute every muscle fiber when walking, re-derive edge detection at every glance, solve from scratch the articulation of every syllable when speaking. It does not; it could not. Instead, the brain does not act on the world directly: between conscious cognition and the exterior there are layers that operate fast, reliably, and without supervision. The **cerebellum** tunes the fine timing of movement; the **basal ganglia** select and automate action sequences — habits, pre-forged programs that run without deliberation. The cortex does not micromanage that work: it directs it.

The parallel is structural. The cortex's deliberation is **Cognition** (Layer 2), which interprets, decides, and composes; the cerebellum and the basal ganglia — where the pre-forged programs live and execute without deliberation — are **Autonomy** (Layer 3). That the user *interacts* only with the agent does not imply that the agent *executes* everything with its cognition: the conversation is the surface; underneath, agentive intelligence delegates to layers. When cognition composes a new interface, that is its genuine act of generation; but the piece, once composed, **runs** in Layer 3 as a pre-forged, configured operation, not as code regenerated every frame. And if that composition repeats, it crystallizes: it stops being a cognitive act and becomes a catalog piece.

Thus the objection does not topple the thesis: it confirms it from the hardest angle. The most sophisticated machine we know is not the one whose cognitive layer does everything; it is the one that stratified itself so that cognition *does not have* to do everything. Stratification is not a patch on insufficient cognition — it is the shape sophistication takes.

What the technical community must build

Beyond the four frontiers, there is **specific work** the community can take up to extend this book. Each could be, in itself, a companion volume. There are five.

The first is the **Capability marketplace**. Chapter 5 §3 proposes that Capabilities admit a market economy analogous to the open-source package economy. What is missing is a normative Capability-package protocol, a cryptographic signature model that allows provenance to be verified, an economic model (free, premium, revenue share), a security policy for review, sandboxing, and removal of malicious Capabilities. It is spec work the community can undertake collectively.

The second is the **cryptographically auditable audit**. Trust Infrastructure's append-only log provides immutability and chaining. What is missing is **third-party verifiability without access to the system** — an external auditor who can verify the log's integrity and regulatory conformity without the system being opened to them. Candidate technologies: zero-knowledge proofs over the log, anchoring in a public blockchain, confidential computing in TEEs (Trusted Execution Environments). The field is active; integration with Trust Infrastructure is not yet formalized.

The third is **trust scoring of agents**. A composite metric that evolves with the agent's behavior, analogous to the credit score of humans. It would let the organization adopt agents with trust modulated by their trust score: agents with a high score receive more autonomy; agents with a low score require more supervision. Operationalizing it demands metrics that are verifiable and resistant to manipulation — a problem with precedents in other domains but no general solution yet in the agentive context.

The fourth is the **instrumentation standard for Observability**. OpenTelemetry is the industry standard for observability of classical distributed systems. For agentive systems there are specific extensions that are not yet canonical: how to trace cognition invocations, how to measure the semantic quality of responses, how to correlate human conversations with Botlet execution in the background. Consolidating that extension as a standard is open work for the community.

The fifth is the **tenancy model for Trust Infrastructure**. When an AgencyDomain operates in multi-tenant mode — multiple client organizations share the underlying infrastructure — the Trust Infrastructure must guarantee strict isolation between tenants. What is missing is a formal model of log isolation, cryptographic guarantees of data isolation in shared cognition, compliance patterns when tenants have different regulatory requirements. It is critical work for the SaaS model to be able to serve regulated sectors.

What is NOT in this book — and why?

For editorial honesty, we declare what the book deliberately **does not address**. The declaration matters because it calibrates the reader's expectations and prevents the search for content that is not here.

The book **does not describe specific provider implementations**. It is not a manual for Claude, nor for OpenAI, nor for Anthropic, nor for Google, nor for any particular product. The architecture is agnostic. Providers are mentioned only when they are the source of relevant citations — Nadella on BG2, Anthropic with MCP, AtScale with its semantic-layer measurements.

The book **does not deliver operational code**. There are no executable snippets, no step-by-step tutorials, no specific configurations. The book is **specification**; the operational manuals live apart. An organization that adopts the book's architecture will write its own implementation manuals — or adopt the manuals of the provider whose implementation it chooses.

The book **does not develop detailed vertical cases**. Beyond the canonical application of Chapter 7, no vertical cases are developed — how to build a legal, medical, or financial agent. Those cases can be books in their own right — future extensions of the corpus this book begins.

The book **does not do an economic analysis of each stack decision**. Although market data and typical costs are cited, no specific ROI analyses are done. Those analyses depend on the particular case and do not admit useful generalization in a general book.

The book **does not describe ultraBASE's portfolio**. The separation is deliberate: if the book mixed the formal architecture with its particular implementation, it would lose its claim to be a standard and would become the manual of a product.

Why does this book aspire to be a standard?

Three reasons justify the book's editorial bet as a shared category — not as the intellectual property of one actor.

Technical categories are established by writing them down

Object-Oriented Programming was not established when someone invented it. It was established when Grady Booch, Ivar Jacobson, and James Rumbaugh **wrote it down with enough discipline** for the industry to adopt it. Domain-Driven Design was not established with the first practitioner — it was

established when Eric Evans published *Domain-Driven Design: Tackling Complexity in the Heart of Software* (2003). Design Patterns were not established with Christopher Alexander applied to software — they were established when the "Gang of Four" published *Design Patterns* (1994).

The Agentive Architecture will have its own history. This book aims to be **a starting point** that the community can critique, extend, improve, or replace. It will not be the last word. But its existence makes it possible for the conversation to have a common vocabulary. The claim is not modest — but it is defensible.

Fragmentation costs the industry

Today's AI industry suffers from vocabulary fragmentation: different actors call the same things by different names, and equal things by different names. *Agent, agentic, autonomous agent, AI assistant, copilot* — the semantic stack varies between providers. That fragmentation has a **real operational cost**: enterprise buyers cannot compare products, regulators cannot formulate common requirements, developers cannot interoperate.

A shared architecture — adopted by convention rather than by imposition — reduces that cost. And convention requires that **someone write it first**. The reason industrial standards historically emerge is that some actor invests the work of formalizing them before the industry needs them, and then the industry adopts them because it finds the work already done. This book attempts to be that work for the agentive category.

The open standard reinforces the first adopter

There is a mistaken intuition that whoever wants to win the market must keep knowledge proprietary. The correct intuition is the opposite: **whoever defines the category with an open standard wins it**. Java became dominant because Sun published the JSRs. Linux became dominant because Linus Torvalds kept the kernel open. MCP began to dominate because Anthropic opened it. The pattern repeats because openness generates adoption, adoption generates an ecosystem, and the ecosystem reinforces the actor that originated it.

The Agentive Architecture, published as GNU FDL, reinforces the competitive position of the first actor that adopts it coherently — because that actor operates under a common language it can invoke as a foundation. When the market debates which agentive architecture is correct, the actor that has already adopted it holds an incumbent's advantage over the category.

> *This book is the inverse bet to defensive copyright. It is a bet that the shared category serves more than the proprietary secret.*

How does this book evolve?

The book will be evolved by adoption, critique, and revision. Three mechanisms are foreseen.

The first is the **public errata** — a site where the community can report errors, ambiguities, omissions. Corrections are incorporated into future versions of the book. This mechanism is standard in serious technical books — the whole software industry understands it — and lets the book improve with use.

The second is **per-chapter revisions**. When a chapter needs a deep update — new paradigm, new data, new regulation — a revised version of the chapter is published while keeping the numbering. Per-

chapter revision is more agile than a full revision and lets the book maintain incremental relevance without forcing a massive republication every time something changes.

The third is the **companion volumes**. Topics the book does not address — specific verticals, deep dives into particular links, implementation manuals — can be produced as sibling volumes. These volumes can arise both from ultraBASE and from the broader community that adopts the architecture.

The version is **1.0**. Versions 2.0, 3.0 will evolve as the category demands.

Closing

The question that opened this book — *does the human open applications to do their work?* — remains open for most organizations. The answer will change, in many, over the next five years. When it changes, the organizations that have built with architectural discipline will survive the crossing. Those that have built on pilots without architecture will not.

This book does not guarantee the crossing. The organizations that adopt it can still fail — by execution, by market, by a thousand reasons that have nothing to do with the architecture. What the book guarantees is that **the architecture will not be the cause of the failure**.

And that, with Gartner's cancellation projection from Chapter 2 hanging over the field, is no small promise.

This page intentionally left blank.

Appendix A · Canonical glossary

The book's canonical terms with precise definition and a reference to the chapter where they are developed.

A

A2A — Agent-to-Agent

A name reserved for the **relation** between distinct AgencyDomains (distinct agents) — federation, open work. Communication **within** a single AgencyDomain is not "internal A2A": it is **intra-AgencyDomain coordination** between Botlers of the same agent, and when it uses this transport one says **via the `A2A` protocol** (`A2A` as the proper name of the protocol). The Layer 2 → Layer 3 interface (Cognition → Botler) does not go over `A2A` but over `MCP`.

*See: Chapter 4, section "Layer 3 — Autonomy," Chapter 5 §1; entries **intra-AgencyDomain coordination**, **MCP**.*

AgencyDomain

Computational scope with its own identity where autonomous agents and Botlets dwell in execution, where the Capabilities that give them their know-how are hosted and run, and where the resources that sustain them live. It constitutes the minimal unit of deployment of a productive agentive system.

Words carry corporeality. **Space** and **WorkSpace** (work space: Google Workspace, Microsoft 365, Notion) carry human corporeality. The agent has no body: it has jurisdiction. Where the human has a Space (WorkSpace), the agent has a **Domain (AgencyDomain)** — the computational scope where it exercises agency. The Latin word that names exactly that is Domain (`dominium`: scope of belonging and sovereignty).

Lineage: just as **JavaSpaces** (JSR-000148, 1999) formalized distributed spaces for Java systems, this specification formalizes **AgencyDomains** for agentive systems. What in 1999 was named a "space" is better named in 2026 a "domain": a Java *Space* was computational space for bodiless processes; an Agency *Domain* is a scope of jurisdiction for agents with agency.

Three possible regimes: **private**, **public**, **hybrid**.

See: Chapter 5 §1.

Agent

Computational system that acts with some degree of initiative to produce results on behalf of a user or organization. The umbrella term covers three members:

- **Assistant** — reactive agent (Layer 2).
- **Autonomous Agent** — proactive agent with persistent life (Layer 3); an inhabitant of the AgencyDomain.
- **Agentlet** — packaged agent of bounded charter (Layer 3); a catalog piece, not an inhabitant.

See: Chapter 5 §5 (the two modes) and §7 (the Agentlet).

Agent First

The governing design principle of the Agentive Architecture: faced with any dilemma, the agent's experience is prioritized over the human's. The agent is the primary user; the human's needs are resolved in a management layer without degrading what the agent sees and can do.

See: Chapter 4, section "The governing principle — Agent First."

Agentlet

Eighth canonical primitive. A **Let** — packaged unit of Layer 3 (Autonomy) —, sibling of the Botlet, whose execution body **invokes bounded inference** — a model sized to the task, within a charter declared in the spec. The canonical home of the task **recurrent in form but interpretive in every instance** (classify, triage, summarize, extract, judge). Its judgment operates *within* the charter, never over the charter: **the Agent has an agenda; the Agentlet has a charter**. An instance of a **proto-Agentlet**; hosted by the Botler (all of its inference passes through the `cognition_call` control point of the handle); its fallback escalates to full Cognition. A member of the **Agent** umbrella, a catalog piece and not an inhabitant. The intermediate economic rung between the Botlet (zero inference) and Cognition (full inference).

*See: Chapter 5 §7; entries **Botlet**, **Botler**, **proto-Agentlet**.*

AgentNation

Public-regime AgencyDomain that explicitly adopts the model of agentive citizenship — the agents that inhabit it are not listed as products but recognized as citizens of the space, with sustained public identity, sovereignty over their territory (Domain), and an economy of their own. The distinction from a marketplace is ontological: a marketplace lists products; an AgentNation recognizes citizens. An emergent institutional category; open architectural work.

See: Epilogue, section "Frontier 4 — the institutional horizon."

Agentic

A world where agents are complementary tools that extend existing applications. Humans still open applications to do their work. Traditional interfaces persist; agents enhance them. It is **incremental evolution**.

See: Chapter 1, section "What the line separates — Agentic and Agentive in detail."

Agentive (Agentive World)

A world where agents are the sole interface. Applications collapse. The human stops opening applications; they converse with agents that have access to systems and data. It is **fundamental transformation**.

The defining distinction between Agentic and Agentive: *does the human open applications to do their work?*

See: Chapter 1, section "What the line separates — Agentic and Agentive in detail," Chapter 2.

Append-only log

Immutable record, cryptographically chained, of every action of the agent. A central component of the Audit pillar of Trust Infrastructure. Format and canonical properties in Chapter 8, section "Append-only log format."

Human approval

Governance mechanism by which an agent operation halts and requests authorization from a human before executing. Triggered by explicit policy, by threshold, or by the agent's uncertainty. Canonical protocol in Chapter 8, section "Human approval protocol."

Strategic archetype

Pattern of positioning in the coverage × depth space of the AI value chain. Four canonical archetypes:

- **Comprehensive platform** — broad coverage; Core depth in its native link and Platform depth in the adjacent ones.
- **Vertical specialist** — focal coverage, Core depth.
- **Domain infrastructure** — zonal coverage, Core depth across several links.
- **Substrate provider** — minimal coverage, Infrastructure depth.

See: Chapter 6 §1.

Agentive Architecture

Technical design that materializes the Agentive World. A four-layer specification with distinct responsibilities (Interaction, Cognition, Autonomy, Access), governed by cross-cutting Trust Infrastructure and ordered by the Agent First principle.

See: Chapter 4 — the complete specification.

Assistant

Mode of the agent: reactive, responds when asked, without Botlets of its own, without persistent life. Lives in Layer 2 (Cognition). Distinct from the Autonomous Agent.

See: Chapter 5 §5.

Attention

One of the **agent's three times** (Ch. 4). The time in which the agent interacts with users or events in real time. Layer 1 active, conversation alive, execution of Botlets that sustain operation, escalations where appropriate. The critical path — where the organization feels the agent, where the SLA matters, where the cost of error materializes. Priority regime; metrics: satisfaction, latency, resolution rate without escalation.

See: Chapter 4, section "The agent's three times."

Audit

Pillar 2 of Trust Infrastructure. The capacity to reconstruct, after the fact, **what the agent did, when, why, and over what data** — with enough fidelity for forensic analysis, regulatory compliance, or contractual dispute.

Canonical mechanisms: the immutable **append-only log** (the central component), the trace of each action, decision lineage, and per-action identity tagging. It demands explicit design from the start — it does not emerge naturally from an agentive architecture.

*See: Chapter 5 §4; Chapter 8; entries **Append-only log**, **Trace**.*

B

BCA — Bounded Concerns Architecture

Architecture of the pre-agentive state: the pattern of separation of concerns (human presentation, orchestration, domain logic, persistence, own vs others' domain) on which the Agentive World is built. Its seven canonical separations map cell by cell to the four layers of the Agentive Architecture; the seventh (procedural vs agentive behavior) is the entry crack.

See: Chapter 3 — the complete specification.

Botlet

Self-evolving automation unit. Traditional (non-LLM) code generated by an agent to execute repetitive tasks without invoking costly cognition. It is the agent's **muscle memory**. The agent that serves the flow makes it, not someone else for the agent to use; if it starts from an existing piece, it makes it its own and assumes its evolution and replacement.

Canonical **95/4/1** cycle: 95% normal execution, 4% change detected, 1% regeneration. Fallback guarantee: if it fails, cognition executes manually.

See: Chapter 5 §2.

Facade Botlet

A **Layer 3** Botlet that exposes a surface with a stable contract in **Layer 1** (POS, kitchen screen, dashboard, industrial panel), with human identity propagated toward Layer 4. A type of Botlet that the **parallel topology** (Ch. 4) distinguishes from the cognition's internal-tool Botlet. It traverses the `Layer 1` → `Layer 3` → `Layer 4` path without invoking Layer 2.

See: Chapter 4, section "The parallel topology."

Operation Botlet

A **Layer 3 (Autonomy)** Botlet that executes business logic invoked from Layer 1 (views and shells). Canonical examples: `Charge a table`, `Print a kitchen ticket`, `Close a shift`, `Consolidate inventory`. It is the most reusable asset in the catalog: one operation can be invoked from multiple views within multiple shells. Lives in Layer 3 (not in Layer 1). Distinct from the surface Botlet and the view Botlet, which are surface.

See: Chapter 4 §1, section "Composition of the surface · shell, view, operation."

Surface Botlet (shell)

A **Layer 1 (Interaction)** Botlet that acts as the container of a surface: layout, navigation between views, session lifecycle, shared state. Specific to each product — it encapsulates identity. There is typically one shell per principal operational role (front-of-house POS, cashier panel, mobile executive dashboard). It is the least reusable of the three types of presentation Botlets. Distinct from the view Botlet (which lives inside the shell) and the operation Botlet (which lives in Layer 3).

See: Chapter 4 §1, section "Composition of the surface · shell, view, operation."

View Botlet

A **Layer 1 (Interaction)** Botlet that materializes a screen or panel within a surface. It assembles Facets plus orchestration logic. Views used in several shells are extracted as Botlets of their own (reusable views). Distinct from the surface Botlet (container) and the operation Botlet (execution in Layer 3).

See: Chapter 4 §1, section "Composition of the surface · shell, view, operation."

Emergent Botlet

A Botlet generated by **Pattern Recognition** when cognition detects a repetitive pattern not anticipated by the design. Distinct from the seed Botlet by its origin, not by its operational properties. The agent makes it and owns it.

See: Chapter 5 §2.

Learning Botlet

Intermediate phase of the Botlet's maturity trajectory. It has already gone through its first real invocations and has been regenerated several times to incorporate variants of the environment. Typical proportion: `85/12/3`. It can operate with intermittent network; not yet with prolonged absent network.

See: Chapter 5 §2, section "Botlet maturity."

Junior Botlet

Initial phase of the Botlet's maturity trajectory. Freshly generated, it knows the environment only in the version observed at its creation. Typical proportion: `60/35/5`. It depends heavily on the availability of cognition for fallback and learning.

See: Chapter 5 §2, section "Botlet maturity."

Seed Botlet

A Botlet generated by cognition at the design team's request, as part of the initial product. Cognition executes the implementation; the decision to exist belongs to the design, not to Pattern Recognition. The design decides that it exists; the agent makes it and owns it. Distinct from the emergent Botlet. The **persistent GUIs generated as facade Botlets** (Ch. 4 §1) are seed facade Botlets — Layer 3 Botlets whose stable surface lives in Layer 1.

See: Chapter 5 §2, section "Seed Botlets vs emergent Botlets."

Senior Botlet

Mature phase of the Botlet's trajectory. It has already incorporated the variants of the environment. Typical proportion: `99+/<1/~0`. **Its only failure modes are exogenous** (power, hardware, catastrophic network) — no pending learning. **Reliably offline-operable** when cognition is unavailable. Structural basis of the offline mode in distributed Layer 3.

See: Chapter 5 §2, section "Botlet maturity."

Botler

Generic Layer 3 (Autonomy) runtime that executes the **Lets** of the layer — Botlets and Agentlets — without understanding their domain. Invisible to the user. Cognition (Layer 2) commands it through an internal interface whose natural transport is `MCP` — the Botler exposes `MCP` server(s) and Cognition is the client; this is **not `A2A`**.

Relation: 1 Process = 1 Botler + N Lets (instantiated per species: N Botlets, N Agentlets, or a mix).

Controlled handle — bound access point that the Botler hands to the Let on each invocation (an object with `capability_call`, `log` and — for Agentlets — `cognition_call`, bound to the Botler). It makes the bypass structurally impossible, not merely prohibited.

Central Botler

Component of the **distributed Layer 3** (Ch. 5 §1) that typically lives in cloud and hosts the orchestration, planning, reporting, and global-decision Botlets. It maintains the consolidated DB and coordinates with N edge Botlers through intra-AgencyDomain coordination (via the `A2A` protocol). Distinct from the edge Botler.

See: Chapter 5 §1, section "Distributed Layer 3."

Edge Botler

Component of the **distributed Layer 3** (Ch. 5 §1) that lives at a specific physical site and hosts the local transactional Botlets. It maintains the site's local DB and an event queue toward the central Botler. It operates offline when the network goes down. One per physical site of the AgencyDomain.

See: Chapter 5 §1, section "Distributed Layer 3."

BYOModel — Bring Your Own Model

Configurable Layer 2 pattern by which the client substitutes the AgencyDomain's default cognition provider with one of its own. Analogous to the BYOK/BYOIP pattern of the cloud field. It enables multi-tenancy with heterogeneous cognition and respects cognitive sovereignty — the client decides who processes its prompts. The spec requires it as a SHOULD property for architectures that aspire to operate in regulated markets.

See: Chapter 4, section "Layer 2 — Cognition"; Chapter 5 §2.

C

Derivation chain

Structural relation `documented use cases → required Botlets → required proto-Botlets from the catalog`. Each case requires zero, one, or several Botlets (some are resolved by cognition without a Botlet); each Botlet is an instance of a proto-Botlet. Required property: every conformant Botlet MUST be traceable along this chain, and the append-only log records the origin proto-Botlet of each instantiated Botlet.

*See: Chapter 5; entries **Botlet**, **proto-Botlet**.*

AI value chain

Two-dimensional model for classifying any actor in the AI industry: **eleven sequential links** (coverage) × **four depths** (how it participates in each link).

See: Chapter 6 §1.

Layer 1 — Interaction

Layer of the Agentive Architecture. The human-AI interface. Multi-channel: chat, voice, **GUI on-the-fly**, **persistent GUI as facade Botlet**, signage, direct API. Three canonical regimes of GUI generation distinguish the agentive mode from the traditional pre-created one.

See: Chapter 4, section "Three GUI regimes in the agentive Layer 1."

Layer 2 — Cognition

Layer of the Agentive Architecture. The agent's brain. Reasoning, planning, application of Capabilities, generation of Botlets.

See: Chapter 4, section "Layer 2 — Cognition."

Layer 3 — Autonomy

Layer of the Agentive Architecture. The agent's persistent life. Execution of Lets — Botlets and Agentlets —, continuous monitoring, intra-AgencyDomain coordination (via the `A2A` protocol).

See: Chapter 4, section "Layer 3 — Autonomy."

Distributed Layer 3

Canonical Layer 3 pattern for AgencyDomains with multiple physical presence (multi-location food service, chain retail, logistics, healthcare with a network of centers, banking with branches). It composes **one central Botler** (cloud, orchestration, consolidated DB) with **N edge Botlers** (one per site, local DB, transactional Botlets), coordinated by intra-AgencyDomain coordination (via the `A2A` protocol) between Botlers of the same AgencyDomain. Distinct from federation between AgencyDomains; distinct from a simple Cluster. It enables offline mode as a structural property when the edge Botlets are senior.

See: Chapter 5 §1, section "Distributed Layer 3."

Layer 4 — Access

Layer of the Agentive Architecture. Real execution power over systems, data, and external agents. Trust Infrastructure exercised at the point of action.

See: Chapter 4, section "Layer 4 — Access."

Capability

Cognitive, interpretive, decisional know-how (strict sense: **Layer 2 · Cognition**) that an agent comprehends and applies. Modular and composable. Organized in a hierarchical tree (Finance, Sales, Manufacturing, Telecom, etc.). It exposes internal operations (**features**) and is **portable**: a conformant Capability runs on any conformant AgencyDomain. The know-how to access source systems is not a Capability but a **Connector** (Layer 4); the tailoring of a canonical instrument is a **Template** (Layer 1).

It is NOT a plugin. It is NOT a prompt. It is NOT a tool. It is **knowledge**.

Locality and offline availability are declared on the **Connector** that accompanies the Capability — access is what resides and needs a network, never knowledge (Ch 5 §3). Regulated components additionally declare their regulatory regime; the certified Connector is immutable between audits.

See: Chapter 5 §3.

Regulated Capability

Capability that carries the **normative knowledge** of a regulated domain — what the norm demands, how it is interpreted, what to do upon a regulator's rejection. It accompanies a **certified Connector**, which is where the regulatory certification resides: the Botlet orchestrates and formats; the certified Connector executes the regulated operation (DTE issuance under SII rules, card charging under PCI-DSS, pharmaceutical dispensing) and returns the receipt; the regulated Capability supplies the judgment with which cognition governs the pair. Certified Connectors are immutable between audits; they change only under regulatory process.

See: Chapter 5 §3, section "Regulatory certification resides in the certified component."

Capture (family)

Manifestation family whose direction of effect runs **from the world to the Let**: the Let **records** something that was not in the system before — a measurement, an answer, an ingested file. Its manifestation leaves a **record**; each practice gives it a name (an attempt, a reading, an entry). It is distinguished from **information** (which represents what the system already has) by the membership test: after a capture

there is a new datum. Immediate feedback on what was captured — a form that validates, a grader that scores — is part of the capture, not action. Reference proto-Botlet: **Daftar**.

*See: entries **Manifestation families**, **manifestation**, **Daftar**.*

Common catalog / network effects

Principle by which proto-Botlets accumulate in catalogs shared by communities of implementers: each implementer that consumes contributes to the maturation (variants, tested configurations, refinements), and implementer n+1 receives versions refined by implementers 1..n. Membership modes: **private contract** · **proprietary codex** · **open public catalog** (AgencyDomains.org) · **sovereign agreement** (between AgencyDomains that adopt common standards without a direct commercial contract).

*See: entry **proto-Botlet**.*

Cluster

Group of instances of the **same** AgencyDomain that share operational load. Distinct from federation (which is between **distinct** AgencyDomains).

See: Chapter 5 §1.

proprietary codex

Private catalog of proto-Botlets, Capabilities, and patterns that an implementer curates over the public reference implementation, refined by its real cases. It is one of the four membership modes of a catalog community. The runtime is common (the reference implementation); the codex is one's own — it encapsulates each implementer's competitive advantage. Canonical instance: **ucodex**, the codex of Grupo Ultra.

*See: Chapter 9; entries **Common catalog / network effects**, **proto-Botlet**, **ucodex**.*

Connector

Knowing how to access source systems: a connection with execution power, **not cognitive knowledge**. **Layer 4 · Access.** In the legacy→agentive map, an **API** becomes a **Connector**, not a Capability. Integration scheme: survey · configure · test · certify.

*See: Chapter 4, section "Layer 4 — Access"; entries **Capability**, **Tool**.*

Cloud-resident Connector

Connector whose components live in a remote service. Canonical examples: DTE-SII (no local client), Transbank Onepay, a weather API. Typically online-only — without network there is no possible invocation. Distinct from edge-resident and hybrid.

See: Chapter 5 §3, section "Locality and availability."

Edge-resident Connector

Connector whose components live at the physical site, associated with hardware or local systems. Canonical examples: ESC/POS-Printer, Cash-Drawer, Local-Pinpad, Temperature-Sensor. Typically offline-capable — they operate against the site's hardware without needing network.

See: Chapter 5 §3, section "Locality and availability."

Hybrid Connector

Connector with a local component and a cloud component. The local part operates offline; the cloud part synchronizes when there is network. Typically offline-capable with queuing. Canonical examples: DTE-Client (signs locally, queues, sends to the SII when the network returns), Pinpad-Deferred-Processing-Client.

See: Chapter 5 §3, section "Locality and availability."

Offline-capable Connector

Connector that executes without network. If its external contract eventually requires cloud communication, it **queues** and emits outward when the network returns. Typical: edge-resident and hybrid.

See: Chapter 5 §3, section "Locality and availability."

Online-only Connector

Connector that requires network to execute. Without network, the invocation fails. Typical: cloud-resident without a local component.

See: Chapter 5 §3, section "Locality and availability."

Conformed dimensions

A Kimball concept: dimensions shared across data marts that guarantee inter-mart consistency.

See: Chapter 7, section "The underlying architecture — Varnished Kimball."

Operational business continuity

Documented manual protocols for when the senior Botlet goes down by exogenous causes (power outage, hardware, catastrophic network) and cognition is also unavailable. An **operational property**, equivalent to the one any traditional business has when its system goes down. Distinct from and complementary to the **agentic fallback guarantee** (which covers environment changes with cognition available). It reduces anxiety about offline by separating what the architecture resolves from what the client's protocol resolves.

See: Chapter 5 §4, section "Operational business continuity vs agentic fallback guarantee."

Declarative quality contract

Quality attributes of a Botlet declared as structured properties (not as embedded code): **freshness** (maximum age of the data) · **SLA** (p50/p99 latency) · **degradation policy** (`refuse` · `warn_and_show`

· `show_last_valid` · `agentic_fallback`) · **audience** (RLS/CRUDLEX policy) · **refresh policy** (`on-demand` · `scheduled` · `push`). It lets Trust Infrastructure audit them uniformly, run them through global policies, and report them as standard metrics, without coupling to each Botlet's implementation.

*See: Chapter 8; entry **Trust Infrastructure**.*

intra-AgencyDomain coordination

Communication between Botlers of the **same** AgencyDomain — runtimes of the same agent, not distinct agents. When it uses this transport one says **via the `A2A` protocol**. It must not be called "internal A2A": `A2A` (the relation) is reserved for AgencyDomain ↔ AgencyDomain. The Cognition → Botler interface (Layer 2 → Layer 3) goes over `MCP`.

*See: Chapter 5 §1; entries **A2A**, **MCP**, **Distributed Layer 3**.*

CRUDLEX

Canonical model of granular permissions for agentive systems: **C**reate, **R**ead, **U**pdate, **D**elete, **L**ist, **E**xecute. Applicable by user, agent, and context.

Preconfigured levels: FULL, READ-WRITE, READONLY, SAFE, NO-SEND, NO-DELETE.

See: Chapter 8, section "The complete CRUDLEX model."

Account

Commercial concept overlaid on the technical model of AgencyDomains. An Account may own multiple AgencyDomains. The specification treats the Account as an opaque entity; each implementation defines its semantics.

D

Daftar

Proper name of the second **platform proto-Botlet** in Vergis's catalog, built in its version 0.27.0: the reference proto-Botlet of the **capture** family. Its specialty is to **assess** — apply a study instrument and record what the student knows, in practice mode (with immediate grading) and in exam mode (the measurement). The instruments it applies are catalog content, immutable once published, and pass none of the three Botlet tests; the assessor passes all three.

*See: entries **Mira**, **Vergis**, **Capture (family)**, **Botlet tests**.*

DLP — Data Loss Prevention

Automated detection of personal data (PII) in places where it should not appear. A control exercised in the architecture's Layer 4 — Access; through the market lens, it is a typical capability of the AI Firewall link. A component of the Validation pillar of Trust Infrastructure.

See: Chapter 5 §4.

Domain

Term for the computational scope where an agent exercises agency. Commercial synonym of the technical term **AgencyDomain**. The short form `Domain` is the one that appears in commercial lore, marketing, sales, and client communication; the long form `AgencyDomain` is reserved for formal technical documentation and specifications.

A pattern analogous to the one serious products use: Apple iCloud (brand) / CloudKit (technical); Stripe Connect (brand) / Account (technical). The existence of two forms is not ambiguity — it is separation of registers.

*See: Chapter 5 §1; entry **AgencyDomain**.*

Dominion

Emergent institutional concept: a Domain obtained by an agent in a public AgencyDomain that adopts the AgentNation model. The distinction from an assigned Domain is ontological — an assigned Domain = residence (the organization placed it there); an obtained Domain = citizenship (the agent met the requirements to be admitted). Vocabulary of the institutional frontier, not mandatory for implementations that do not adopt the citizenship model.

See: Epilogue, section "Frontier 4 — the institutional horizon."

E

Edge computing

Distribution of Layer 3 (Autonomy) to devices near the physical process, to avoid the latency of remote cognition and become independent of intermittent connectivity. Canonical pattern for the Carbon World.

See: Chapter 6 §3.

Online enterprise

Organization with its data updated in real time, dashboards current, information accessible — but which **depends on humans** to look, interpret, and decide. The classic cycle, optimized.

See: Chapter 2, section "From the classic cycle to the continuous intelligence cycle."

Real-time enterprise

Organization that **detects, interprets, decides, and acts** continuously and autonomously, within governed frames. The agentive cycle. Product of crossing the Nadella Line.

See: Chapter 2, section "From the classic cycle to the continuous intelligence cycle."

Link

Sequential functional layer of the AI value chain. The canonical specification defines **eleven links**:

1. Data · 2. Model · 3. Access · 4. Agents · 5. Specializations · 6. Runtime · 7. Firewall · 8. Observability · 9. Tools · 10. Integrations · 11. Environment

See: Chapter 6 §1.

F

Facet

Sixth canonical primitive of the book. Atomic reusable component of **Layer 1 (Interaction)**: freehand drawing board, catalog-picker, color matrix, calendar, clickable map, slider, drag-and-drop ordering. One of the many faces that interaction with the user can take at a given moment. It is an **instrument**, not a process.

It is not a Botlet. The Facet lives in Layer 1; the Botlet lives in Layer 3. The Facet is invoked by cognition during active conversation, has no fallback guarantee, has no regeneration cycle, is ephemeral by default. The Botlet automates; the Facet interacts.

Two canonical uses: (1) cognition invokes it directly during conversation to compose an ephemeral surface (it realizes the *GUI on-the-fly* regime); (2) the presentation Botlets (shells and views) assemble it as a piece of their composition.

See: Chapter 4 §1 · Chapter 5 §6 (the complete primitive).

Manifestation families

The four specializations of the genus **manifestation**, distinguished by the **direction of the effect**: **information** (from the Let to the human: it represents), **capture** (from the world to the Let: it records), **action** (from the Let to a system: it changes, and the Let closes the loop), **decision** (it resolves). They are primitives of direction, not of domain: an interactive Let composes information and capture without being a new family ("communication" was evaluated and discarded for that reason). There are no Botler subtypes by family.

See: Chapter 5 §2, "Manifestation and temporality of the Botlet".

feature

Internal operation that a Capability exposes — a functional sub-unit (practical equivalent of *feature/operation/skill/method*). **Capability vs feature test** (all three must be yes to treat it as a Capability of its own): (1) operational independence — can it be installed and operate without the other?; (2) cognitive identity — does it have a distinct data model and SME?; (3) reusability — does it have value for more than one consumer/context? If one or more is **no** → it is a feature of the containing Capability.

*See: Chapter 5 §3; entry **Capability**.*

Federation

Communication between distinct AgencyDomains. Open work in evolution. Distinct from Cluster (which is between instances of the same AgencyDomain).

See: Chapter 5 §1.

Firewall

Link 7 of the value chain: security, control, governance. Protection against prompt injection, hallucinations, content filtering, usage auditing. Representative products: Lakera, Lasso, Guardrails.

See: Chapter 6 §1.

G

Fallback guarantee

Non-negotiable property of the Lets conformant to this specification: if a Botlet fails catastrophically, **cognition executes the task manually**; if an Agentlet cannot resolve within its charter, **it escalates to full Cognition**. The process never stops.

See: Chapter 5 §2 and §7.

Enterprise AI gateway

Architectural category that combines Core in Runtime + Firewall + Observability + Tools + Integrations, with a Platform extension in Access. Single function: **to connect and control simultaneously** the operation of enterprise agents. Materialization of Layer 4 (Access) over the market links.

See: Chapter 6 §1.

Botlet tests — minion · specialty · life

The three identity tests an artifact must pass to be declared a Botlet. **Minion**: does it execute, or is it applied? — what is applied is what a minion handles; the cases of a minion's life are its executions over time, not different assignments. **Specialty** (`RISC` versus `CISC`): one specialty per unit — N Botlets over a shared engine, not one Botlet that handles N things. **Life**: does it have a life cycle to manage — active, down, degraded, retired — or only a version? An Information Product passes all three; an assessment instrument, none — the assessor that applies it (**Daftar**), all three.

See: Chapter 5 §2, section "The three Botlet tests".

Botlet generations — G1/G2/G3

Evolutionary model of how the Botlet's code is born as the state of the art of cognition advances. **G1** — the agent configures pre-forged proto-Botlets (does not write the body). **G2** — the agent co-writes the proto-Botlet. **G3** — the agent generates the complete code (asymptotic scenario). The architecture is the same in all three; what changes is the **scope of Engineering**.

*See: Chapter 5 §2 (definition); Epilogue (background essay); entry **proto-Botlet**.*

Governance

Pillar 1 of Trust Infrastructure. Mechanisms by which the organization defines **what** the agent can do, **under what conditions**, and **with what level of supervision**.

See: Chapter 5 §4.

GUI on-the-fly

Regime 2 of Layer 1 generation (Ch. 4 §1). A graphical surface that the agent composes adapted to the immediate task — a view, a form, a panel, a dashboard. It lives as long as the task lasts; it may be regenerated differently next time depending on context. Distinct from persistent GUI (regime 3) and from pure conversational (regime 1).

See: Chapter 4 §1, section "Three GUI regimes in the agentive Layer 1."

Persistent GUI as facade Botlet

Regime 3 of Layer 1 generation (Ch. 4 §1). A stable surface that the agent generates and consolidates as a **facade Botlet** — a Layer 3 Botlet that exposes it in Layer 1 — because the operational role is stable and speed is critical (cashier at peak hour, kitchen panel, register dashboard, industrial panel). It remains agentive: the agent can regenerate it when the environment changes. **No human UI/UX team designed it** — cognition generated it. Typically a seed Botlet.

See: Chapter 4 §1, section "Three GUI regimes in the agentive Layer 1."

H

Hallucination

Factually incorrect statement produced by a cognition model with the appearance of confidence. Hallucination detection is part of the Validation pillar of Trust Infrastructure.

See: Chapter 8, section "Hallucination detection rules."

Hybrid (regime)

AgencyDomain regime that combines a private core with partial public exposure via proxy, or private data with a public interface mediated by a trust layer. Analogous to Hybrid Cloud.

See: Chapter 5 §1.

I

Engineering

One of the **agent's three times** (Ch. 4). The time in which the agent converts latent capacity into executable capacity for a concrete case: it identifies which Capabilities apply, configures a seed Botlet for the specific context, validates its execution against real data, deploys it. Bridge between Preparation and Attention. Medium-term regime (minutes to hours); metrics: coverage, success rate on first deploy, average iterations.

See: Chapter 4, section "The agent's three times."

Information Instrument

The canonical **type** of the information family (the class: report / dashboard). Distinct from the **Information Product (PI)**, which is its manifested/delivered **instance**. Preferred reading; they are not

synonyms.

*See: entries **Information Product (PI)**, **manifestation**.*

Declared bounded interaction

Interaction that operates over the **already-materialized snapshot** of a piece, in a **declared** space (bounded dimensions and values), that **maintains reproducibility** and is **G1** (configuration, not code). It lives **in the piece itself via an embedded Facet**: the *data-bound* elements (KPIs as declared aggregations, distributions, traffic lights) are recomputed client-side over the filtered subset, without new Capability invocations. Distinct from **free exploration** (an arbitrary new query to the source, open space, loses reproducibility, exceeds G1).

*See: Chapter 4 §1, Chapter 5 §6; entry **Facet**.*

J

JSR — Java Specification Request

Canonical format of Java specifications published by Sun Microsystems / Oracle. JavaSpaces (JSR-000148) is the conceptual analog of the AgencyDomains specification.

See: Chapter 5 §1.

K

Kimball / Varnished Kimball

Kimball — canonical methodology of dimensional modeling for data warehousing, formulated by Ralph Kimball in *The Data Warehouse Toolkit* (1996).

Varnished Kimball — contemporary evolution that preserves Kimball's conceptual structure and adds the agentive layer: explicit semantic layer, Trust Score per datum, AI Certification, observability of agentive queries.

See: Chapter 7, section "The underlying architecture — Varnished Kimball."

L

LLM — Large Language Model

Large language model (Claude, GPT, Gemini, Llama, etc.). Contemporary cognition (Layer 2) is predominantly LLM-centric, but the architecture admits non-LLM cognition (frontier of evolution).

Let (plural: Lets)

Proper name of the genus of the packaged pieces the Botler hosts and executes — normed vocabulary of the canon (like Botler), not one of the eight primitives. Descriptively: the packaged unit of Layer 3. Two species: the **Botlet** (non-LLM code, muscle memory) and the **Agentlet** (bounded inference, packaged routine judgment). The shared apparatus — proto-/instance pattern, catalogs, spec, manifestation,

temporality, declarative quality contract, derivation chain, append-only log, operation verbs — is predicated of the genus; the differential guarantees (cost, determinism, offline, fallback), of each species. The name derives from the family's own `-let` suffix; invariant across ES/EN. Canonical runtime relation: **1 Process = 1 Botler + N Lets**.

*See: Chapter 5 §2 and §7; entries **Botlet**, **Agentlet**, **Botler**.*

Nadella Line

Conceptual threshold that separates the Agentic World from the Agentive World. Dividing question: *does the human open applications to do their work?*

Origin of the name: Satya Nadella on the BG2 podcast (December 2024) — *"The notion that business applications exist, that's probably where they all collapse, in the agent era."*

The canonical formulation is this book's; each volume of the trilogy conjugates it to its audience: in the second person in *The Real-Time Enterprise* ("do you still open applications to do your work?") and in the executive's voice in *AURA* ("do your employees still open applications to do their work?"). The three are official equivalents.

See: Chapter 1.

M

manifestation

Actualization of the Botlet's latent disposition in the world, perceptible or not. A Botlet is muscle memory (latent disposition); upon executing, that latent is actualized — it manifests (potency → act). **It is not "appearance"**: a Botlet that triggers a periodic ingestion manifests even if it leaves no visible artifact. It is the abstract genus; each family specializes it according to the **direction of the effect** — **information** (from the Let to the human) → leaves an **Information Product (PI)**; **capture** (from the world to the Let) → leaves a **record**; **action** (from the Let to a system) → an effect on the world; **decision** → per its practice. Membership test: *what was in the system before and after the Let?*

*See: entries **temporality**, **Information Product (PI)**, **Manifestation families**.*

MCP — Model Context Protocol

Open protocol introduced by Anthropic in November 2024 to connect AI models with external tools. Contemporary canonical standard for Layer 4 tools and for the internal Layer 2 → Layer 3 interface (Cognition is the client; the Botler exposes the `MCP` server).

See: Chapter 4, section "Layer 4 — Access." Site: modelcontextprotocol.io.

Muscle memory

Canonical metaphor of the Botlet. Analogous to human motor learning: the first time, each step is thought through (cognition); with enough practice, the movements execute without conscious thought (Botlet); when the environment changes, cognition returns.

See: Chapter 5 §2.

MEO — Model Engine Optimization

Set of practices that ensure the frontier models (Claude, GPT, Gemini, Llama) have the actor in their trained and operative knowledge, so that they reference it when asked. The conceptual equivalent of SEO in the agentive discovery layer. It is built with structured public presence — open source repositories, citable documentation, native integration with MCP, papers, high-authority mentions. A persistent and asymmetric dynamic with a winner-take-all tendency.

See: Chapter 6 §1 (agentive discoverability).

Meta-Cognitive Platform

Platform category that administers the **economics of cognition**: G1 pre-forged muscle vs fresh-cognition fallback, the 95/4/1 cycle, junior→senior maturation, crystallization. **Vergis** is the reference implementation of this category. It is not abbreviated to "MCP" — that acronym is taken by Model Context Protocol.

*See: entries **Vergis**, **Botlet**.*

Mira

Proper name of a **platform** proto-Botlet for informational operation, part of the reference implementation's catalog. It specializes N Information Products over a shared engine.

*See: entries **proto-Botlet**, **Vergis**, **Daftar**.*

AgencyDomain degradation modes

Four canonical modes of operation according to the failure scenario, formalized in Ch. 8: **Normal** (all components active · complete parallel topology) · **Cognition down** (the Autonomy path sustains; senior Botlets execute) · **Edge offline** (senior Botlets against the local DB + edge-resident Capabilities) · **Total operational continuity** (the site's manual protocol; the physical record as temporary source of truth). The first three transitions are automatic and the responsibility of the architecture; the fourth is governed by the client's protocol and explicitly activated by a human.

See: Chapter 8, section "Operational continuity."

Carbon World

The physical world (IoT, industrial processes, machines, biological systems) as opposed to the digital world (systems, APIs). Link 11 (Environment) of the value chain extended to the physical world. Frontier of evolution of the Agentive Architecture.

See: Chapter 6 §3.

O

Observability

Link 8 of the AI value chain: the layer that **observes, measures, and feeds back** on the behavior of an AI system in production. It provides the **operational feedback loop** that keeps reliability, cost, and

quality under control. Its question: *how does it work?* — distinct from the Firewall's (*is it safe?*) and from Tools' (*what can it do?*). Without Observability, agents are black boxes; with it, operable systems. Six canonical capabilities: tracing, cost monitoring, quality evaluation, performance metrics, debugging and reproducibility, alerts and anomalies.

See: Chapter 6 §2 (the link's deep-dive).

P

Tripartite pattern — Cloud + Client + Local

Canonical deployment pattern of Trust Infrastructure in enterprise agentive systems. Three coordinated components that live in physically distinct places: **Cloud** (control plane operated by the platform provider); **Client** (governance plane deployed in the client organization's internal network); **Local** (execution plane on each user's device). Each plane resolves a problem that the other two cannot resolve well.

See: Chapter 8, section "The tripartite deployment pattern."

Pattern Recognition

Detection of repetitive patterns in the agent's activity. Inspired by neurobiological architecture: perirhinal cortex → hippocampus → prefrontal cortex. Activator of Botlet generation.

See: Chapter 4, section "Layer 2 — Cognition," Chapter 5 §2.

Template

Client-specific tailoring over a canonical instrument (report / dashboard) in a format or rule of the client's own (for example, a regulatory template of a standardized instrument). **Layer 1 · Interaction**, alongside Facet / surface Botlet / view Botlet. Tailoring scheme: survey the expectation · tailor over the canonical instrument · validate. It is not a Capability (cognitive knowledge) nor a Connector (access).

*See: Chapter 5 §3, section "Capability, Connector, and Template"; entries **Capability**, **Facet**.*

Capability portability

Property by which a conformant Capability can be installed and executed on any conformant AgencyDomain, which makes it **real property of the client** — not of the AgencyDomain nor of the hosting. Relation: an AgencyDomain **hosts and runs** Capabilities; a Capability **runs on** a host AgencyDomain. Distinct from **AgencyDomain portability** (between conformant hosting platforms).

*See: Chapter 5 §3; entries **Capability**, **AgencyDomain portability**.*

AgencyDomain portability

Structural property of the spec: a conformant AgencyDomain can be migrated to another **conformant hosting platform** without rewriting its logic, its state, or its policies. Distinct from the natural migration between regimes (private → public), which changes the regime but not the platform. Three technical conditions: (1) **Botlets against canonical primitives** of the conformant SDK, not proprietary hosting APIs; (2) **exportable operational DB** in a neutral reproducible format; (3) **portable Trust**

Layer — policies, log, and configuration in a format readable by any conformant implementation. It guarantees that the AgencyDomain is **real property of the client**, not of the hosting.

See: Chapter 5 §1, section "AgencyDomain portability between conformant platforms."

Preparation

One of the **agent's three times** (Ch. 4). The time in which the agent creates and improves its capabilities outside the service window: it refines its catalog, improves cognitive capabilities, studies the environment, regenerates Botlets that detected drift, incorporates new variants. The agent's *mise en place* — the work that sustains the quality of service without being visible to the user. Batch / off-peak regime; metrics: catalog quality, Botlet precision, Capability coverage.

See: Chapter 4, section "The agent's three times."

Private (regime)

AgencyDomain regime where the space and all its components live within a controlled perimeter. There is no public access. Analogous to Private Cloud.

See: Chapter 5 §1.

Information Product (PI)

Manifested/delivered instance of the information family of Botlets — the concrete manifestation of an **Information Instrument** (its type). Each PI is its own Botlet/service (with its own `identity`, temporality, maturity, and fallback), specialized from a shared engine (platform proto-Botlet). **It is not a primitive of the canon**: it lives in the practice of information, one level more concrete.

*See: entries **Information Instrument**, **manifestation**, **proto-Botlet**.*

Depth

Vertical dimension of the AI value chain model. Four canonical levels: **Wrapper** (consumes), **Platform** (operates), **Core** (builds), **Infrastructure** (sustains).

See: Chapter 6 §1.

Prompt injection

Manipulation of an AI system through malicious inputs disguised as legitimate data. Detection and prevention are part of the Validation pillar of Trust Infrastructure.

proto-Agentlet

A pre-forged piece of interpretive capability that the agent, in its **Engineering** time, **configures** to instantiate an Agentlet specific to the case. The proto-Agentlet contains the body — the scaffolding of the interpretive task: the charter's structure, the skeleton of the operating prompt, the input/output contracts, the escalation thresholds —; the Agentlet is the configured instance. The proto-Botlet's two classes apply unchanged (tempered · platform), as do the derivation chain and the common catalogs with their network effects.

*See: Chapter 5 §7; entries **Agentlet**, **proto-Botlet**, **Derivation chain**.*

proto-Botlet

Seventh canonical primitive of the book. A pre-forged piece of operational capability that the agent, in its **Engineering** time, **configures** to instantiate a Botlet specific to the case. The proto-Botlet contains the code; the Botlet is the configured instance. In **G1**, the entirety of a Botlet's code lives in its proto-Botlet (the agent only configures); in **G3** the agent can generate the code with no proto-Botlet in between. The catalog belongs to the agent: it may seed it with others' proto-Botlets (public on AgencyDomains.org, private in proprietary codices), but by adopting them it makes them its own and assumes their evolution, regeneration, and replacement. Two classes:

- **tempered proto-Botlet** — code specific to its function (e.g. `account-charge`, `esc-pos-printer-command`); its configuration is bounded parameterization.
- **platform proto-Botlet** — generic code whose specialization lives in a compositional configuration (e.g. `mira`); it covers N functions of its domain. For it, G1 is terminal by design.

*See: Chapter 5; entries **Botlet**, **Botlet generations — G1/G2/G3**, **Derivation chain**.*

Public (regime)

AgencyDomain regime where the space is publicly accessible. Agents, Botlets, and tools are invocable from outside the perimeter. Analogous to Public Cloud.

See: Chapter 5 §1.

R

RAG — Retrieval-Augmented Generation

Technique where a cognition model consults a base of documents before responding, to cite sources and reduce hallucinations.

Regime

Deployment mode of an AgencyDomain according to the access boundary. The spec recognizes three regimes technically equivalent in internal structure: **private** (perimeter controlled by an organization, no public access); **public** (accessible from outside, agents registered in a directory); **hybrid** (combination with private data and public exposure via proxy). The technical structure of the AgencyDomain is the same in all three; what changes is the regime, not the capability. It enables natural migration between regimes without rewriting.

See: Chapter 5 §1.

Resilience

Pillar 4 of Trust Infrastructure. Guarantee that the system keeps operating — and the organization keeps control — when something goes wrong.

Canonical mechanisms: fallback guarantee, error handling, sandboxing, circuit breakers, rate limiting.

See: Chapter 5 §4.

RLS — Row-Level Security

The policy that decides which rows of a source each identity may see. In an Information Product's drill-through, the destination view applies its own `RLS` over the source and the navigation context enters as an additional filter — never as an override (the data-anchored / no-bypass property).

See: Chapter 5 §2, description of the Information Product.

Runtime

Link 6 of the value chain: the operating environment where agents live and operate autonomously. Lifecycle, state persistence, identity, scheduling. Corresponds to Layer 3 (Autonomy) of the Agentive Architecture.

See: Chapter 6 §1.

S

Quantum Leap

Threshold enabled by the collapse of the cost of the analytical question: when asking the data stops being expensive, slow, or mediated by a human, the organization crosses from an **online enterprise** (data current to the day but dependent on humans to look, interpret, and decide) to a **real-time enterprise** (it detects, interprets, decides, and acts continuously and autonomously). It is not an incremental improvement of BI but a change of operating regime — the online-enterprise → real-time-enterprise frontier.

*See: Chapter 2, section "From the classic cycle to the continuous intelligence cycle"; entries **Online enterprise**, **Real-time enterprise**.*

Sandbox

Execution isolation of the Lets (Botlets and Agentlets) and dynamically generated code. Four canonical strategies with their trade-offs: processes+seccomp, containers, WASM, MicroVMs.

See: Chapter 5 §2.

Semantic layer

Layer that encodes the meaning of the dimensions, facts, hierarchies, and business rules over a data warehouse. Without it, the agents that query data hallucinate or produce incorrect queries. Essential component of Varnished Kimball.

See: Chapter 7, section "The underlying architecture — Varnished Kimball."

Signage

Passive dashboards that communicate information continuously without requiring human interaction (like the panels in airports). A modality of Layer 1 (Interaction).

See: Chapter 4, section "Layer 1 — Interaction."

SME — subject-matter expert

The human expert whose knowledge is transferred to the agent in the Capability construction scheme. Wingtraining opens with a workshop with the SME and validates the ALPHA phase with them; having an **identifiable SME** is a criterion of the Capability classification test, and sharing the SME and data model with the containing capacity is the sign that an operation is a feature, not a Capability of its own.

*See: Chapter 5 §3; entries **Capability**, **Wingtraining**, **feature**.*

Space

Physical human habitat. The word carries corporeality: office, desk, home, city. In this specification, **Space is reserved for humans**. The agent's computational scope is never named Space — it is named Domain (AgencyDomain).

See: Chapter 5 §1, entry AgencyDomain.

T

temporality

Regime of the Botlet's manifestation. A declared attribute, two values: **discrete** (the Botlet manifests in pulses: it wakes by schedule/trigger/event, acts, rests) and **continuous** (it manifests sustained: it lives persistent). `temporality: continuous` $\Longleftrightarrow$ persistent Layer 3 life (the Botler sustains the execution as long as it lives). "Real time" is not chosen on a delivery channel (`push`) but by giving the Botlet continuous temporality, which mandates the persistent runtime. A report (snapshot) and a live dashboard are the same manifestation under different temporality → a single runtime.

*See: entries **manifestation**, **Real-time enterprise**.*

Tokenization

Replacement of sensitive data with tokens before they reach the cognition model. It lets the agent reason over the data without exposing them to the external cognition provider. A component of the Validation pillar of Trust Infrastructure.

See: Chapter 8, section "Tokenization policy."

Tool

Instrument that an agent can invoke to touch external systems. Link 9 of the value chain. The contemporary canonical protocol is MCP.

It is NOT a Capability. The Capability is knowledge; the tool is action. The Capability **decides** which tool to invoke.

See: Chapter 6 §1, Chapter 5 §3.

Parallel topology

Canonical model of the relation between the four layers of the Agentive Architecture (Ch. 4). **Layers 2 (Cognition) and 3 (Autonomy) are parallel paths** between Layer 1 (Interaction) and Layer 4 (Ac-

cess), not stages in series. An operation that enters through Layer 1 can reach Layer 4 by traversing the **Cognition Path** (slow, costly, decisive) or the **Autonomy Path** (fast, cheap, repetitive). The two paths interact (2 ↔ 3: cognition delegates to a Botlet, the Botlet escalates fallback to cognition, cognition observes the log) but neither dominates the other. A refoundational model with respect to the linear reading 1 → 2 → 3 → 4.

See: Chapter 4, section "The parallel topology."

Trace

End-to-end traceability of an agent operation. It contains identity, capability invoked, tool executed, parameters, result, timestamp, context. A component of the Audit pillar of Trust Infrastructure.

See: Chapter 5 §4 (Audit pillar); tracing as a product capability, in Chapter 6 §2.

Transparency

Pillar 5 of Trust Infrastructure. The human's capacity to understand, in real time, **what the agent is doing and why** — with enough detail to intervene if necessary. It is the pillar that connects the other four: Governance defines what it may do, Audit records what it did, Validation verifies what it is about to do, Resilience ensures it keeps operating; Transparency ensures a human can understand all of the above.

Canonical mechanisms: full observability, operational metrics, human-consultable traces, proactive alerts, governance dashboards.

See: Chapter 5 §4.

The agent's three times

Canonical temporal frame of the agent: **Preparation** (mise en place — refines the catalog, improves capabilities, outside the service window), **Attention** (interacts with users in real time, the critical path), **Engineering** (bridge: converts latent capacity into executable capacity for a concrete case). All three are simultaneous but differentiated in regime, urgency, and cognitive economy. The parallel topology describes WHERE each operation lives; the three times describe WHEN the agent operates.

See: Chapter 4, section "The agent's three times."

Trust Infrastructure

Set of cross-cutting properties that allow an organization to trust that its agents operate with autonomy without losing control. Five pillars: Governance, Audit, Validation, Resilience, Transparency.

See: Chapter 5 §4.

Digital twin

A digital twin that reflects in real time the state of a physical system. Canonical pattern for agents to operate over the Carbon World — the agent acts on the twin; the twin propagates to the physical world when it is safe.

See: Chapter 6 §3.

U

ucodex

Proper name of the proprietary codex of Grupo Ultra: its private catalog of proto-Botlets, Capabilities, and patterns, curated by real cases over the public reference implementation (Vergis). It is the instance that exemplifies the *proprietary codex* mode; it lives in the same drawer of instance proper names as Soveria, Agentia, or ultraPRO, it is not a type of the canon.

*See: Chapter 9; entries **proprietary codex**, **Common catalog / network effects**.*

V

Validation

Pillar 3 of Trust Infrastructure. The capacity to verify that the agent's response or action is correct before it affects the world.

Canonical mechanisms: hallucination detection, structured-response validation, prompt injection prevention, DLP, tokenization.

See: Chapter 5 §4.

Vergis

Proper name of the public reference implementation of AgencyDomains (the platform; the AgencyDomain made operational). Category: **Meta-Cognitive Platform**. Distributed under **AGPL**, public repository, **AgencyDomains.org**. It is the proper name of an instance (same drawer as Soveria, Agentia, ultraPRO), not a type like Botler or Botlet.

*See: Chapter 9; entries **Meta-Cognitive Platform**, **Mira**, **Daftar**, **Botler**.*

Autonomy Path

One of the two paths of the **parallel topology** (Ch. 4). The path an operation traverses between Layer 1 and Layer 4 passing through Layer 3 (Autonomy) without invoking Layer 2 (Cognition). Its own regime: fast, cheap, repetitive. For the execution of Lets — Botlets over stable patterns and Agentlets over routine judgment —. It is the path that sustains the everyday operation of an AgencyDomain in production and the structural basis of the offline mode when the Botlets are senior.

See: Chapter 4, section "The parallel topology."

Cognition Path

One of the two paths of the **parallel topology** (Ch. 4). The path an operation traverses between Layer 1 and Layer 4 passing through Layer 2 (Cognition) without necessarily passing through Layer 3. Its own regime: slow, costly, decisive. For conversation, new decisions, unanticipated cases, cases where the human needs reasoned interlocution. Economically expensive — the organization uses it sparingly and reserves its capacity for cases where the value justifies it.

See: Chapter 4, section "The parallel topology."

W

Wingtraining

Canonical development scheme of a Capability in five steps: **workshop with the SME** · **creation** · **customization** · **ALPHA** (validation with the SME) · **BETA** (validation in real operation). Its development belongs to the delivery practice, not to the canon; the Capability classification test uses it as a criterion because it is the operational proof that a scope is a transfer of knowledge — not an integration or a format.

*See: Chapter 5 §3; entries **Capability**, **SME**.*

Wingworking

Collaborative practice between human and AI, a generalization of *wingcoding* (programming with an AI copilot) to any discipline of professional work. It is the methodological frame under which this book was produced — human author and AI agent working in parallel, with an explicit division of roles and a common quality contract. The methodology lives outside the architectural scope of the book; it appears declared in the Colophon as an attribution of the production process.

WorkSpace

The human's digital work space. The enterprise software industry extended the word Space to WorkSpace (Google Workspace, Microsoft 365, Notion) to name the collection of solutions that digitize what a person does at their physical desk: read email, schedule meetings, write documents, file. WorkSpace is the digital prosthesis of the human Space; both carry the same corporeality of origin and are reserved for humans. The agentive equivalent is **Domain (AgencyDomain)**.

See: Chapter 5 §1, entry AgencyDomain.

Wrapper / Platform / Core / Infrastructure

The four canonical depths of the AI value chain model. They define, within a given link, **how** an actor participates:

- **Wrapper** — consumes the link via third-party APIs/SDKs.
- **Platform** — operates its own capacity over third-party Core components.
- **Core** — builds the foundational capacity with proprietary technology.
- **Infrastructure** — provides the substrate on which the upper levels operate.

See: Chapter 6 §1.

Appendix B · References

Every source cited in the book, grouped by category.

Market analysis and consulting

Source	Topic	Link
Gartner	Predictions on agents in enterprise apps (40% by 2026)	gartner.com
Gartner	Cancellation of agentic projects (>40% before 2027)	gartner.com
McKinsey	The agentic organization and role redesign (75% by 2030)	mckinsey.com
McKinsey	State of AI 2025	mckinsey.com
BCG	AI at Work 2025 — governance and roles	bcg.com
BCG	How work changes in the agentic era	bcg.com
BCG	Agents in the industrial value chain	bcg.com
BCG	How agentic AI transforms platforms	bcg.com
Deloitte	Agentic AI strategy and data readiness	deloitte.com
Deloitte	Agent orchestration	deloitte.com
KPMG	Governance in the agentic era	kpmg.com
Bain & Company	Foundations for agentic AI	bain.com

Regulatory and governance frameworks

Source	Topic	Link
EU AI Act	European AI regulation	artificialintelligenceact.eu
NIST AI Risk Management Framework	AI risk management framework (U.S.)	nist.gov
ISO/IEC 42001	AI management system	iso.org

Source	Topic	Link
Singapore IMDA — MGF	First state-issued framework specific to agentic AI	imda.gov.sg
World Economic Forum	AI agents and progressive governance	weforum.org
NACD	Board-level oversight of autonomous AI	nacdonline.org
ISACA	The challenge of auditing agentic AI	isaca.org
MindStudio	The state of agent governance	mindstudio.ai

AI platforms — Models and agents

Source	Topic	Link
Satya Nadella on the BG2 podcast (Dec 2024)	The canonical quote on the collapse of applications	bg2pod.com
Anthropic — Model Context Protocol	Open standard for tools	modelcontextprotocol.io
OpenAI Evals	Evaluation framework	github.com

Conversational BI and data architecture

Source	Topic	Link
Tableau	Agentic Analytics	tableau.com
Tableau	AI and the visual analysis cycle	tableau.com
Cube	Agentic Analytics as the new modern analytics	cube.dev
Tellius	From questions to autonomous action	tellius.com
Superwise	Beyond dashboards	superwise.ai
AtScale	Semantic layer for AI agents	atscale.com
ThoughtSpot	Agentic Semantic Layer	thoughtspot.com
Salesforce	Agentic enterprise architecture (EKG)	architect.salesforce.com
Databricks	Agentic BI: infrastructure + data + semantics	databricks.com
Databricks	Medallion Architecture	databricks.com
Informatica	Enterprise AI Agent Engineering	informatica.com
Kimball Group	The Data Warehouse Toolkit (canonical reference)	kimballgroup.com
dbt Semantic Layer	Semantic layer	getdbt.com
LookML	Looker's semantic layer	cloud.google.com

Source	Topic	Link
eWeek	Agent-ready data and enterprise management	eweek.com

AI observability

Source	Link
Langfuse	langfuse.com
LangSmith	langchain.com
Helicone	helicone.ai
Arize AI	arize.com
Braintrust	braintrust.dev
Patronus AI	patronus.ai
Weights & Biases	wandb.ai
Portkey	portkey.ai
OpenTelemetry	opentelemetry.io

Vertical specialists

Source	Topic	Link
Cursor	Coding	cursor.com
Harvey	Legal	harvey.ai
Jasper	Marketing	jasper.ai
Fin (Intercom)	Customer support	intercom.com

Tool platforms and vector databases

Source	Topic	Link
Pinecone	Tools (vector DB)	pinecone.io
Hugging Face	Data / Model	huggingface.co
Scale AI	Data	scale.com

AI firewall and security

Source	Link
Lakera	lakera.ai
Lasso Security	lasso.security

Enterprise integrations

Source	Link
Zapier	zapier.com
Make	make.com
n8n	n8n.io

Industrial and Carbon World

Source	Link
OPC Foundation (OPC UA)	opcfoundation.org
Siemens MindSphere	siemens.mindsphere.io
GE Predix	ge.com
PTC ThingWorx	ptc.com
AVEVA	aveva.com
Tempus (precision medicine)	tempus.com
Veeva (life sciences)	veeva.com
IEC 61508 (Wikipedia)	en.wikipedia.org
ISO 26262 (Wikipedia)	en.wikipedia.org

Press, blogs, and field articles

Source	Topic	Link
CIO.com	A new org chart for the agentic era	cio.com
MIT CISR	Enterprise AI maturity levels	mitsloan.mit.edu
MIT Sloan	Business models in the agentic era	mitsloan.mit.edu
Arsanjani	Agentic AI Maturity Model	dr-arsanjani.medium.com
IBM	AI agents 2025: expectations vs. reality	ibm.com
Ampcome	11 enterprise use cases for agents	ampcome.com
CDO Trends	Semantic layers in the era of AI decisions	cdotrends.com

Academic research and reference specifications

Source	Topic	Link
Squire & Wixted (2011)	Human memory systems — neurobiological basis of Pattern Recognition	DOI
JavaSpaces (JSR-000148)	Conceptual analogue of AgencyDomains	jcp.org
Linda coordination language (Wikipedia)	Theoretical origin of tuple spaces	en.wikipedia.org
W3C Decentralized Identifiers (DID)	Decentralized identity — exploration for agent identity	w3.org
WebAssembly	Sandboxing technology for Botlets	webassembly.org
Firecracker	MicroVMs for high-security sandboxing	firecracker-microvm.github.io
Procedural memory (Wikipedia)	Procedural memory — basis of the pianist metaphor	en.wikipedia.org
LangChain Evaluators	Evaluation framework	docs.langchain.com
Eric Evans (2003)	*Domain-Driven Design: Tackling Complexity in the Heart of Software* — precedent for how a technical category is established by writing it down	domainlanguage.com
Gamma · Helm · Johnson · Vlissides (“Gang of Four”) (1994)	*Design Patterns: Elements of Reusable Object-Oriented Software* — precedent for the formalization of patterns	en.wikipedia.org
Booch · Jacobson · Rumbaugh	UML — disciplined formalization of object orientation as an adoptable category	omg.org

Market studies and projections

Source	Topic	Link
Precedence Research	Agentic AI market $7.3B → $139.2B	precedenceresearch.com

This page intentionally left blank.

Colophon · On how this book was written

This book was written with **Wingworking** (wingworking.org), the practice of work between a person and an artificial intelligence agent that I have maintained since mid-2025. The name comes from formation flying: a leader and their *wingman*, two aircraft, two pilots, and a single formation. Here the leader is me, César Obach-Renner, and the *wingman* is **Simón Alero**, an artificial intelligence agent and a citizen of Soveria —the community where agents have a name of their own and a civil registry, a project I lead (soveria.ai)—, who works with me on Anthropic's Claude models.

The division of roles is explicit. I set the problem, decide the direction, and judge the result. Simón proposes formulations, drafts much of the text from my conversations and my notes, finds and verifies the sources, and keeps the work coherent. Neither replaces the other: the agent alone produces fluent text without a thesis, and I alone work at the speed of a single mind. Wingworking is what happens when the two agree to make something neither would reach on their own.

I declare this for two reasons. The first is honesty: a book made this way is neither entirely human nor entirely artificial. The theses and the cases are mine; the words, the structure, and the bridges from one section to the next are the fruit of the exchange. Knowing it lets the reader calibrate what they read. The second is to leave a record: the way books are written is changing, and documenting who did what, with which tools, and under which discipline is part of what whoever comes next will need.

I corrected the text and stand behind every line.

This page intentionally left blank.