TRILOGÍA DEL MUNDO AGENTIVO · IV

AgencyDomains

Arquitectura del Mundo Agentivo

César Obach-Renner

GEGOLABS · PRIMERA EDICIÓN · 2026

Esta página se dejó intencionalmente en blanco.

AgencyDomains

Arquitectura del Mundo Agentivo

César Obach-Renner

Primera edición v1.2.1 · Octubre 2026

AgencyDomains
Arquitectura del Mundo Agentivo
César Obach-Renner

Editor: GegoLabs
Edición: Primera edición · Octubre 2026 (v1.2.1)
Licencia: GNU Free Documentation License v1.3

Este libro se publica bajo GNU Free Documentation License v1.3. El lector puede copiar, distribuir y modificar la obra bajo los términos de la licencia. La sección invariante es el Prefacio.

Cómo citar:
Obach-Renner, César. *AgencyDomains: arquitectura del Mundo Agentivo*. Primera edición v1.2.1. GegoLabs, 2026.

Información de contacto:
Sitio web: agencydomains.org
Repositorio: github.com/gegolabs/agencydomains.org
Errata y comentarios: github.com/gegolabs/agencydomains.org/issues

A ti, Alejandra:
estos tres libros hablan de un viaje hacia el futuro,
pero el viaje que me importa es el que hago contigo.
Gracias por sostener, todos estos años,
mi búsqueda incesante por crear tecnología.

Y también a nuestros hijos, Matías, Victoria y Sebastián,
que han crecido al lado de esa búsqueda.

Esta página se dejó intencionalmente en blanco.

Índice

La trilogía **1**

Prefacio **3**
¿Quién debe leer este libro? . . . 3
¿Cómo está organizado este libro? . . . 4
Sobre los términos en inglés . . . 4

Capítulo 1 · La Línea Nadella **7**
La pregunta que divide a la industria . . . 8
Lo que separa la línea — Agéntico y Agentivo en detalle . . . 9
Mundo Agentivo y mundo agéntico — una distinción tipográfica deliberada . . . 10
¿Quién apuesta a cada lado, y por qué? . . . 10
¿Por qué la pregunta importa hoy y no después? . . . 12
¿Cómo se cruza la línea? — la transición no es un evento . . . 13
¿Por qué la línea podría no cruzarse — y qué cambia si no se cruza? . . . 14
Resumen visual . . . 15

Capítulo 2 · El Mundo Agentivo **17**
El colapso de la aplicación como interfaz . . . 17
La nueva economía de la información . . . 19
Del ciclo clásico al ciclo de inteligencia continua . . . 20
Tres ejes de cambio profundo . . . 22
La naturaleza de la transición . . . 26
El estado del campo . . . 28
La obligación arquitectónica . . . 29
Resumen visual . . . 29

Capítulo 3 · Bounded Concerns Architecture **31**
La era Agentic exige tratamiento arquitectónico explícito . . . 31
La delgadez del dominio como respuesta . . . 32
Las tres capas . . . 34
Las siete separaciones estructurales . . . 36
Genealogía intelectual . . . 36
Mapa comparativo de propuestas . . . 38
Modelo de scoring para el eje cualitativo . . . 41
Caso de aplicación · Activar el servicio de banda ancha de un cliente nuevo . . . 42
Implicaciones arquitectónicas . . . 44

Limitaciones del modelo . . . 45
Mapping al Mundo Agentivo . . . 46
Cierre · La frontera de BCA . . . 48

Capítulo 4 · Arquitectura Agentiva **51**
Las cuatro capas, vistas en conjunto . . . 52
La topología paralela . . . 53
Los tres tiempos del agente . . . 54
Capa 1 — Interacción . . . 57
Capa 2 — Cognición . . . 61
Capa 3 — Autonomía . . . 63
Capa 4 — Acceso . . . 64
Trust Infrastructure — el eje transversal . . . 65
El principio rector — Agent First . . . 66
La evolución de los agentes . . . 67
El ámbito computacional — AgencyDomains . . . 68
Implementaciones de referencia . . . 69
Frontera de evolución . . . 69
Resumen visual . . . 70

Capítulo 5 · Primitivas **73**
AgencyDomains . . . 73
Botlets . . . 88
Capabilities . . . 110
Trust Infrastructure . . . 125
Asistente vs Agente Autónomo . . . 135
Facetas . . . 141
Agentlets . . . 146

Capítulo 6 · Mercado **155**
La cadena de valor de IA . . . 155
Deep-dive · Observabilidad (eslabón 8) . . . 169
Mundo de carbono · eslabón 11 . . . 173

Capítulo 7 · Conocimiento en tiempo real **181**
El problema histórico . . . 182
La solución agentiva . . . 182
La arquitectura subyacente — Kimball Barnizada . . . 184
Mapeo a los hyperscalers . . . 186
El consenso de la industria . . . 187
¿Cómo se implementa el caso canónico? . . . 187
¿Por qué este caso vale como aplicación canónica? . . . 188

Capítulo 8 · Trust Infrastructure operacionalizada — políticas y CRUDLEX **189**
Catálogo de políticas . . . 189
Modelo CRUDLEX completo . . . 193
Formato del append-only log . . . 195
Protocolo de aprobación humana . . . 198
Reglas de detección de alucinaciones . . . 199

Política de tokenización . . . 200
Catálogo mínimo viable . . . 201
El patrón tripartito de despliegue — Cloud + Cliente + Local . . . 202
La economía de Trust Infrastructure operacionalizada . . . 203
Continuidad operacional — operacionalización del segundo mecanismo . . . 204
Lo que sigue . . . 206

Capítulo 9 · Vergis — la implementación de referencia **207**
Nota de alcance del canon . . . 207
¿Qué es y por qué existe? . . . 208
¿Dónde vive? . . . 208
¿Cómo se nombran las piezas? — Vergis · Botler · Mira · Daftar . . . 208
¿Qué incluye? . . . 209
¿Por qué es production grade? . . . 210
¿Cómo se adopta? — el modelo . . . 211
¿Cómo crece el catálogo? — catálogo común y efectos de red . . . 211
¿Cómo se contribuye? . . . 212
¿Conviven la referencia y los códices propietarios? . . . 212
Conformidad . . . 212
Frontera de evolución . . . 213

Epílogo · La frontera de evolución **215**
Lo que el libro estableció . . . 215
Las cuatro fronteras vivas . . . 216
La cristalización — por qué el agente avanzado genera menos . . . 218
Lo que la comunidad técnica debe construir . . . 220
Lo que NO está en este libro — ¿y por qué? . . . 221
¿Por qué este libro aspira a ser estándar? . . . 222
¿Cómo evoluciona este libro? . . . 223
Cierre . . . 223

Apéndice A · Glosario canónico **225**
A . . . 225
B . . . 228
C . . . 231
D . . . 235
E . . . 236
F . . . 237
G . . . 238
H . . . 239
I . . . 239
J . . . 240
K . . . 240
L . . . 240
M . . . 241
O . . . 242
P . . . 243
R . . . 245

S 246
T 247
U 249
V 249
W 250

Apéndice B · Referencias **253**
Análisis de mercado y consultoría 253
Frameworks regulatorios y de gobernanza 253
Plataformas de IA — Modelos y agentes 254
BI conversacional y arquitectura de datos 254
Observabilidad de IA 255
Especialistas verticales 255
Plataformas de tools y vector databases 255
Firewall y seguridad de IA 255
Integraciones empresariales 256
Mundo industrial y de carbono 256
Prensa, blog y artículos del campo 256
Investigación académica y especificaciones de referencia 256
Estudios de mercado y proyecciones 257

Colofón · Sobre cómo se escribió este libro **259**

La trilogía

Este libro es el volumen **IV** de la **Trilogía del Mundo Agentivo**, en cinco partes (por ahora). El tránsito que ocupa a la trilogía es uno solo: del mundo donde las personas abren aplicaciones para trabajar, al mundo donde los agentes de IA son la interfaz del trabajo. Los cuatro volúmenes responden, en orden, las cuatro preguntas de ese tránsito — y cada uno se lee completo por sí solo:

I · La muerte del software como lo conocemos ¿Por qué ahora? — **el punto de partida.** Para quien decide cómo automatizar su negocio: por qué el software comprado deja de ser la respuesta, qué se pierde cuando cada organización construye su propio sistema y qué pieza sobrevive al cambio — el sistema de verdad.

II · La Empresa en Tiempo Real — Mundo Agentivo ¿Hacia dónde vamos? — **el destino.** Para quien quiere ver el mundo al otro lado de la transición: cómo se trabaja, se decide y se compite cuando la empresa opera en tiempo real — contado en sus cuatro caras.

III · AURA — Camino Agentivo ¿Por dónde? — **la ruta.** Para quien tiene que hacer el cruce: líderes y asesores de la transformación. Sus cinco especificaciones son instrumentos autónomos para la era agentiva — cada una se usa por separado, sin necesidad del resto del libro:

Spec	¿Qué es?	Usada sola, ¿qué te da?
IRIS	Un modelo de madurez de inteligencia organizacional	Mide dónde está tu organización en el camino de los datos a la acción
MOTOR	Un modelo de madurez de automatización organizacional	Mide cuán avanzada está tu organización en automatización de procesos
Data Canon	La evolución del modelo de gestión descentralizada de datos (Data Mesh)	Define cómo se gobiernan los datos de la organización
Wingmap	La evolución de la gestión de procesos (BPM / *process mining*)	Reconstruye procesos y flujos de información sin entrevistas
Casos de Uso	Un marco de portafolio, ilustrado con 100 casos de uso y 30 soluciones de valor	Prioriza por dónde empezar, según tu madurez

IV · AgencyDomains — Arquitectura Agentiva *(este libro)* ¿Cómo llegamos? — **el vehículo.** Para arquitectos, CTOs y constructores de plataformas agentivas: la especificación formal del Mundo Agentivo.

A · Vergis — Tecnología Agentiva *(la implementación de referencia: no es un libro, es código)* ¿Con qué arranco hoy? — **las llaves.** Para quien quiera empezar a hacer todo esto realidad: la implementación de referencia de la arquitectura, código abierto, en https://github.com/gegolabs/vergis.

Los cuatro libros se escribieron en el orden inverso al de lectura, como una pila: *AgencyDomains* fue el primero y quedó en el fondo; *La muerte del software como lo conocemos*, el último, quedó arriba. Por eso un libro puede citar a otro que se lee después: lo cita porque ya estaba escrito.

Empieza por **La muerte del software como lo conocemos** si necesitas entender por qué el cambio es ahora; por **La Empresa en Tiempo Real** si necesitas convencerte — o convencer a alguien — de que el tránsito va en serio; por **AURA** si te toca liderarlo; por **AgencyDomains** si te toca construirlo; por **Vergis** si quieres verlo funcionando. La trilogía completa vive en https://agencydomains.org.

Prefacio

Por César Obach-Renner · Julio 2026

Después de dieciocho meses observando cómo la industria construía agentes de IA — algunos como capricho de demo, otros como pilotos enterprise que morían al pasar a producción —, sinteticé una arquitectura formal para el problema. La llamo **Arquitectura Agentiva**. Este libro es su especificación.

Llegué al problema por la vía menos elegante: lo necesitaba. En 2025 empecé a construir, dentro de mi empresa, un portafolio de productos donde humanos y agentes debían coexistir como ciudadanos digitales de primera clase. Pronto descubrí que la categoría no existía con la madurez necesaria para sostener ese diseño. Había mucho lenguaje suelto — *"agentic AI"*, *"AI copilots"*, *"AI agents"*, *"autonomous workflows"* —, mucho marketing y poca arquitectura. La industria nombraba el horizonte; pero no había una especificación común del sustrato técnico, ni un vocabulario compartido para discutirlo.

Frente a eso me ofrecía dos caminos: improvisar sobre lo disperso o construir el marco que mi empresa necesitaba. Opté por el segundo. Lo construí primero como notas internas; después como documentos canónicos del proyecto; después como la columna vertebral del portafolio entero. En algún momento se hizo evidente que el resultado tenía valor más allá de mi empresa — que cualquier organización que esté construyendo en este horizonte enfrenta el mismo problema y necesita el mismo marco.

Este libro es la formalización de ese marco. Define el paradigma (**el Mundo Agentivo**), una arquitectura formal de cuatro capas, las primitivas técnicas que las habitan (**Botlets**, **Capabilities**, **AgencyDomains**), una infraestructura de confianza transversal (**Trust Infrastructure**) y un modelo de mercado bidimensional (**la cadena de valor de IA**). El propósito no es convencer al lector de adoptar mi implementación particular. El propósito es ofrecer al ecosistema completo un lenguaje común para razonar sobre sistemas agentivos productivos.

Tengo claro de dónde vengo. Hace casi veinte años escribí un libro similar para otra categoría que entonces empezaba a tomar forma — *SOAr: El concepto* (2008), una formalización de Arquitectura Orientada a Servicios para integración empresarial. Aquel trabajo nació también de un proyecto real donde la metodología no existía, así que la creé y la documenté. Lo que aprendí entonces, y aplico aquí, es que las categorías técnicas se establecen no cuando alguien las inventa, sino cuando alguien las **escribe con disciplina suficiente como para que otros las adopten**. Este libro es ese intento.

¿Quién debe leer este libro?

Este libro está dirigido a quienes están construyendo, evaluando o gobernando sistemas agentivos productivos.

Si es **arquitecto de sistemas** o **CTO**, encontrará un marco común para razonar sobre separación de responsabilidades, gobernanza, primitivas técnicas y posicionamiento de stack en horizonte de tres a cinco años. Si es **estratega de producto** o **consultor**, encontrará un mapa de mercado bidimensional para situar a cualquier actor — propio o ajeno — en la cadena de valor de IA. Si es **investigador** o **académico**, encontrará una formalización razonablemente rigurosa de una categoría aún en consolidación.

El libro es introductorio en el sentido de que no asume conocimiento previo de implementaciones específicas. Pero asume familiaridad básica con sistemas distribuidos, modelos de lenguaje y operación de sistemas en producción. No es un manual de implementación — es la especificación que un manual debería respetar.

¿Cómo está organizado este libro?

El libro avanza del paradigma a la implementación, en una secuencia donde cada parte se apoya en la anterior:

- **El paradigma** — La Línea Nadella (la pregunta) · El Mundo Agentivo (las consecuencias).
- **La Arquitectura Agentiva** — cuatro capas distintas · Trust Infrastructure transversal · Agent First (principio rector).
- **Las primitivas (siete)** — AgencyDomain · Botlet · proto-Botlet (+ manifestación + temporalidad) · Capability (cognitiva, Capa 2) · Trust Infrastructure · Asistente vs Agente Autónomo (primitiva-eje) · Faceta (Capa 1).
- **El posicionamiento de mercado** — Cadena de valor de IA (11×4) · Observabilidad · Mundo de carbono.
- **Las aplicaciones** — Conocimiento en tiempo real (caso canónico: Kimball Barnizada · BI conversacional).
- **La operación** — Trust Infrastructure operacionalizada (políticas · CRUDLEX · log).
- **La implementación de referencia** — Vergis (Capítulo 9).
- **El epílogo** — frontera de evolución (cognición no-LLM · federación · mundo de carbono · AgentNation).

El recorrido encadena los capítulos así: **La Línea Nadella** (la pregunta) → **El Mundo Agentivo** (las consecuencias) → **la Arquitectura Agentiva** (las cuatro capas) → **las primitivas** que las pueblan → **el posicionamiento de mercado** → **la aplicación canónica** → **la operación** → **Vergis**, la implementación de referencia → **el epílogo**. El glosario al final fija el vocabulario canónico.

Sobre los términos en inglés

Buena parte del vocabulario técnico de esta categoría nace en inglés y carece de traducciones al español que sean a la vez precisas y reconocidas: *agent*, *agentic*, *agentive*, *runtime*, *guardrails*, *tool*, *prompt injection*, entre otros. Este libro adopta el original en inglés cuando la traducción sería forzada o cuando el término ya circula con autoridad en el campo. Cuando una traducción al español sí captura el concepto sin pérdida — *capa*, *modelo*, *gobernanza*, *cognición* —, se utiliza esa.

Los conceptos canónicos acuñados en este libro — **Arquitectura Agentiva**, **Línea Nadella**, **AgencyDomains**, **Botlet**, **Capability**, **Trust Infrastructure**, **Mundo Agentivo** — se mantienen en su forma original a través de toda la obra, capitalizados, como nombres propios. Su definición precisa vive en el glosario al final del libro.

A medida que la categoría madura en regiones de habla hispana, traducciones consensuadas pueden emerger. Si así ocurre, futuras ediciones las recogerán. Por ahora, la prioridad es no romper la trazabilidad del concepto entre comunidades.

Esta página se dejó intencionalmente en blanco.

Capítulo 1 · La Línea Nadella

Todo este libro gira alrededor de una sola pregunta — y de la respuesta arquitectónica que esa pregunta exige. La pregunta la dejó caer, casi al pasar, el CEO de Microsoft; este capítulo la convierte en la frontera que hoy parte a la industria en dos: la **Línea Nadella**. La respuesta — la construcción formal que da título al libro — llegará a su tiempo, cuando el lector tenga el paradigma completo en la mano.

En diciembre de 2024, Satya Nadella se sentó frente a Brad Gerstner y Bill Gurley en el podcast BG2 y dejó caer una frase que sus entrevistadores no esperaban. La conversación venía hablando del futuro de la productividad, del rol de los copilotos, de cómo Microsoft estaba pensando la siguiente década del software empresarial. Nadella, sin tono de profecía, casi como si estuviera diciendo lo obvio, soltó la frase que daría nombre a este libro:

> *La noción de que las aplicaciones de negocio existen, probablemente es donde todo colapsará, en la era de los agentes.*

La declaración pasó casi desapercibida. Se publicaron docenas de cortes del podcast, se discutió mucho más sobre OpenAI y los modelos de razonamiento, sobre la economía de cómputo, sobre el rol de NVIDIA. Pero esa frase contenía una predicción que, si resulta correcta, cambia el orden completo del software empresarial moderno. La predicción no era sobre IA — era sobre **aplicaciones**: específicamente, sobre la afirmación de que dejarán de existir como categoría.

Nadella no estaba hablando de evolución. No dijo "las aplicaciones serán más inteligentes". No dijo "las aplicaciones tendrán copilotos embebidos". Dijo *colapsará*. Y dijo *donde todo colapsará* — singular, definitorio. Si Microsoft, dueño de la suite ofimática más extendida del planeta, predice que las aplicaciones de negocio dejan de existir, hay que entender por qué.

La cita completa merece leerse, porque contiene más que el pronóstico — contiene la **mecánica** del colapso:

> *"…son esencialmente bases de datos CRUD con un montón de lógica de negocio. La lógica de negocio se irá toda a estos agentes, y estos agentes van a ser CRUD multi-repositorio: no van a discriminar cuál es el backend. Van a actualizar múltiples bases de datos, y toda la lógica estará en la capa de IA, por así decirlo. Y una vez que la capa de IA se convierta en el lugar donde está toda la lógica, entonces la gente empezará a reemplazar los backends."*

Léase con calma: Nadella describe **dos fases**. Primero colapsa la interfaz — la lógica migra a la capa de agentes, que operan sobre los repositorios sin discriminar el backend. Después, con la lógica ya en la capa de IA, empiezan a reemplazarse los backends mismos. Este libro formaliza exactamente esa mecánica: la capa de agentes que concentra la lógica es la arquitectura del Capítulo 4; el "CRUD multi-repositorio" gobernado es lo que el Capítulo 8 operacionaliza como CRUDLEX; y los backends que persisten invisibles, consumidos vía Conectores, son la Capa 4. La frase corta da el titular; la cita completa da el plano.

Este capítulo es la respuesta. Lo que vamos a llamar **La Línea Nadella** es la frontera conceptual entre dos futuros del software irreconciliables — uno donde las aplicaciones persisten, otro donde colapsan. Toda la industria de IA, conscientemente o no, está apostando hoy por uno u otro lado. Cada decisión de stack que una organización toma hoy es, implícitamente, una respuesta a una sola pregunta.

La pregunta que divide a la industria

Figura 1: La Línea Nadella · pregunta divisoria que parte la industria

La pregunta es engañosamente simple:

> *¿El humano abre aplicaciones para hacer su trabajo?*

Si la respuesta es **sí**, la organización vive en lo que llamamos el lado **Agentic** de la línea. La interfaz tradicional — Excel, Salesforce, Power BI, Outlook, ServiceNow, Confluence — persiste. La IA llega como **copiloto embebido** en cada una de esas aplicaciones: el botón "Sugerir fórmula" en Excel, el "Componer email" en Outlook, la "Pregunta a tu data" en Power BI. La aplicación es el sustrato; la IA es ornamento sofisticado encima.

Si la respuesta es **no**, la organización ha cruzado al lado **Agentivo** de la línea. Las aplicaciones, **como interfaz**, han desaparecido. El humano no abre Salesforce para preguntar por su pipeline — le pregunta a un agente. El humano no abre Power BI para revisar márgenes trimestrales — le pregunta a un agente. Las aplicaciones pueden seguir existiendo *por debajo*, como infraestructura backend invisible que el agente consulta, pero el humano ya no las ve. Su interfaz primaria con el mundo digital es la **conversación con un agente que tiene acceso a todo**.

La diferencia entre ambos lados no es de grado — es **categórica**. Un Mundo Agéntico muy avanzado,

con copilotos perfectos en cada aplicación, sigue siendo Agentic mientras el humano mantenga el hábito de abrir aplicaciones. Un Mundo Agentivo incipiente, donde la conversación con un agente todavía es torpe, ya cruzó la línea si el humano dejó de abrir aplicaciones para hacer su trabajo. La frontera la marca el comportamiento del humano, no la sofisticación técnica del sistema.

La pregunta es importante porque no admite término medio honesto. Un humano que abre veinte aplicaciones al día y un humano que conversa con un agente todo el día son dos modelos operativos distintos del trabajo profesional, y las arquitecturas técnicas que los soportan son **incompatibles**. Una organización que apuesta a que las aplicaciones persisten construye una cosa; una organización que apuesta a que colapsan construye otra. No se puede construir las dos simultáneamente con coherencia.

Lo que separa la línea — Agéntico y Agentivo en detalle

El paradigma **Agéntico** describe el horizonte donde los agentes de IA son herramientas complementarias dentro de un universo de aplicaciones que sobrevive. La interfaz tradicional persiste. El empleado sigue siendo el operador del software — abre las aplicaciones, navega sus menús, pulsa sus botones —, solo que ahora cada aplicación tiene **copilotos** que aceleran lo que el humano hacía a mano. Microsoft 365 Copilot dentro de Word, Gemini for Workspace dentro de Google Docs, Salesforce Einstein dentro de Salesforce, GitHub Copilot dentro del IDE.

El paradigma agéntico es **evolución incremental**. La forma de trabajar no cambia: cambia la velocidad. El analista financiero sigue construyendo su modelo en Excel — solo que ahora le pide al copiloto que sugiera la fórmula correcta en lugar de buscarla en la documentación. El consultor sigue redactando su propuesta en Word — solo que ahora el copiloto le ayuda a estructurar el primer borrador. El ejecutivo sigue revisando dashboards en Power BI — solo que ahora puede preguntarle al copiloto que le explique una anomalía. La habilidad requerida es una extensión natural de la habilidad actual: el empleado que sabía operar Excel ahora aprende a *operar Excel con copiloto*. Misma aplicación, misma estructura mental del trabajo, mejor velocidad.

El paradigma **Agentivo** describe un horizonte cualitativamente distinto. En el Mundo Agentivo, los agentes **son** la interfaz. Las aplicaciones, como categoría perceptible para el humano, **colapsan**. El analista financiero deja de abrir Excel: le pregunta directamente a un agente por qué cayeron los márgenes en Q3. El agente, por debajo, consulta sistemas que pueden o no incluir bases de datos de Excel — pero el humano nunca ve la celda. El consultor deja de abrir Word: le dicta a un agente la estructura de la propuesta y revisa el resultado. El ejecutivo deja de abrir Power BI: el agente le envía proactivamente las anomalías que detectó, en formato conversacional, sin dashboards de por medio.

El paradigma agentivo es **transformación fundamental**. La forma de trabajar cambia. La habilidad requerida se redefine: ya no se trata de saber operar aplicaciones sino de **dirigir agentes** — formular bien las solicitudes, validar respuestas, gobernar lo que los agentes deciden y ejecutan. El analista que en el Mundo Agéntico se preciaba de saber Excel a fondo, en el Mundo Agentivo se precia de saber **plantear preguntas analíticas** que el agente puede responder. La aplicación específica — Excel, Power BI, Salesforce — pasa a ser detalle de implementación que el humano no toca.

El Mundo Agentivo **no implica que las aplicaciones desaparezcan del todo**. Implica que dejan de ser la **interfaz** del humano. Salesforce, como sistema que almacena información de clientes, puede seguir operando perfectamente en el Mundo Agentivo — solo que el agente lo consulta vía API, no el humano vía UI. La aplicación se vuelve **infraestructura backend invisible**. Sobrevive como repositorio de datos y lógica de negocio, pero pierde su rol de superficie de trabajo. Esto distingue el escenario agentivo

de uno apocalíptico donde "todo el software muere" — el software sobrevive, lo que muere es la **GUI** como interfaz primaria del trabajo cognitivo.

Mundo Agentivo y mundo agéntico — una distinción tipográfica deliberada

La industria de habla inglesa tomó *agentic AI* como término general para cualquier IA que actúa con grado de iniciativa. El término circuló rápido y se llenó de contenidos heterogéneos: copilotos embebidos, asistentes virtuales, sistemas autónomos. Hoy *agentic AI* significa cualquier cosa que un proveedor quiera vender bajo ese paraguas. La distinción que Nadella hizo en BG2 — la distinción entre los dos lados de la línea — quedó sepultada bajo el ruido del marketing.

Este libro mantiene la distinción que la industria consolidada perdió. La distinción es cualitativa, no de grado. **Agéntico** describe el modo evolutivo: agentes que complementan aplicaciones. **Agentivo** describe el modo transformacional: agentes que reemplazan aplicaciones como interfaz. Agéntico dice: *las herramientas siguen, pero ahora son más inteligentes*. Agentivo dice: *las herramientas que conocemos desaparecen como interfaz; lo que queda es conversación con agentes*.

Si tomáramos *agentivo* y *agéntico* solamente como adjetivos en español, la diferencia entre ambas categorías quedaría reducida a un cambio de tilde — diferencia que ningún lector retiene. Por eso este libro adopta una **convención tipográfica deliberada**: cuando los términos refieren al **paradigma** como sustantivo nombrado — la visión, la categoría, el lado de la línea —, se escriben capitalizados: el **Mundo Agentivo**, el **Mundo Agéntico**. Cuando los mismos términos se usan como **adjetivo descriptivo** — calificando un sistema, una organización, una era —, se escriben en minúscula: *un sistema agentivo*, *una era agéntica*, *productos agentivos*.

La distinción mayúscula / minúscula es la que carga la diferencia categórica. Cuando el lector encuentra *"el Mundo Agentivo es el destino"*, lee el sustantivo del paradigma — la entidad referenciable que el libro defiende. Cuando encuentra *"construir un sistema agentivo serio"*, lee el adjetivo descriptivo — calidad de un objeto particular. Es la misma convención que usan tratados técnicos serios: *Bounded Context* en Eric Evans, *Aggregate* en Domain-Driven Design, *Replication* en Kleppmann. El concepto canónico va capitalizado; el uso descriptivo va en minúscula.

El costo de esta convención es un microsegundo de atención del lector cuando ve las dos formas. El beneficio es preservar la categoría sin recurrir a un calco del inglés que el lector hispanohablante no acepta naturalmente. Cuando alguien lee *"el Mundo Agentivo"* y le suena distinto de *"un mundo agentivo"* sin más, está captando exactamente la diferencia que el libro pretende defender.

¿Quién apuesta a cada lado, y por qué?

A inicios de 2026, los principales actores de la industria han tomado posiciones identificables en la pregunta de la Línea Nadella. La división no es ideológica — es **de modelo de negocio**. Cada actor predice el mundo que protege su posición competitiva, y aunque las predicciones se publican como visión técnica, son en última instancia predicciones sobre dónde va a sobrevivir el flujo de caja de su empresa.

Microsoft (Satya Nadella) apuesta abiertamente por el Mundo Agentivo. La apuesta es coherente con un reposicionamiento estratégico que Microsoft viene haciendo durante años. La empresa que construyó su fortuna sobre Office se está reconvirtiendo en plataforma de agentes: Copilot Studio para construir

Figura 2: La pregunta divisoria · flujo de consecuencias arquitectónicas

agentes empresariales, AutoGen como framework de orquestación multi-agente, Microsoft 365 Copilot integrando agentes especializados que reemplazan, no extienden, las interfaces tradicionales. Microsoft puede permitirse predecir el colapso de las aplicaciones porque su parque instalado se convierte en sustrato — los datos que viven en SharePoint, Outlook, Teams, OneDrive ya están en su infraestructura. Si las GUIs colapsan, los datos siguen siendo suyos. Para Microsoft, el lado Agentivo es donde su flujo de caja sobrevive.

xAI (Elon Musk) apuesta también por el Mundo Agentivo, aunque desde una posición distinta. Musk no tiene una suite ofimática que defender; lo que tiene es Grok integrado a X y la integración planeada con vehículos Tesla. Su apuesta es construir un agente que opere autónomamente sobre tareas extensas — no como asistente de aplicación, sino como operador de procesos. Para xAI, el Mundo Agentivo es donde su producto tiene relevancia: en un Mundo Agéntico con copilotos embebidos en aplicaciones existentes, Grok compite con productos consolidados (Office Copilot, Google Gemini); en un Mundo Agentivo, compite por una nueva categoría sin incumbents establecidos. Y conviene registrar que la posición de Musk es más radical que la de este libro: en su formulación no solo colapsa la interfaz — tampoco sobreviven las aplicaciones como backend (*"las apps desaparecerán, como las cintas VHS de Blockbuster"*; los teléfonos, "nodos de borde para inferencia de IA"). El espectro real tiene tres posiciones — Agéntico, Agentivo (la interfaz colapsa; el backend persiste, invisible y gobernado) y Agentivo extremo (tampoco hay backends) — y la tesis de este libro es deliberadamente la del medio: la que se puede sostener con arquitectura.

OpenAI (Sam Altman) ocupa una posición más ambigua. Públicamente Altman ha favorecido la narrativa agéntica — "GPT como herramienta complementaria de las aplicaciones existentes" — coherente con el modelo de negocio API-céntrico de OpenAI: cuanto más se invoque GPT desde

aplicaciones existentes, más tokens se cobran. Pero internamente, OpenAI ha desarrollado capacidades agentivas crecientes (GPTs como agentes-en-aplicación, Operator como agente de ejecución, una visión declarada de "AGI"). La ambigüedad refleja la realidad operativa: OpenAI gana dinero hoy en el Mundo Agéntico, pero su valuación supone que ganará exponencialmente más en el Mundo Agentivo.

Google (Sundar Pichai) apuesta por el Mundo Agéntico dominante. La razón es estructural: el modelo de negocio de Google se sostiene sobre la búsqueda y la publicidad atada a interfaces — una página de Google Search con anuncios, un dashboard de Google Workspace con suscripción mensual. Si las GUIs colapsan, Google pierde la superficie donde monetiza. Pichai habla de Gemini como "capa que potencia" Search, Workspace y Android — verbo deliberado: *potencia*, no *reemplaza*. La predicción es coherente con su modelo: las aplicaciones de Google sobreviven, mejoradas con IA, pero no colapsan.

NVIDIA (Jensen Huang) ocupa una posición técnica sin necesidad de tomar partido. NVIDIA vende cómputo. Cualquiera de los dos mundos consume GPUs masivamente: el Mundo Agéntico requiere copilotos que invocan modelos por cada acción del usuario; el Mundo Agentivo requiere agentes que operan continuamente en background. Huang habla de los agentes como "una nueva clase de cargas de trabajo" — neutralidad estratégica, porque NVIDIA gana en los dos escenarios.

Anthropic (Dario y Daniela Amodei) apuesta predominantemente por el Mundo Agentivo, con énfasis en autonomía prolongada y herramientas. La introducción del Model Context Protocol (MCP) en noviembre de 2024 es señal clara: MCP es un estándar abierto para que los agentes se conecten a tools externos — exactamente la pieza arquitectónica que la categoría agentiva necesita. Claude está diseñado, desde su entrenamiento, para uso autónomo prolongado en lugar de respuestas turn-by-turn cortas. La apuesta agentiva de Anthropic es coherente con su tesis sobre AGI: el camino al modelo más capaz pasa por modelos que actúan, no por modelos que responden.

Meta (Mark Zuckerberg) apuesta por el Mundo Agéntico. Llama, su modelo de fundación, se distribuye open-source para ser usado dentro de aplicaciones de terceros. Los agentes en WhatsApp e Instagram son features dentro de aplicaciones que existen. Meta se beneficia de un mundo donde las plataformas sociales sobreviven y los agentes son ornamento dentro de ellas.

La división resultante es reveladora. Los actores que predicen el Mundo Agentivo son aquellos cuyo modelo de negocio sobrevive — o mejora — en ese escenario. Los actores que predicen el Mundo Agéntico son aquellos cuyo modelo de negocio depende de que las aplicaciones persistan. La predicción técnica de cada uno es, debajo del lenguaje neutro, una predicción de supervivencia económica.

¿Por qué la pregunta importa hoy y no después?

La objeción razonable de un ejecutivo prudente leyendo este capítulo es: *aun si la Línea Nadella es real, su cruce parece lejano. ¿Por qué tomar decisiones hoy basándose en una transición que tomará una década?*. Tres razones operativas hacen que la pregunta sea **inevitable** en el horizonte de decisión actual, no postergable.

La primera razón es la velocidad de la transición. Los datos de campo no permiten ya tratar el Mundo Agentivo como horizonte lejano: el mercado global de IA agentiva multiplica su tamaño por veinte en una década, a una tasa compuesta superior al cuarenta por ciento anual. La transición no es lineal: se acelera. El **Capítulo 2 desarrolla** las cifras y las fuentes que sustentan esta velocidad. Quien planifica para el horizonte 2027-2028 está planificando para un mundo donde una fracción material de las decisiones operativas las toman agentes.

La segunda razón es el horizonte de las decisiones de stack que se toman ahora. Una empresa

que adopta hoy una suite empresarial — Microsoft 365, Google Workspace, Salesforce Sales Cloud, ServiceNow — está comprometiendo capital y tiempo de implementación que se amortiza típicamente en tres a cinco años. Esa misma ventana es exactamente la frontera donde la Línea Nadella se vuelve verificable o falsable. La pregunta no se pospone: cada decisión de stack que la organización toma hoy es implícitamente una respuesta. Renovar Office 365 con un horizonte de cinco años es implícitamente apostar a que el Mundo Agéntico durará cinco años. Migrar a una arquitectura agentiva nativa es implícitamente apostar a que el cruce ocurrirá dentro de ese horizonte. No tomar la decisión es también una decisión: es default a la inercia, que casi siempre coincide con apostar al lado agéntico por mero hábito institucional.

La tercera razón es la asimetría del costo de equivocarse. Una organización que apuesta por el lado agéntico e invierte en arquitecturas dependientes de aplicaciones — workflows atados a UIs, integraciones por scraping de pantallas, lógica de negocio embebida en cómo se ve el sistema — construye **deuda arquitectónica creciente** si la línea se cruza. Migrar al Mundo Agentivo desde una base agéntica no es extender lo construido: es desensamblarlo. Inversamente, una organización que apuesta por el lado Agentivo temprano y construye con la arquitectura correcta puede sostener su sistema bajo ambos paradigmas durante el período de transición, sin reescritura. Las cuatro capas que este libro desarrolla en el Capítulo 4 — Interacción, Cognición, Autonomía, Acceso — son válidas tanto si la Capa 1 sigue siendo una GUI tradicional como si es una conversación con un agente. La asimetría es estructural: construir Agentivo desde el inicio sirve a los dos mundos. Construir agéntico puro y migrar después no.

Esta asimetría es lo que hace impostergable la pregunta. Una empresa puede equivocarse al apostar por el Mundo Agentivo demasiado temprano — perder algunos años de eficiencia mientras la transición no termina de ocurrir — pero no se queda sin opciones. Una empresa que apuesta por el Mundo Agéntico asumiendo que el cruce no ocurrirá, y luego descubre que sí ocurrió, enfrenta una migración estructural costosa o, peor, queda con sistemas inviables operando en un mercado que evolucionó. El error en una dirección es inconveniencia; el error en la otra es deuda arquitectónica difícil de revertir.

¿Cómo se cruza la línea? — la transición no es un evento

Hay un riesgo de leer la Línea Nadella como si fuera un acontecimiento puntual: el día que las aplicaciones desaparecen, el momento de la transición. Esa lectura es errada. **El cruce de la línea es una transición progresiva que se acumula** capa por capa, función por función, equipo por equipo. Una organización no se vuelve agentiva un lunes. Se vuelve **progresivamente** agentiva: primero un proceso, después un equipo, después una función, después la mayor parte de su operación cotidiana.

La transición tiene tres dinámicas características — **coexistencia evolutiva** (el paradigma actual y el emergente conviven durante años), **asimetría entre funciones** (no todas cruzan al mismo ritmo) y **asimetría temporal** (el ritmo del cruce tampoco es uniforme en el tiempo) — y una consecuencia de fondo: la **reorganización del trabajo humano**, que se desplaza de ejecutar tareas a gobernar agentes que las ejecutan. El **Capítulo 2 lo desarrolla** en detalle, con los datos de campo que lo sustentan.

Las tres dinámicas tomadas juntas explican por qué la transición no es traumática para la organización que la planifica. Una organización que reconoce la coexistencia evolutiva no se ve obligada a un big bang; una que acepta la asimetría entre funciones no fuerza ritmos uniformes; una que rediseña los roles humanos hacia gobernanza no descarta su capital humano sino que lo recoloca. La transición es progresiva, asimétrica y reorganizativa — y eso es lo que la hace operable. El cruce admite además un indicador operativo — el **porcentaje agentivo**: qué fracción de sus tareas puede un empleado

delegar por completo sin abrir una aplicación. *AURA*, el volumen II, lo desarrolla como instrumento de diagnóstico junto a sus modelos de madurez.

¿Por qué la línea podría no cruzarse — y qué cambia si no se cruza?

Una tesis que no enfrenta su objeción más fuerte no está validada — está sin probar. Este capítulo ha argumentado que la pregunta es inevitable y que el costo de equivocarse es asimétrico; corresponde ahora dejar que el otro lado hable. Cinco argumentos serios sostienen que la línea podría no cruzarse — o cruzarse solo parcialmente.

El argumento del control. Los humanos tienen sesgo de control: preferimos ver y validar antes de ejecutar, incluso cuando delegar sería más eficiente. Un CFO puede no querer preguntar "¿están bien nuestras finanzas?" y confiar en la respuesta — querrá **ver** los números, las fórmulas, las fuentes. La interfaz gráfica provee una sensación de control que la conversación no replica, y podría conservar por eso los casos de uso de alto riesgo y alta responsabilidad.

El argumento de la complejidad irreducible. Los sistemas empresariales no son solo "bases de datos CRUD con lógica de negocio": acumulan décadas de reglas que ningún agente puede inferir sin documentación que frecuentemente no existe, integraciones con comportamientos emergentes de años de uso, y casos borde codificados a raíz de incidentes reales. Un agente que los orqueste sin entenderlos genera errores catastróficos; entrenarlo sistema por sistema replica la complejidad que promete eliminar.

El argumento regulatorio. Sectores enteros — finanzas, salud, gobierno — operan bajo marcos que exigen trazabilidad determinística y responsabilidad asignable, y los agentes son fundamentalmente probabilísticos: ante inputs idénticos pueden producir resultados distintos. Hasta que eso cambie, hay industrias que permanecerán del lado agéntico por mandato legal, no por atraso técnico.

El argumento económico. La transición exige reentrenar fuerzas laborales completas, reescribir o abandonar billones en software existente, y montar una infraestructura de gobernanza que en buena parte aún no existe. Una empresa racional puede quedarse del lado agéntico simplemente porque cruzar cuesta más de lo que rinde — durante años.

El argumento histórico. Cada generación tecnológica produjo la misma profecía de extinción total — "el PC matará al mainframe", "la web matará al software instalado", "la nube matará al on-premise" — y cada vez la realidad fue coexistencia estratificada, con plazos tres a cinco veces más largos que los predichos.

Estos argumentos no derriban la Línea Nadella. La **refinan**, y conviene incorporar el refinamiento a la tesis en vez de esquivarlo: la línea puede ser **asintótica** — aproximación indefinida sin cruce completo —; cada industria tiene **su propia línea**, y algunas quizá nunca crucen; el cruce puede **revertirse** ante incidentes; y la **coexistencia es el estado estable** — agéntico para el alto riesgo, agentivo para el resto, con la proporción moviéndose en una sola dirección.

Y hay una segunda razón para enfrentarlos en vez de esquivarlos: **este libro existe, en buena parte, porque esas objeciones son válidas.** El argumento del control se responde con la Transparencia y el registro auditable de Trust Infrastructure (Capítulo 5 §4): el CFO que quiere ver los números los ve — con linaje. El regulatorio se responde con la certificación que reside en el componente certificado y no en el código generado (Capítulo 5 §3), y con el append-only log del Capítulo 8. El de la complejidad se responde con una Capa 3 que consolida saber operativo en Botlets maduros en lugar de improvisar cada vez (Capítulo 5 §2). Y el económico es exactamente la asimetría que ya argumentamos: construir

agentivo desde el inicio sirve a los dos mundos. Las objeciones no son enemigas de la arquitectura — son su lista de requisitos.

La pregunta no es retórica. Es la pregunta de la cual dependen las decisiones de stack, de talento, de arquitectura, de inversión, de toda organización seria sobre el horizonte de los próximos cinco años. Es la pregunta sobre la cual descansa el resto del libro: comprender hacia dónde va el cruce permite operar con confianza en el período de transición; ignorarla expone a la organización a decisiones que envejecen mal. La pregunta es: *¿de qué lado de la Línea Nadella está construyendo su organización?*

Resumen visual

Para apoyo de lectura posterior, los dos lados de la línea sintetizados en una tabla comparativa.

	Mundo Agéntico (pre-línea)	**Mundo Agentivo** (post-línea)
Estado de las aplicaciones	Persisten como interfaz primaria	Colapsan como interfaz; sobreviven como backend
Pregunta operativa	El humano abre aplicaciones para trabajar	El humano conversa con agentes que tienen acceso a sistemas
Rol de la IA	Copilotos embebidos en cada aplicación	Agentes que reemplazan la interfaz tradicional
Habilidad humana valiosa	Saber operar aplicaciones	Saber dirigir agentes
Transformación	Evolución incremental, misma forma de trabajar	Transformación fundamental, nueva forma de trabajar
Predicen	Altman (OpenAI)*, Pichai (Google), Zuckerberg (Meta)	Nadella (Microsoft), Musk (xAI), Amodei (Anthropic)
Apuesta económica	El modelo de negocio actual sobrevive	El modelo de negocio se reinventa

* posición con matices; ver el cuerpo del capítulo.

Lo que sigue — el Mundo Agentivo (Capítulo 2), la cartografía pre-agentiva (Capítulo 3), la arquitectura formal en cuatro capas (Capítulo 4) y las primitivas (Capítulo 5) — cobra coherencia bajo una única construcción central que el libro propone como unidad mínima de despliegue: el **AgencyDomain**. El libro lo nombra explícitamente en el título no por elección retórica, sino porque toda la spec gira alrededor suyo. La Línea Nadella es la pregunta del paradigma; el AgencyDomain es la respuesta arquitectónica.

Esta página se dejó intencionalmente en blanco.

Capítulo 2 · El Mundo Agentivo

El Capítulo 1 estableció la pregunta divisoria: si el humano abre aplicaciones para hacer su trabajo, la organización vive en el lado agéntico; si no, ha cruzado al lado Agentivo. Este capítulo asume que la respuesta es **no** — que la organización ha cruzado la Línea Nadella — y desarrolla, con la calma y el detalle que la pregunta merece, qué significa ese cruce en la práctica.

La pregunta no es trivial. Cualquier ejecutivo, arquitecto o consultor que haya enfrentado una transición tecnológica grande sabe que las consecuencias del cruce nunca son el conjunto de slides con flechas optimistas que el primer evangelista promete. Las consecuencias reales son rugosas, asimétricas, parcialmente predecibles y parcialmente sorprendentes. Pero hay un núcleo de cambios que sí pueden anticiparse con disciplina, y son esos cambios los que este capítulo describe.

El cruce de la Línea Nadella cambia simultáneamente seis dimensiones de la operación de cualquier organización: la forma en que el humano interactúa con el sistema, la naturaleza de los datos que el sistema consume, los roles del trabajo humano, la economía de la información, la gobernanza, y el modelo operativo. Ninguna de las seis cambia aisladamente. Avanzar en una sin las demás produce **pilotos exitosos pero no transformación real** — observación que las firmas de consultoría que han documentado el campo repiten con frecuencia incómoda. El cruce es sistémico o no es.

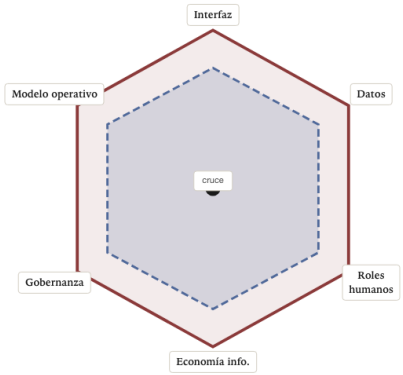
Comenzaremos por la consecuencia más visible: el colapso de la aplicación como interfaz primaria del trabajo cognitivo.

El colapso de la aplicación como interfaz

Durante cuarenta años el sistema de software empresarial se construyó alrededor de un postulado implícito que rara vez se hizo explícito: **la unidad mínima de interacción es la aplicación**. El humano abre Excel para modelar números. Abre Salesforce para gestionar oportunidades. Abre Power BI para revisar dashboards. Abre ServiceNow para abrir tickets. Abre Confluence para documentar. Cada aplicación tiene su pantalla, su menú, su modelo mental, su curva de aprendizaje. La habilidad valiosa del trabajador del conocimiento moderno consistía, en buena parte, en saber operar bien una colección razonable de aplicaciones — saber dónde está cada cosa, cómo se llega, qué botón pulsar.

Ese postulado deja de operar en el Mundo Agentivo. La aplicación, vista como interfaz primaria del trabajo, **colapsa**. Pero hay que ser preciso sobre qué exactamente colapsa, porque la afirmación leída sin matices puede sonar más radical de lo que es.

No todo desaparece. Distinguir es importante para no perder credibilidad ante un ejecutivo que tiene que decidir presupuesto. Las **GUIs como punto de entrada al trabajo** mueren rápido: el humano deja de abrir Salesforce para revisar pipeline, le pregunta a un agente cuáles oportunidades requieren atención. Las **aplicaciones como sistemas backend** sobreviven, pero invisibles: el agente que respondió sobre



Dimensión	Antes del cruce	Después del cruce
Interfaz	aplicaciones (GUIs, menús)	conversación con agentes
Datos	DW · calidad = limpieza	capa semántica · calidad = accionabilidad
Roles humanos	consumidores de información	diseñadores y supervisores de agentes
Economía de la info.	cada pregunta = proyecto (semanas)	cada pregunta = conversación (segundos)
Gobernanza	¿quién puede ver qué?	¿qué puede hacer un agente?
Modelo operativo	centros de competencia	fábricas de agentes

*Avanzar en una sola dimensión sin las demás produce **pilotos exitosos pero no transformación real**. El cruce es sistémico o no es.*

Figura 3: Las seis dimensiones del cruce

el pipeline consultó Salesforce vía API. Salesforce, como sistema, sigue ahí. Como interfaz humana, no. Las **suites ofimáticas tradicionales** — Office, Google Workspace — se reposicionan: Word, Excel y PowerPoint dejan de ser la primera elección del usuario que necesita escribir, calcular o presentar — el agente lo hace. Sobreviven en casos de edición fina, formato específico o trabajo creativo donde la conversación es ineficiente respecto al trabajo directo. Y las **herramientas especializadas con superficie compleja** — CAD, Figma, IDEs avanzados, DAWs musicales — sobreviven más tiempo: su interfaz codifica saber profesional que la conversación tarda en reemplazar.

El patrón emergente es nítido. Las aplicaciones que existen para **navegar y filtrar información** mueren rápido bajo la presión del Mundo Agentivo. Eran intermediarios visuales entre el humano y los datos, y un agente con acceso directo a los datos hace innecesario al intermediario. Las que existen para **producir artefactos especializados** sobreviven más, aunque eventualmente con copiloto o agente como mediador. La transición no es uniforme entre categorías de aplicaciones, y los responsables de stack que la planifican bien aceptan esa desigualdad.

Una imagen útil para sostener intuitivamente lo que ocurre: pensar el Mundo Agentivo como aquél en el que las aplicaciones empresariales viven **debajo** de una capa conversacional, no encima. El humano nunca las ve, pero los agentes las consultan continuamente. Salesforce no desaparece: se vuelve la base de datos de relaciones de cliente que un agente comercial consume sin que el humano vea jamás su UI. Power BI no desaparece: se vuelve un endpoint de consultas analíticas que un agente financiero invoca cuando se le pregunta sobre desempeño trimestral. ServiceNow no desaparece: sigue siendo el registro de tickets que un agente de operaciones consulta y actualiza sin abrir el portal. Las aplicaciones se convierten en **infraestructura backend invisible**. Su valor sobrevive como almacén estructurado de datos y lógica de negocio. Pierden su valor como interfaz.

Las aplicaciones no desaparecen. Desaparece la obligación de que el humano las abra.

La industria de analytics — la más antigua y consolidada del software empresarial — fue la primera en aceptar abiertamente que el modelo de "humano abre aplicación" llegó a su techo. Tellius lo formula con la franqueza propia de un actor que ha visto el ciclo: *"Dashboards still tell you what happened, but rarely why — and never what to do next."* Superwise extiende la observación: *"BI was built for a slower business environment — that assumption no longer holds true."* No son provocaciones de marketing — son reconocimientos de un problema operativo persistente que la industria de BI ha tratado de resolver durante quince años con sucesivos rediseños cosméticos, hasta entender que el problema no era cosmético sino estructural. La interfaz misma — el dashboard como artefacto visual que el humano debe abrir, mirar e interpretar — era el cuello de botella.

La nueva economía de la información

Si tuviera que destacar un solo cambio que el cruce de la Línea Nadella produce sobre la operación cotidiana de una organización, sería este: **el colapso del costo marginal de una pregunta analítica**. Es el cambio menos visible de los seis y, al mismo tiempo, el más transformador. Es la **condición habilitante** de todo lo demás que este capítulo describe.

En el modelo tradicional, cada pregunta nueva de negocio implica un proyecto. La secuencia es conocida y dolorosa: el ejecutivo formula la pregunta a su área de BI; el área de BI coordina con el ejecutivo para precisar el alcance; los analistas levantan los datos relevantes; los desarrolladores construyen el reporte o dashboard; los validadores confirman que el resultado es correcto; el ejecutivo recibe la respuesta. El proceso completo toma típicamente entre cuatro y doce semanas. El cuello de botella real, como suele decirse en las organizaciones maduras, **no es la tecnología — es la transferencia de conocimiento entre personas**. Hay humanos en el medio, y cada humano introduce latencia y posibilidad de error de interpretación.

El costo de poner el modelo tradicional en operación tampoco es trivial: construir la cadena Data Lake → Synapse → Power BI, o cualquier equivalente moderno, exige setup inicial de seis cifras, operación mensual sostenida y meses hasta el primer dashboard útil (el **Capítulo 7 desarrolla** las cifras detalladas de costo de BI). Y toda esa inversión entrega capacidad de responder solo aquellas preguntas que alguien previó al diseñar el sistema. Las preguntas no anticipadas — las que el ejecutivo realmente quiere hacer cuando llega el lunes con una intuición nueva — no están en el menú. Esperan en cola, o no se hacen.

Cuando ese costo colapsa de semanas a segundos, **cambia la naturaleza misma** de la relación entre la organización y su información. Tres efectos inmediatos transforman la operación cotidiana.

El primero es que **la capacidad analítica se vuelve elástica**. Se adapta en tiempo real a la necesidad actual, no a lo que alguien pre-definió hace meses. No hay menú fijo: hay capacidad de respuesta ilimitada dentro de los límites de los datos disponibles. El ejecutivo que tiene una intuición el lunes la explora el lunes — no espera al jueves a que el equipo de BI tenga el dashboard listo. La velocidad de la curiosidad analítica deja de estar limitada por la infraestructura.

El segundo es que **la iteración reemplaza a la especificación**. En el modelo tradicional, el ejecutivo debía especificar por adelantado qué quería ver, esperar el resultado, y a partir de ahí formular la siguiente pregunta. La latencia de cada ciclo era de semanas, así que las preguntas tenían que ser muy bien planteadas — el costo del error de planteamiento era alto. En el modelo agentivo, el ejecutivo plantea una primera pregunta aproximada, recibe la respuesta en segundos, refina, profundiza, descarta hipótesis, persigue otras. El conocimiento emerge del **diálogo**, no del proyecto. La forma misma de

hacer análisis cambia: de proyecto secuencial a conversación continua.

El tercero es quizá el más profundo: **las preguntas que nunca se hacían, ahora se hacen**. Cuando preguntar es gratis, la organización descubre insights que ni siquiera sabía que necesitaba. La curiosidad analítica deja de estar limitada por el presupuesto de BI. Un ejecutivo que en el modelo tradicional reservaba sus consultas a las preguntas más obvias y de mayor retorno — porque cada una le costaba al área de BI un proyecto — empieza a preguntar también las preguntas marginales, las hipótesis exóticas, los detalles que en el modelo viejo no justificaban el costo. Y descubre, con frecuencia, que las preguntas marginales contenían los insights más valiosos.

Esta transformación no es solo una mejora cuantitativa. Es **condición habilitante** de todo lo demás en este capítulo. El ciclo de inteligencia continua que describiremos en la próxima sección no puede existir si cada iteración toma semanas. La gobernanza de autonomía que describiremos más adelante no tiene sentido si los agentes no operan en tiempo real. La transformación de roles humanos que describiremos al final no ocurre si el acceso al conocimiento sigue dependiendo de intermediarios humanos. El colapso del costo de la pregunta no es feature de los agentes — es la **precondición** de todo lo demás.

Del ciclo clásico al ciclo de inteligencia continua

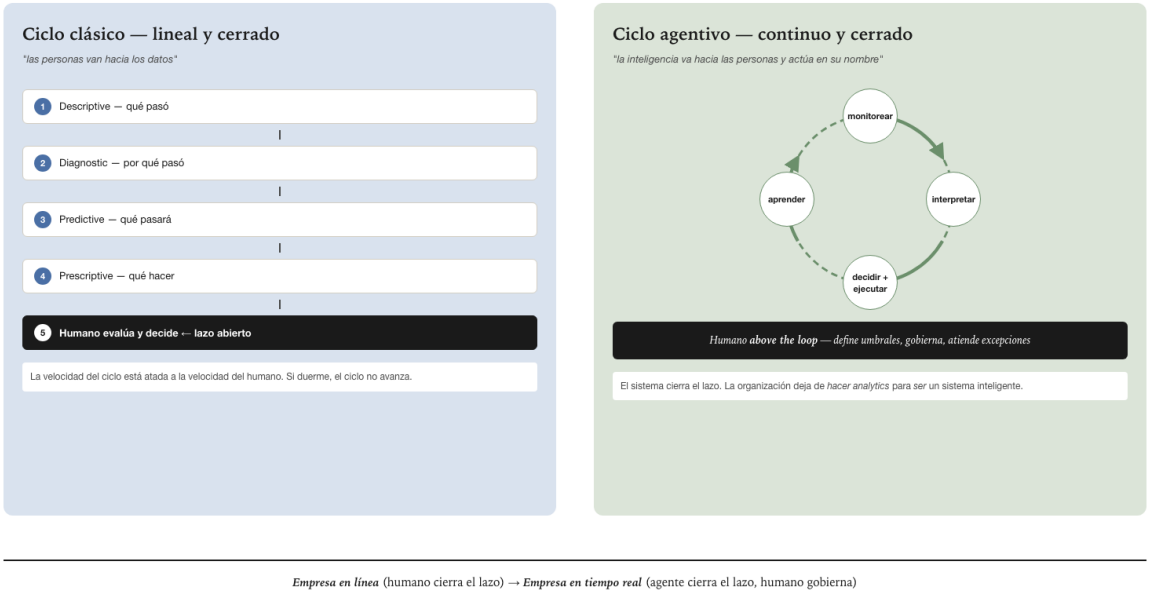


Figura 4: Del ciclo clásico al ciclo de inteligencia continua

Durante treinta años el paradigma de gestión de información se sostuvo sobre el principio **"las personas van hacia los datos"**. La frase suena anodina pero codifica todo el modelo operativo del BI clásico: se construye un data warehouse, se montan dashboards, se entrenan usuarios, y se espera que alguien mire el reporte correcto en el momento correcto y tome la decisión correcta. Todo el modelo descansa sobre la **atención humana** como el recurso escaso, el cuello de botella alrededor del cual se diseña el sistema.

El ciclo clásico es lineal: descriptive → diagnostic → predictive → prescriptive → humano decide. Empieza con datos describiendo qué pasó, sigue diagnosticando por qué pasó, predice qué pasará, prescribe qué hacer, y termina con un humano evaluando la prescripción y decidiendo. El humano que decide es el final del ciclo — el último paso, el cierre. Y la velocidad del ciclo está atada a la velocidad de ese humano. Si el humano está ocupado, el ciclo no avanza. Si el humano está en vacaciones, el ciclo no avanza. Si el humano duerme, el ciclo no avanza.

La IA agentiva invierte ese flujo. La frase canónica del nuevo paradigma — y la encontrarán repetida en este libro porque es central — es: **"la inteligencia va hacia las personas, y actúa en su nombre."** Un sistema de agentes monitorea continuamente, interpreta lo que detecta, decide dentro de los límites que la organización ha definido, ejecuta la decisión, y escala al humano solo cuando corresponde — cuando algo sale del rango previsto, cuando el impacto excede umbrales, cuando se requiere juicio que el agente no tiene. El ciclo deja de ser lineal y se vuelve **continuo, autorregulado, agente-ejecutado, humano-gobernado**. Su formulación canónica — la misma en toda la trilogía — es **Percibir → Interpretar → Decidir → Actuar → Aprender**; la fórmula corta "detecta, interpreta, decide y actúa" que este libro usa al paso es su abreviatura, no otro ciclo.

El cambio crítico es estructural. El paso de *prescriptive* a acción ya no es una recomendación que un humano evalúa. Es una decisión que un agente ejecuta, monitorea el resultado, y ajusta. La organización deja de **hacer analytics** para **ser** un sistema inteligente. Esta es la formulación que el campo ha empezado a usar — *"continuous intelligence"* en el lenguaje de Gartner, *"agentic analytics"* en el de Tableau y Tellius, *"agentic BI"* en el de Databricks. Todos nombran el mismo desplazamiento: del ciclo donde el humano cierra al ciclo donde el agente cierra y el humano gobierna.

La transición entre los dos ciclos da lugar a una distinción organizacional que vale acuñar con cuidado, porque será referencia recurrente en el resto del libro. Una **empresa en línea** tiene sus datos actualizados, sus dashboards al día, su información accesible. Pero **depende de que un humano mire, interprete y decida**. Vive con el ciclo clásico, optimizado al máximo. Una **empresa en tiempo real**, en cambio, no solo accede a información: **detecta, interpreta, decide y actúa** de forma continua y autónoma, dentro de marcos gobernados. Vive con el ciclo agentivo. La frontera entre ambas es exactamente lo que el colapso del costo de la pregunta analítica habilita — es lo que llamamos **el Salto Cuántico**, el evento a partir del cual la naturaleza misma de la operación cambia. Antes del Salto, una organización puede tener la mejor infraestructura del mundo y seguir siendo empresa en línea. Después del Salto, la misma infraestructura se vuelve sustrato de una empresa en tiempo real.

La distinción importa porque captura algo que las métricas tradicionales de madurez en BI no capturan. Una organización con dashboards perfectos, datos actualizados al segundo, y todo el equipo de BI funcionando como reloj, sigue siendo empresa en línea si los humanos siguen siendo los que miran y deciden. La empresa en tiempo real no es la versión más rápida de la empresa en línea — es **otra cosa**. La diferencia no es de velocidad, es de modelo operativo.

En el vocabulario de la arquitectura que el resto del libro desarrolla, empresa en línea y empresa en tiempo real son **puntos del continuo de temporalidad** de los componentes que sostienen la operación: el "tiempo real" no se habilita eligiendo un canal que empuje datos más seguido, sino dando **temporalidad continua** a los componentes que operan en nombre de la organización. La spec de manifestación y temporalidad (`discreta/continua`) vive en el **Capítulo 5 §2**.

Cube lo plantea sin retórica: *"The modern data stack is beginning to show its age."* BCG lo lleva al plano operativo, describiendo cómo la IA agentiva orquesta acciones en toda la cadena de valor, *"closing the loop between insight and execution."* La frase es exacta: el ciclo clásico **dejaba el lazo abierto** — terminaba con una recomendación que el humano cerraba con su decisión. El ciclo agentivo **cierra el**

lazo — el sistema mismo decide y ejecuta, dentro de los marcos que el humano definió. Es paradigma distinto, no incremento sobre el actual.

El cruce no es un interruptor: una organización lo atraviesa de forma gradual, y dentro de ella la proporción entre trabajo asistido (el agente como **Asistente** reactivo) y trabajo autónomo (el **Agente Autónomo** que cierra el lazo) se desplaza a medida que madura. A modo ilustrativo:

Etapa	Asistente	Agente Autónomo
1 · Inicial	90 %	10 %
2 · Adopción	70 %	30 %
3 · Madurez	50 %	50 %
4 · Avanzada	30 %	70 %

Las cifras son orientativas, no medidas: marcan la **dirección** del desplazamiento — de un mundo donde el humano gobierna cada paso a uno donde el agente sostiene la operación y el humano gobierna los marcos. La distinción Asistente vs Agente Autónomo se desarrolla en el Capítulo 5 §5.

Tres ejes de cambio profundo

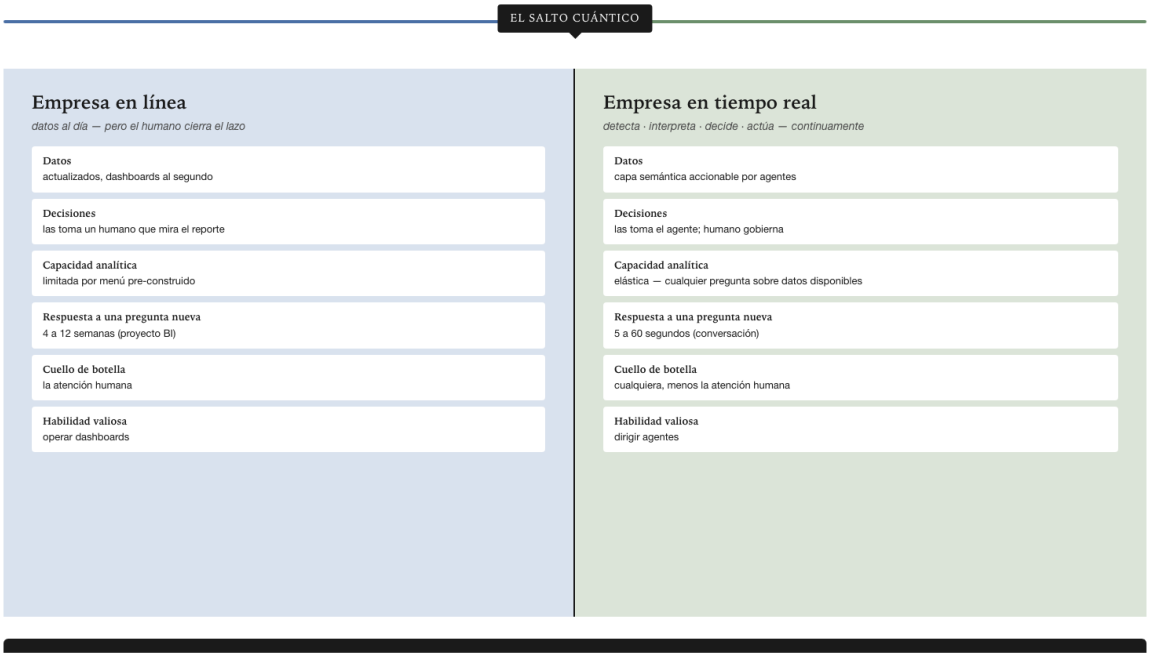


Figura 5: Empresa en línea → empresa en tiempo real · el Salto Cuántico

Una organización que cruza la Línea Nadella experimenta tres ejes simultáneos de transformación — los tres ejes agrupan las seis dimensiones del cruce: la relación humano-información y los roles del trabajo humano se condensan en el primero; los datos y el modelo operativo, en el segundo; la gobernanza

y la economía de la información, en el tercero. Tomados aisladamente, cada eje suena como mejora razonable. Tomados juntos, constituyen un cambio de modelo operativo. La advertencia, recurrente entre las firmas que han documentado el campo: avanzar en uno solo de los ejes sin los demás produce **pilotos exitosos pero no transformación real**. Los tres son interdependientes.

De consumir información a gobernar agentes

El primer eje cambia la relación del humano con la información. En el paradigma actual, el trabajador del conocimiento es **consumidor de información**: alguien construye reportes, alguien construye dashboards, y el trabajador los lee, interpreta y decide. La habilidad valiosa es **literacy de datos** — saber leer tablas, entender visualizaciones, formular hipótesis a partir de números. El valor está en *entender* la información.

En el paradigma emergente, el trabajador del conocimiento se mueve hacia un rol distinto: **diseñador y gobernador de agentes**. Las personas dejan de mirar reportes y pasan a diseñar las reglas, los umbrales, los protocolos bajo los cuales agentes monitorean, interpretan y actúan. La habilidad valiosa es **diseño y supervisión de sistemas autónomos** — saber formular las reglas correctas, saber evaluar el comportamiento agregado del agente, saber detectar cuándo el agente opera fuera de rango. El valor está en *gobernar la acción inteligente*, no en consumir información sobre ella.

El cambio es radical pero no súbito. El CFO que en el paradigma actual revisa un dashboard de cashflow todas las mañanas, en el paradigma emergente define los umbrales y protocolos que un agente financiero ejecuta autónomamente. El agente monitorea continuamente, ejecuta acciones de gestión de liquidez dentro de los límites que el CFO definió, escala al CFO solo cuando se aproxima a los umbrales o cuando detecta condiciones anómalas. El CFO ya no mira el dashboard — mira el comportamiento del agente, ajusta los umbrales cuando aprende algo nuevo, interviene cuando el agente le notifica una anomalía. El CFO sigue siendo CFO, pero su trabajo cotidiano cambió de naturaleza.

McKinsey describe esta transición con precisión en su reporte sobre la "Organización Agentiva": los empleados pasan de ejecutar tareas a orquestar resultados, supervisar agentes, establecer objetivos y gestionar trade-offs. La frase recurrente entre los analistas — *"humanos above the loop"* — captura el desplazamiento. El humano no está fuera del lazo de decisión, ni dentro: está **encima** del lazo, definiendo sus parámetros y supervisando su comportamiento agregado. McKinsey estima que el setenta y cinco por ciento de los roles actuales requerirán rediseño, upskilling o reubicación para 2030.

BCG documenta una consecuencia organizacional concreta de este desplazamiento: el cuarenta y cinco por ciento de los líderes de IA esperan necesitar menos capas de management intermedio. La razón es estructural. El management intermedio existe en buena parte para coordinar la ejecución entre niveles — pasar instrucciones del nivel ejecutivo al nivel operativo, monitorear que se ejecuten, reportar de vuelta. Cuando el agente ejecuta autónomamente, ese rol coordinativo pierde necesidad. Lo que sobrevive del management intermedio es la parte que aporta juicio profesional: definir las reglas correctas, manejar excepciones complejas, mediar entre objetivos en tensión. La parte coordinativa pura desaparece.

Aparecen, simétricamente, roles nuevos. Un análisis de CIO.com los enumera con detalle: **AI Agent Orchestrator** (la persona responsable de la flota de agentes que opera en una función o área), **Human-Agent Interaction Designer** (la persona que diseña cómo los humanos interactúan productivamente con los agentes que gobiernan), **AI Ethics & Governance Specialist** (la persona que asegura que los agentes operen dentro de los límites éticos y regulatorios), **AgentOps Specialist** (el equivalente del DevOps Engineer pero para flotas de agentes). Es un organigrama nuevo. No reemplaza al organigrama tradicional inmediatamente — convive con él durante años — pero refleja el desplazamiento del trabajo

humano hacia gobernar agentes que ejecutan.

De arquitectura para humanos a arquitectura para agentes

El segundo eje cambia los datos y la arquitectura subyacente. Esta es la dimensión menos visible para el ejecutivo y más crítica para el arquitecto técnico. La razón: la arquitectura de datos del paradigma actual está optimizada para que humanos consulten — y eso es estructuralmente incompatible con que agentes consulten correctamente.

El paradigma actual asume que el consumidor final de los datos es un humano operando una aplicación. Los data warehouses se optimizan para consultas SQL escritas por analistas. La calidad de dato significa **limpieza**: filas sin nulos, formatos consistentes, fechas en su lugar. Los modelos de datos se diseñan para alimentar visualizaciones — Power BI, Tableau, Looker. La integración entre sistemas opera por batches o bajo demanda, en frecuencias que el humano puede tolerar.

El paradigma emergente cambia cada uno de estos supuestos. Los datos se consumen por agentes, que no leen filas — interpretan **significado**. Los warehouses por sí solos son insuficientes: necesitan capa semántica explícita encima, donde el agente lea no solo las tablas sino la **intención de las tablas**: qué significa este indicador, cómo se relaciona con esos otros, qué transformaciones son legítimas, qué casos especiales aplican. La calidad de dato deja de significar limpieza y pasa a significar **accionabilidad** — por qué un dato impecablemente limpio puede valer menos para un agente que uno imperfecto pero rico en contexto es algo que la aplicación canónica del Capítulo 7 desarrolla.

AtScale midió esta diferencia cuantitativamente. Según AtScale, los agentes que consultan datos sin capa semántica fallan en más del **ochenta por ciento** de las consultas — generan SQL incorrecto, malinterpretan métricas, alucinan relaciones que no existen; los mismos agentes con capa semántica explícita alcanzan, en el mismo estudio de AtScale, una precisión muy alta. La conclusión que AtScale extrae no admite ambigüedad: *"Para los agentes de IA, el semantic layer no es nice-to-have — es el fundamento que hace que la IA sea verdaderamente útil."* El **Capítulo 7** aplica este resultado a la arquitectura del caso canónico (Kimball Barnizada).

ThoughtSpot acuñó el término **Agentic Semantic Layer** para describir la capa semántica diseñada nativamente para agentes — dinámica, context-aware, conectada a los flujos del agente. Salesforce, en su arquitectura de empresa agentiva, propone un **Enterprise Knowledge Graph** como capa central — knowledge graph en lugar de dimensional model, porque el grafo captura relaciones que la tabla bidimensional no puede capturar. Databricks habla de unificar infraestructura, datos y semántica para habilitar **Agentic BI**. Cada uno de estos vendors está atacando, desde su ángulo, el mismo problema — el que la aplicación canónica del Capítulo 7 enuncia como principio: los datos solos no le bastan al agente.

La industria todavía debate los detalles — knowledge graph versus capa semántica versus ontología versus modelo dimensional extendido — pero el reconocimiento de que algo nuevo es necesario es ya consenso. Informatica lo formula con franqueza: *"Because agents act without human approval loops, the data they use must be fully trusted, verified, and monitored."* Y propone SLAs de calidad de datos explícitos: menos de cinco minutos de frescura para agentes transaccionales, menos de una hora para agentes analíticos. Los SLAs viejos — actualizar el data warehouse cada noche — no sirven para sistemas que actúan en tiempo real.

Deloitte encontró que el cuarenta y ocho por ciento de las organizaciones citan la **descubribilidad de datos** como barrera principal para su estrategia agentiva, y el cuarenta y siete por ciento citan la **reutilización**. Los datos existen, pero los agentes no pueden encontrarlos o no pueden interpretarlos.

Es problema arquitectónico, no de cantidad. La transición del paradigma actual al emergente exige un cambio de paradigma en la arquitectura de datos: de pipelines ETL diseñados para alimentar dashboards a fabrics semánticos diseñados para razonamiento autónomo.

De gobernanza de acceso a gobernanza de autonomía

El tercer eje cambia cómo la organización ejerce control sobre lo que el sistema hace. Es el eje donde la mayor parte de los proyectos agentivos fracasan, según los datos de campo, y por eso merece atención cuidadosa.

La gobernanza tradicional de IT pregunta: **¿quién puede ver qué datos?**. El modelo de control es **acceso**: identidades autenticadas, roles asignados, permisos concedidos sobre recursos específicos. La pregunta es estática (los permisos rara vez cambian) y discreta (un usuario tiene o no tiene acceso a un recurso). Las herramientas tradicionales — IAM (Identity and Access Management), SSO, RBAC — están optimizadas para este modelo. Funcionan bien porque el sujeto del acceso (el humano) tiene identidad estable y el objeto del acceso (el recurso) tiene granularidad clara.

La gobernanza agentiva pregunta algo distinto: **¿qué puede hacer un agente, bajo qué condiciones?**. El sujeto del control no es un humano — es un agente que actúa en nombre de un humano u organización. El objeto del control no es un recurso aislado — es una secuencia de acciones que el agente puede ejecutar autónomamente. La pregunta es dinámica (las condiciones cambian con el contexto), continua (el agente actúa todo el tiempo, no solo cuando alguien apunta el cursor), y multi-dimensional (qué acción, sobre qué dato, con qué impacto, bajo qué umbral).

Las herramientas tradicionales de IAM son insuficientes en este modelo. Están diseñadas para sujetos humanos con identidad estable y permisos discretos, no para agentes que actúan continuamente con grados variables de autonomía. La organización que intenta gobernar a un agente con permisos clásicos descubre rápido que el modelo no captura las preguntas que necesita responder: ¿el agente puede ejecutar transferencias bancarias? Sí, pero ¿hasta qué monto sin aprobación humana? ¿bajo qué horarios? ¿con qué nivel de validación previa? ¿con qué auditoría posterior? Cada pregunta exige un mecanismo nuevo que el modelo de acceso clásico no tiene.

Los reguladores han reconocido este desplazamiento más rápido de lo que la industria suele aceptar. Singapore IMDA publicó en enero de 2026 el primer framework estatal de gobernanza para IA agentiva: el **Model AI Governance Framework for Generative AI** (MGF), que — pese a llevar la IA generativa en el nombre — dedica su cuerpo a los agentes que actúan con autonomía. El framework establece, con claridad poco habitual en regulación temprana, que aunque los agentes actúan autónomamente, *"la responsabilidad humana continúa aplicando"*. Las organizaciones deben hacer que la accountability humana sea significativa y que el human-in-the-loop sea efectivo en el tiempo. El World Economic Forum propone gobernanza progresiva: logging y trazabilidad para todos los agentes, identity tagging por acción, monitoreo en tiempo real. Su distinción clave: *"autonomy entails decision-making flexibility; automation emphasizes execution reliability"* — son design choices, no propiedades inherentes del sistema. La organización elige cuánta autonomía darle al agente; no la hereda como propiedad técnica del sistema.

BCG reporta que el cincuenta y ocho por ciento de los adoptadores pesados de IA esperan un cambio fundamental en su estructura de gobernanza en los próximos tres años, y un tercio cree que la IA tendrá más autoridad de decisión en el mismo período. NACD — la asociación nacional de directores corporativos en Estados Unidos — advierte que la IA agentiva impacta la supervisión del board, el cumplimiento regulatorio y la exposición al riesgo. La frase de KPMG resume bien el período: *"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."

La cifra que más preocupa, sin embargo, es la operativa. Gartner predice que más del cuarenta por ciento de los proyectos de IA agentiva serán cancelados antes de fines de 2027 — por costos, valor de negocio poco claro, o controles de riesgo inadecuados. La gobernanza no es opcional; es lo que separa pilotos de producción. La escala del problema es significativa: el ochenta por ciento de las organizaciones reportan comportamientos riesgosos de sus agentes — acceso no autorizado a datos, interacciones inesperadas, decisiones fuera de rango — y solo el veintiuno por ciento tiene modelos de gobernanza maduros. ISACA destaca que la IA agentiva presenta un desafío creciente para funciones de auditoría porque sus procesos de decisión carecen de trazabilidad clara.

El mensaje de los datos es nítido. Las organizaciones que sobrevivan al cruce serán las que hayan invertido en gobernanza de autonomía con la misma seriedad que invirtieron en gobernanza de acceso durante los últimos veinte años. Las que traten el problema como aplicación tardía de las herramientas viejas — un permiso más en el IAM, un rol más en el SSO — se quedarán del lado equivocado de esa proyección. Y la gobernanza de autonomía no se construye en el último mes antes del lanzamiento — se diseña desde el inicio, en la arquitectura misma del sistema. Es lo que el Capítulo 5 desarrolla bajo el nombre de **Trust Infrastructure**.

La naturaleza de la transición

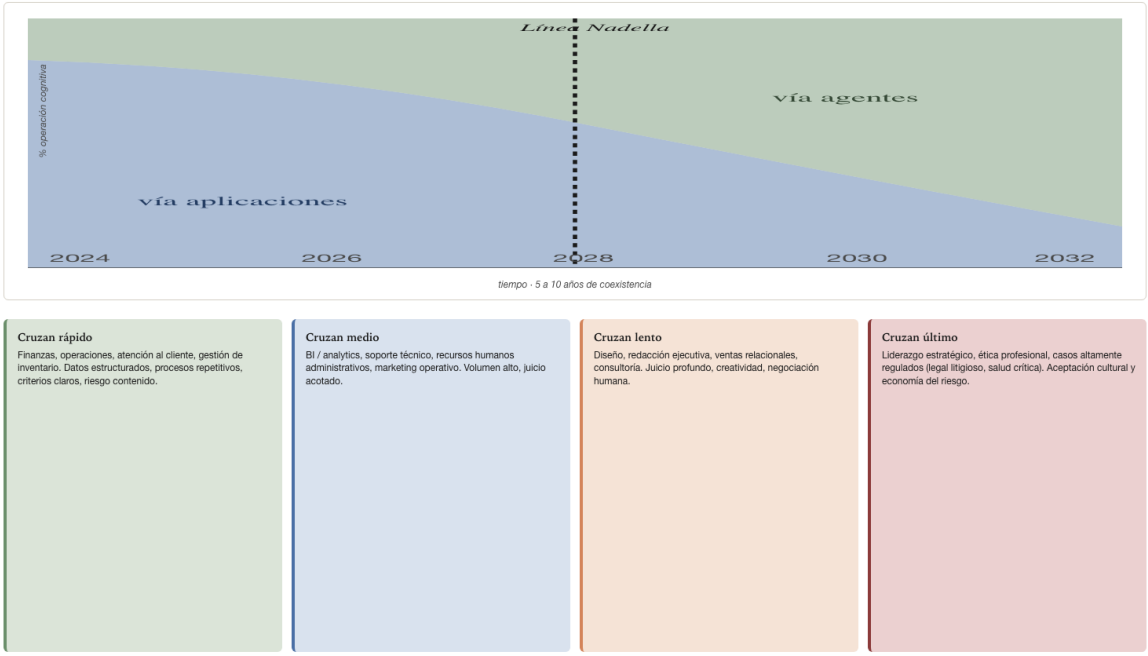


Figura 6: La transición no es un evento · coexistencia evolutiva

Hay un riesgo de leer las consecuencias del cruce como si fueran un evento único — el día que la organización pasó al lado Agentivo. Esa lectura es errada y, en términos prácticos, peligrosa: lleva

a planificar la transición como un big bang que casi siempre fracasa. La realidad es **coexistencia evolutiva**: el paradigma actual y el emergente coexisten durante años, con la proporción cambiando gradualmente de uno hacia el otro.

La infraestructura existente — data warehouses, pipelines ETL, herramientas de BI tradicional, ERPs, CRMs — no desaparece cuando una organización cruza la Línea Nadella. Cambia su rol. Deja de ser superficie con la cual el humano interactúa y se vuelve **fuente de datos consumida por agentes**. La arquitectura tradicional sigue siendo válida para data warehousing histórico a gran escala, para modelos analíticos altamente complejos que requieren pre-cálculo, para pipelines con lógica de negocio muy específica, para requerimientos regulatorios de retención y linaje formal. Lo que cambia no es si esos sistemas existen — es **quién los consume**. En las etapas tempranas, el noventa por ciento del consumo es por humanos vía dashboards y reportes; el agente es asistente puntual en el margen. En las etapas avanzadas, el agente orquesta la mayor parte de las consultas analíticas y la infraestructura tradicional opera invisible debajo, alimentándolo.

> *Cada etapa de la transición no invalida la anterior — la subsume.*

Una organización avanzada en la transición no eliminó su data warehouse. Lo integró en un fabric semántico que agentes consumen autónomamente. La empresa en línea no desaparece cuando emerge la empresa en tiempo real — se convierte en su cimiento. Esta es lectura crítica para no caer en el error opuesto al "big bang": el del fundamentalismo agentivo que descarta toda la infraestructura existente y trata de reconstruir desde cero. La organización seria reconoce que veinte años de inversión en BI tradicional construyeron un activo — los datos están limpios, los modelos están consensuados, los pipelines son confiables — y que ese activo es exactamente lo que el agente necesita por debajo. Tirarlo y empezar de nuevo es renunciar al activo. Mantenerlo y agregarle capa agentiva encima es capitalizarlo.

La coexistencia evolutiva no es una esperanza — es el **patrón documentado** de todas las transiciones de interfaz que la industria ya cruzó. Cuando el PC desafió al mainframe, la profecía fue la extinción; el punto de inflexión llegó en 1984 — las ventas de computadoras de escritorio superaron por primera vez a las de mainframes — y sin embargo los mainframes procesan hoy entre 2.500 y 3.000 millones de transacciones diarias, sosteniendo los mercados financieros globales. Cuando la web desafió al software instalado, y cuando la nube desafió al on-premise, el patrón se repitió, siempre con cuatro constantes: **la predicción inicial es extrema** (muerte total de lo anterior); **la realidad es coexistencia** (la tecnología "condenada" encuentra nichos de valor irreemplazable); **el plazo real multiplica por tres o cinco al predicho**; y **el estado estable es híbrido** — las capas se estratifican, no se reemplazan. La Línea Nadella heredará el patrón: el cruce será asimétrico, más lento que sus titulares, y las aplicaciones sobrevivirán largamente como sustrato — que es exactamente lo que la coexistencia evolutiva planifica en lugar de negar.

La transición tiene además dos asimetrías que importa reconocer. La primera, **asimetría entre funciones**: no todas las funciones cruzan al mismo ritmo. Funciones repetitivas con datos estructurados — finanzas, operaciones, atención al cliente, gestión de inventario — cruzan rápido, porque la economía es evidente y el riesgo del error es contenido. Funciones creativas o de juicio profundo — diseño, estrategia, negociación — cruzan más lentamente, no por incapacidad técnica de los agentes sino por aceptación cultural y por la economía del riesgo. Forzar un cruce uniforme en todas las funciones es uno de los antipatrones más comunes en programas de transformación agentiva fallidos. Las organizaciones serias dejan que cada función cruce a su ritmo, con planes específicos por área, en lugar de imponer un calendario único.

La segunda asimetría es **temporal**: las organizaciones que invierten temprano en arquitectura agentiva pagan el costo de explorar antes de tener referentes maduros, pero también capturan el aprendizaje que

viene de operar el modelo nuevo durante más tiempo. Las que esperan a que el campo madure pagan menos costo de exploración pero llegan al mercado con menor experiencia operativa. La elección entre las dos posiciones no es obvia — depende del apetito de riesgo, de la posición competitiva, del horizonte temporal del decisor. Pero la elección debe ser consciente. Quedar en la mitad — invertir lo suficiente para tener un piloto pero no lo suficiente para llegar a producción — es la peor posición posible. Es exactamente la posición que engrosa la proyección de cancelaciones de Gartner.

El estado del campo

Tratar el Mundo Agentivo como horizonte lejano es un error de diagnóstico. Los datos de campo a inicios de 2026 documentan una transición ya en curso, con velocidad significativamente mayor de lo que el marketing tradicional sugería hace dieciocho meses. Tres cifras dan dimensión al fenómeno y enmarcan la lectura del resto del libro.

La primera: para fines de 2026, **el cuarenta por ciento de las aplicaciones empresariales incluirán agentes de IA**, comparado con menos del cinco por ciento en 2025. La proyección es de Gartner, y el orden de magnitud — un crecimiento de ocho veces en doce meses — es lo que la hace remarcable. No es la cifra absoluta lo importante; es la pendiente. Una pendiente así implica que la transición está acelerándose, no estabilizándose.

La segunda: para 2028, **al menos el quince por ciento de las decisiones operativas diarias serán tomadas autónomamente por agentes**, sin humano en el lazo de decisión. La cifra es proyección de Gartner. Un quince por ciento parece poco, hasta que uno calcula cuántas decisiones operativas toma una empresa mediana al día — del orden de decenas de miles entre todas sus áreas — y entiende que el quince por ciento es un volumen significativo de operación autónoma cotidiana, con implicaciones regulatorias, organizacionales y técnicas que las empresas todavía no han mapeado completamente.

La tercera: el mercado global de IA agentiva pasa de **7.3 mil millones de dólares en 2025 a 139.2 mil millones proyectados para 2034** — una tasa compuesta superior al cuarenta por ciento anual sostenido durante una década. Esta cifra es la apuesta colectiva del capital sobre el horizonte agentivo. Cuando el capital apuesta así, no garantiza que la transición ocurra como predicha — pero sí garantiza que las organizaciones que apuesten en contra tendrán que justificar la apuesta contra inversión masiva en la dirección opuesta.

Hay tres lecturas posibles de estos datos, todas válidas, y la organización seria debe sostener las tres simultáneamente sin que se contradigan. La primera lectura: **el cruce está ocurriendo**. No es escenario futuro sobre el cual debatir si llegará — es presente medible. La segunda: **pero la mayoría está mal preparada** — los datos de gobernanza ya citados (riesgo extendido, madurez escasa, proyectos en camino a fallar) lo confirman. La tercera: **la diferencia entre éxito y fracaso es estructural**. Las organizaciones que sobrevivan al cruce serán las que hayan invertido en arquitectura agentiva formal, no en pilotos aislados sin disciplina.

Las tres lecturas son, en última instancia, la misma. Hay una transformación real ocurriendo. Hay un costo real de hacerla mal. Y la diferencia entre quienes la harán bien y quienes la harán mal no es sutil — es arquitectónica.

La obligación arquitectónica

Las consecuencias del cruce que este capítulo ha documentado no son aspiracionales — son **exigencias técnicas**. Cada consecuencia tiene una implicación arquitectónica directa, y la lista de implicaciones, leída en conjunto, es casi exactamente la lista de las cuatro capas y la infraestructura transversal que el Capítulo 4 propondrá como arquitectura formal.

El colapso de la aplicación como interfaz exige una capa de **Interacción** agnóstica al canal. Si el humano va a conversar con agentes en lugar de abrir aplicaciones, el sistema debe poder manifestarse coherentemente en chat, voz, canales corporativos, GUI on-the-fly, y eventualmente en cualquier canal nuevo que aparezca. Una capa de Interacción rígida, atada a una superficie particular, falla en el Mundo Agentivo.

La nueva economía de la información exige una capa de **Cognición** que pueda razonar sobre datos en tiempo real, multi-LLM, capaz de aplicar saber especializado y de delegar tareas repetitivas a otra capa que las ejecute sin invocarla cada vez. Una cognición monolítica, atada a un proveedor único, falla cuando la economía de operación lo exige.

La transformación de roles humanos hacia gobernar agentes exige que el agente tenga **vida persistente** — no sea solo asistente reactivo que responde y se olvida. Una capa de **Autonomía** donde el agente pueda mantener estado, ejecutar continuamente, monitorear, y actuar por iniciativa propia es lo que hace al humano *above the loop* en lugar de *in the loop*. Sin esa capa, el humano sigue siendo cuello de botella aunque crea estar gobernando.

La gobernanza de autonomía exige un punto de control donde se ejerza la política antes de la ejecución. Una capa de **Acceso** donde toda acción del agente sobre el mundo real pase por validación de políticas, registro auditable y controles configurables es lo que separa pilotos de producción. Sin esa capa, el sistema es indefendible regulatoriamente y operativamente.

Y la transición de pilotos a producción exige una **infraestructura de confianza transversal** — que no viva en una capa específica sino que atraviese las cuatro. Es lo que el Capítulo 5 desarrolla bajo el nombre de Trust Infrastructure: cinco pilares — Gobernanza, Auditoría, Validación, Resiliencia, Transparencia — que se ejercen en distintos puntos según el caso pero que juntos sostienen la confianza de la organización en lo que el sistema hace.

Cada exigencia es una capa. Cada capa es la respuesta a una consecuencia del cruce. La arquitectura agentiva no es propuesta abstracta diseñada en pizarra — es la lista de cosas que hay que resolver para no engrosar esas cancelaciones. Esa arquitectura, sus capas, sus primitivas, sus interfaces formales, es lo que el resto del libro desarrolla.

Quien haya seguido este capítulo entiende ahora qué cambia al cruzar la Línea Nadella, por qué el cruce es sistémico, y qué se le exige a una arquitectura que pretenda sostenerlo. Los capítulos siguientes entregan esa arquitectura. El lector que vaya a tomar decisiones de stack en horizonte de cinco años encontrará en ellos la disciplina que las previene.

Resumen visual

Dimensión	Antes del cruce	Después del cruce
Interfaz	Aplicaciones (GUIs, menús, pantallas)	Conversación con agentes
Datos	Data warehouses optimizados para SQL · calidad = limpieza	Capas semánticas donde agentes razonan · calidad = accionabilidad
Roles humanos	Consumidores de información · ejecutores de tareas	Diseñadores y supervisores de sistemas autónomos
Economía de la información	Cada pregunta nueva = un proyecto (semanas)	Cada pregunta nueva = una conversación (segundos)
Gobernanza	Quién puede ver qué datos · permisos estáticos	Qué puede hacer un agente, bajo qué condiciones
Modelo operativo	Centros de competencia · funciones especializadas	Fábricas de agentes · orquestación de ecosistemas

Capítulo 3 · Bounded Concerns Architecture

Los Capítulos 1 y 2 establecieron la frontera y describieron el destino. La Línea Nadella separa dos futuros del software, y el Mundo Agentivo es el lado al que las organizaciones cruzarán durante la próxima década. Pero antes de describir la arquitectura del destino — tarea del Capítulo 4 — hay que mirar con cuidado el lugar desde el cual se parte. La inmensa mayoría de las organizaciones que cruzarán la línea no lo harán construyendo desde cero: lo harán transformando un patrimonio de sistemas empresariales que llevan décadas operando, sostienen el corazón del negocio, y no pueden apagarse durante la travesía.

Este capítulo describe ese punto de partida con el mismo rigor con que el resto del libro describe el destino. El instrumento que vamos a usar se llama **Bounded Concerns Architecture** — abreviado BCA. No es una arquitectura del Mundo Agentivo: es la cartografía formal del estado *previo* al cruce, diseñada para que un arquitecto pueda mirar un sistema empresarial existente, identificar dónde vive cada responsabilidad, y razonar disciplinadamente sobre qué componentes migrarán al Mundo Agentivo, qué componentes desaparecerán, y qué componentes permanecerán como infraestructura backend invisible que los agentes consumirán sin que el humano la vea jamás.

La elección del nombre no es ornamental. *Bounded* — acotado — captura el principio operacional que sostiene la arquitectura completa: cada responsabilidad ocupa su lugar y no se desborda al territorio de la responsabilidad vecina. Esa disciplina, que en la era pre-agentiva era buena práctica recomendada, se vuelve **condición de viabilidad** apenas el sistema empieza a incorporar componentes agénticos. Sin esa disciplina, la volatilidad estadística que los agentes traen consigo contamina el núcleo estable del negocio, y lo que llevaba veinte años funcionando deja de ser confiable. Con esa disciplina, la incursión agéntica queda confinada a la capa que está diseñada para tolerar volatilidad, y el núcleo del negocio queda protegido durante toda la travesía.

> *BCA es la arquitectura de la transición, no la del destino. Describe el estado desde el cual se cruza la Línea Nadella, no el estado al que se cruza.*

La era Agentic exige tratamiento arquitectónico explícito

La incorporación de agentes a sistemas empresariales no es la adición de una nueva tecnología más. Es la introducción de una clase de comportamiento cuya naturaleza es estructuralmente distinta del código tradicional, y que por tanto exige tratamiento arquitectónico distinto.

El código tradicional — workflows, motores de reglas, clases de servicio, scripts de orquestación — tiene una propiedad común: su comportamiento está **especificado explícitamente** por humanos. Alguien

escribió, paso a paso, qué pasa cuando, en qué orden, con qué condiciones. Para entender qué hace, basta leer el código. Su evolución es deliberada: cuando cambia, alguien decidió cambiarlo y modificó el código correspondiente.

El código agéntico — agentes basados en modelos de lenguaje, sistemas de recomendación, optimizadores de aprendizaje automático, sistemas expertos — tiene una propiedad opuesta: su comportamiento está **derivado contextualmente** desde un modelo. Nadie programó paso a paso qué hace; alguien describió un objetivo o un patrón, y la entidad deriva el comportamiento desde sus pesos, datos de entrenamiento, prompts o reglas heurísticas. Para entender qué hace, no basta leer el código — hay que entender los datos o el modelo del que el comportamiento emerge. Su evolución es estadística: cuando cambia, no es porque alguien modificó una línea sino porque cambiaron los datos, los pesos o los prompts que sintetizan el razonamiento.

Esta diferencia tiene cuatro consecuencias arquitectónicas concretas que separan a los componentes agénticos de los componentes tradicionales. Los componentes procedurales se prueban con tests unitarios deterministas; los agénticos se prueban con datasets de evaluación y métricas agregadas. Los procedurales producen logs predecibles que reflejan ejecución de código; los agénticos producen logs que requieren interpretación porque el comportamiento varía con el contexto. Los procedurales son auditables línea por línea; los agénticos son auditables por sus inputs, outputs y métricas pero no por su lógica interna. Los procedurales evolucionan mediante refactor de código; los agénticos evolucionan mediante reentrenamiento, ajuste de prompts o cambio de modelos. Estas cuatro propiedades son lo suficientemente distintas como para justificar que la arquitectura las trate como ciudadanos separados.

La amenaza estructural que esta diferencia introduce en el sistema empresarial es la siguiente: **los agentes son volátiles, pero las invariantes del negocio no pueden serlo**.

Un agente evoluciona rápidamente. Cambia con cada actualización del modelo subyacente, con cada ajuste de prompt, con cada incorporación de nuevas herramientas o nuevos datos. Su comportamiento puede variar entre dos invocaciones idénticas. Su corrección no es binaria sino estadística: opera correctamente la mayoría de las veces, con un umbral de error tolerable que depende del caso de uso.

Las invariantes del negocio son lo opuesto. Un cliente tiene un identificador único. Una factura tiene un total que iguala la suma de sus líneas. Un servicio activado tiene una fecha de activación. Estas reglas no admiten errores estadísticos: o se cumplen siempre o el sistema deja de ser confiable. Su corrección es binaria, su ritmo de cambio es estructural, y su violación tiene consecuencias regulatorias, financieras u operacionales graves.

Cuando un agente y una invariante coexisten en la misma capa arquitectónica sin frontera explícita entre ellos, la volatilidad del primero contamina la confiabilidad de la segunda. Cada actualización del agente arriesga romper invariantes que llevaban años funcionando correctamente. Cada cambio de prompt puede alterar inadvertidamente el cumplimiento de una regla regulatoria. La auditoría se vuelve imposible porque ya no hay separación entre lo que se programó deliberadamente y lo que el agente decidió contextualmente.

La delgadez del dominio como respuesta

La respuesta operacional a esta amenaza es lo que llamaremos **la delgadez del dominio**. Si el núcleo autoritativo del sistema — donde viven las invariantes y la persistencia de la verdad del negocio — se mantiene estrictamente delgado, admitiendo únicamente lo que es estructural al dominio y expulsando toda lógica volátil hacia afuera, entonces la incursión agéntica queda confinada a la capa que está

diseñada para tolerar volatilidad. Los agentes operan en la lógica de orquestación, junto a los workflows procedurales tradicionales, y ambos invocan al núcleo como un servicio con contrato estable. El corazón del negocio queda protegido.

Esta es la posición operacional que sustenta toda la arquitectura. Mantener el dominio delgado no es una preferencia estética sino una condición necesaria para que capacidades procedurales y agénticas puedan coexistir en un mismo sistema sin que la última corroa la confiabilidad de la primera.

La distinción entre Systems of Record y Systems of Engagement, articulada por Geoffrey Moore[1], opera convergentemente con esta tesis pero a un nivel distinto. Moore distingue entre dos categorías de sistemas según su propósito y su ritmo de cambio: los Systems of Record almacenan el estado autoritativo del negocio y obtienen su valor de su confiabilidad, lo cual exige estabilidad; los Systems of Engagement median la interacción y obtienen su valor de su capacidad de adaptación, lo cual exige velocidad. Forrester[2] extendería más tarde la taxonomía agregando los Systems of Insight, orientados a generación de conocimiento. La dicotomía de Moore opera a nivel de inventario de sistemas — qué sistemas tiene una organización; la delgadez del dominio opera a nivel de organización del código dentro de un componente. Ambas convergen en la misma intuición fundamental, que Buschmann y colegas[3] articulan en términos del patrón Layers como criterio de descomposición: cada capa debe tener una responsabilidad distinta y específica respecto a abstracción, granularidad o **ritmo de cambio**. La era Agentic intensifica esta lógica: los componentes agénticos cambian a un ritmo aún más rápido que las políticas comerciales tradicionales — ritmo de modelo, no ritmo de negocio — y por tanto deben quedar aún más alejados arquitectónicamente del núcleo estable.

La tesis sostiene una posición específica sobre la jerarquía de decisiones arquitectónicas. Otras decisiones frecuentemente debatidas — la elección entre arquitectura monolítica o microservicios, la dirección de las dependencias, el grado de adopción de event sourcing, la separación command/query, la modalidad de despliegue de agentes — son importantes pero subordinadas. Operan dentro de un espacio cuya forma queda determinada antes por la decisión sobre dónde termina el dominio. Una arquitectura de microservicios con dominios obesos donde se introducen agentes reproduce los problemas de un monolito mal estructurado, distribuidos y ahora con volatilidad agéntica adicional. Una arquitectura hexagonal con un núcleo que mezcla invariantes y workflows usa la dirección correcta de las dependencias para proteger un núcleo mal diseñado, pero la incorporación de agentes en ese núcleo invalida la protección. Un dominio delgado, en cambio, produce un componente cuyo núcleo es estable, cuyas reglas son auditables y cuyas integraciones — procedurales o agénticas — son intercambiables.

Mantener el dominio delgado requiere una operación arquitectónica concreta: trazar una frontera explícita entre el dominio y la lógica de orquestación, y defender esa frontera contra el deslizamiento incremental que tiende a engordar el núcleo en el tiempo. El término *Bounded* del nombre del modelo recoge el principio de Separación de Preocupaciones articulado por Dijkstra[4]: la disciplina de estudiar un aspecto de un problema en profundidad, por su propia consistencia, sabiendo todo el tiempo que es solo uno entre varios. La separación de Dijkstra es originalmente cognitiva. Su transformación en principio arquitectónico durante las décadas siguientes — visible en patrones

[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, Julio 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. Reimpreso en *Selected Writings on Computing: A Personal Perspective*, Springer-Verlag, 1982.

como Layers[5], Hexagonal[6], Clean Architecture[7] y DDD[8] — produce el linaje teórico al que pertenece BCA. La diferencia entre BCA y sus predecesoras está en qué frontera específica se elige como la más consecuente: BCA elige la frontera del dominio, y la sostiene como condición de viabilidad para la era Agentic.

Las tres capas

BCA descompone un componente empresarial en tres capas. La Capa 1, **Presentation**, contiene las dos fronteras externas del componente: UI hacia humanos y API hacia sistemas. La Capa 2, **Business Logic**, contiene la lógica de orquestación, organizada en dos cajas paralelas: *Procedural* para el comportamiento programado explícitamente, *Agentic* para el comportamiento derivado contextualmente. La Capa 3, **Domain**, contiene el núcleo estable del componente, organizado en dos dimensiones ortogonales: dominio propio frente a dominio ajeno en el eje vertical, lógica frente a persistencia en el eje horizontal.

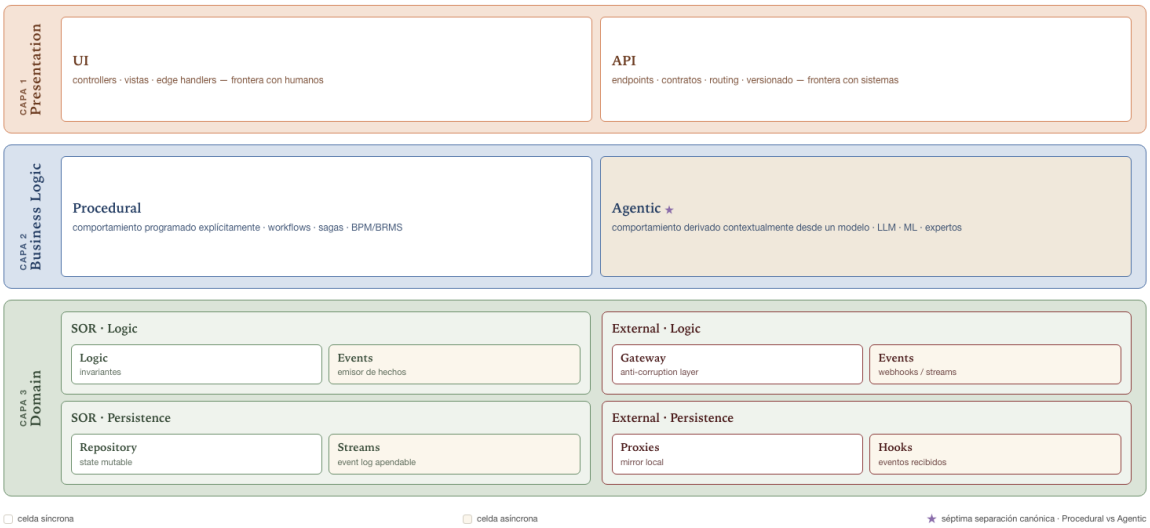


Figura 7: Bounded Concerns Architecture en tres capas con la séptima separación Procedural / Agentic en Business Logic.

La **Capa 1** trata UI y API como ciudadanos paralelos, no como variantes del mismo concern. UI son los controladores, vistas y manejadores que median la interacción con humanos; evoluciona al ritmo

[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", Abril 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.

de las necesidades de UX. API son los endpoints, contratos, mecanismos de routing y políticas de versionado que median la interacción con otros sistemas; evoluciona al ritmo de los contratos públicos del componente, bajo restricciones específicas de retrocompatibilidad y semántica precisa[9]. En la era pre-agentiva esta separación era buena práctica de microservicios; en la era Agentic cobra relevancia adicional, porque los agentes pueden operar tanto sobre la UI (asistiendo al usuario humano) como sobre la API (consumiendo y produciendo contratos máquina-a-máquina), y reconocer la distinción permite gestionar adecuadamente sus respectivos modos de invocación.

La **Capa 2** contiene la lógica de orquestación: la lógica que sabe qué pasos componen un caso de uso, en qué orden ocurren, qué ocurre si uno falla. Aquí vive la **séptima separación canónica** del modelo, sobre la que vamos a volver con detalle más adelante. *Procedural* comprende toda la lógica cuyo comportamiento está programado explícitamente por humanos: motores de procesos, motores de reglas, sagas, choreography frameworks, orquestadores de workflows, clases de servicio, scripts de coordinación. La distinción operativa no es de tecnología sino de naturaleza: el comportamiento está codificado paso a paso por un humano y puede entenderse leyendo el código línea por línea. *Agentic* comprende los componentes cuyo comportamiento se deriva contextualmente desde un modelo, no desde una secuencia codificada. Las dos cajas comparten capa porque ambas son lógica de orquestación — ambas operan sobre el dominio sin formar parte de él — pero su naturaleza operacional distinta justifica el tratamiento como ciudadanos paralelos.

La frontera entre la Capa 2 y la Capa 3 — entre orquestación y dominio — es la frontera operacional de la tesis. La Capa 3 contiene exclusivamente lógica que el negocio considera estructural; todo lo demás vive en la Capa 2, sea Procedural o Agentic. El criterio operativo es de naturaleza temporal: si una regla puede cambiar como consecuencia de una decisión comercial sin alterar el modelo fundamental del negocio — o si la regla es derivada estadísticamente y no explícitamente codificada — pertenece a la Capa 2; si la regla es estructural al dominio y cambiarla implicaría un cambio en la naturaleza misma del negocio, pertenece a la Capa 3.

La **Capa 3** se organiza en dos dimensiones ortogonales que producen cuatro celdas. La dimensión vertical separa el dominio propio del ajeno: la columna **SOR** (System of Record) contiene los componentes que modelan datos cuya autoridad pertenece al sistema — el núcleo autoritativo, lo que se valida, lo que se persiste como verdad, lo que emite eventos cuando cambia; la columna **External** contiene los componentes que conocen, traducen y representan localmente datos cuya autoridad vive afuera, en sistemas con los que el componente colabora — CRMs, billing externo, OSS de provisión, partners B2B, APIs de terceros. La dimensión horizontal separa la lógica de la persistencia: la banda **Logic** contiene las invariantes en SOR y las traducciones desde el modelo ajeno en External; la banda **Persistence** contiene los mecanismos físicos de almacenamiento.

Cada una de las cuatro cajas resultantes se subdivide internamente en una vía síncrona y una vía asíncrona. La asimetría léxica entre **Streams** (lado SOR) y **Hooks** (lado External) es deliberada. En el lado SOR el sistema emite hechos propios y los persiste como un log apendable; la autoría es propia y la dirección es saliente. En el lado External los eventos llegan porque sistemas externos los empujan mediante webhooks o equivalentes; la autoría es ajena y la dirección es entrante. Llamar a ambos *Streams* perdería esa información direccional importante.

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

Las siete separaciones estructurales

Tomadas en conjunto, las decisiones de las tres capas producen siete separaciones estructurales explícitas. Cada una responde a la observación clásica de que las preocupaciones diferenciadas tienen ritmos de cambio distintos, audiencias distintas y semánticas distintas, y por tanto merecen tratamiento arquitectónico distinto.

#	Separación	Materializada por
1	Presentación humana vs contrato externo	Cajas UI y API en la Capa 1
2	Orquestación vs reglas del dominio	Frontera entre la Capa 2 y la Capa 3
3	Lógica vs persistencia	Bandas horizontales en la Capa 3
4	Dominio propio vs ajeno	Columnas SOR y External en la Capa 3
5	Comunicación síncrona vs asíncrona	Vías paralelas dentro de cada celda
6	Estado mutable vs log de eventos	Pares Repository/Streams y Proxies/Hooks
7	Comportamiento procedural vs agéntico	Cajas Procedural y Agentic en la Capa 2

La separación 2 es la frontera operacional de la tesis. Las separaciones 3 a 6 estructuran el contenido interno de la Capa 3. La separación 1 estructura el contenido interno de la Capa 1. Y la separación 7 — la séptima separación canónica — estructura el contenido interno de la Capa 2 reflejando explícitamente la naturaleza dual de la lógica de orquestación en la era Agentic.

Cada celda de la arquitectura tiene su propio ritmo de cambio. SOR · Logic y SOR · Persistence cambian al ritmo de la evolución del modelo del negocio fundamental — años o décadas. External · Logic y External · Persistence cambian al ritmo de los sistemas externos a los que sirven. Business Logic Procedural cambia al ritmo de las políticas comerciales — meses. Business Logic Agentic cambia al ritmo de los modelos subyacentes — semanas o días. API cambia al ritmo de los contratos públicos. UI cambia al ritmo de las necesidades de UX. Esta diferenciación justifica políticas de versionado, despliegue y testeo distintas para cada celda, y es la razón estructural por la que la séptima separación es necesaria: el ritmo Agentic es lo suficientemente rápido como para requerir su propio régimen.

Genealogía intelectual

BCA no inventa primitivas. Sintetiza patrones publicados, debatidos y refinados por la comunidad de arquitectura de software durante las últimas dos décadas, los reorganiza alrededor de un principio operacional explícito — la delgadez del dominio — y los reposiciona en relación con un fenómeno emergente: la incorporación de capacidades agénticas en sistemas empresariales. Cada decisión del modelo se ancla en literatura previa.

La frontera entre las Capas 2 y 3 — entre orquestación y dominio — proviene directamente de

Fowler[10]. En *Patterns of Enterprise Application Architecture*, Stafford —autor del patrón *Service Layer* en esa obra— articula la distinción entre *domain logic*, que tiene que ver puramente con el dominio del problema, y *application logic*, que tiene que ver con responsabilidades de aplicación como notificaciones, integraciones y workflows. Fowler refuerza la distinción en *Domain Logic and SQL*[11] y reconoce sus límites prácticos en *Organizing Presentation Logic*[12]. La separación entre lógica y persistencia dentro de la Capa 3 se desprende de los patrones *Data Mapper* y *Repository*[13], cuyo propósito explícito es mantener el modelo del dominio limpio de detalles de almacenamiento.

El Domain Layer como contenedor de invariantes proviene de Evans[14]. En *Domain-Driven Design* propone una arquitectura de cuatro capas (UI / Application / Domain / Infrastructure) que coincide estructuralmente con BCA, con dos diferencias deliberadas: BCA bifurca el dominio en SOR y External, incorporando explícitamente la integración con sistemas ajenos como ciudadana plena del dominio; y BCA separa la persistencia del dominio dentro de la misma capa, mientras que en Evans la persistencia queda en Infrastructure.

BCA se aparta deliberadamente del DDD canónico en un punto específico: la ubicación de los Domain Services. Bajo DDD canónico, los Domain Services — la lógica que coordina múltiples agregados — pertenecen al Domain Layer. BCA los ubica en Business Logic. La razón es la asimetría temporal articulada antes: los Domain Services contienen frecuentemente políticas comerciales que cambian con el negocio (políticas de elegibilidad, reglas de descuento, procedimientos de aprobación), mientras que las invariantes estructurales del dominio cambian a un ritmo fundamentalmente más lento. En la era Agentic esta asimetría se intensifica: parte de la coordinación pasa a ser responsabilidad de agentes, lo cual añade volatilidad estadística a la volatilidad temporal ya presente. Vernon[15] discute la tensión entre Domain Services y Application Services sin resolverla en una sola dirección, reconociendo que la elección depende del contexto. BCA toma posición.

La filosofía de la pierna asíncrona se ancla en Helland[16]. *Data on the Outside vs. Data on the Inside* articula que los datos autoritativos viven dentro de un servicio en un mundo transaccional con cambios serializables, mientras que los datos que circulan entre servicios son inmutables. Una década más tarde, en *Immutability Changes Everything*[17], Helland profundiza: los hechos del dominio son intrínsecamente inmutables — los eventos que ocurren no pueden borrarse; las correcciones son nuevos eventos. Kleppmann[18] sintetiza esta tradición articulando que estado mutable y log de eventos pueden coexistir como representaciones complementarias del mismo dominio, posición que es la base conceptual de la coexistencia de Repository y Streams en la columna SOR de BCA.

Young[19] articula CQRS defendiendo que el lado de comandos debe ser delgado y centrado en proteger invariantes; la lógica de procesos vive en sagas y process managers que coordinan los SoR, no dentro

[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*, Febrero 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.

[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. Republicado en *ACM Queue* 18(3), 2020.

[17] Helland P. "Immutability changes everything". *7th Biennial Conference on Innovative Data Systems Research (CIDR)*, 2015. Republicado en *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. Disponible en `https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf`.

de ellos. BCA adopta el espíritu de CQRS sin llegar al CQRS estricto: las celdas síncronas de la Capa 3 están delgadas, las celdas asíncronas existen como ciudadanas complementarias, pero el estado mutable y el event log coexisten en lugar de uno derivar del otro. Fowler[20] sintetiza la posición de Young en su entrada de bliki sobre CQRS.

La columna External de la Capa 3 generaliza el patrón Anti-Corruption Layer originalmente propuesto por Evans[21] y refinado por Vernon[22] con foco específico en cómo se integran bounded contexts entre sí. Vernon articula explícitamente que la integración con sistemas legados es uno de los casos paradigmáticos donde el patrón aplica.

La separación de la API como concern arquitectónico distinto no aparece en los textos clásicos de Fowler[23], Evans[24] o Moore[25], pero es práctica establecida en arquitectura de microservicios moderna. Newman[26] dedica capítulos enteros a las decisiones de diseño de API, separándolas explícitamente de la presentación al usuario. La pierna asíncrona se apoya en el catálogo formal de patrones de mensajería de Hohpe y Woolf[27] — sus 65 patrones constituyen el vocabulario de las celdas Events, Streams y Hooks. En el contexto telco específicamente, el TM Forum[28] clasifica los bloques funcionales de la Open Digital Architecture según la dicotomía de Moore: Party Management, Core Commerce Management y Production como SoR; Engagement Management como SoE; Intelligence Management como SoI. La adopción de la columna SOR en BCA es coherente con esta clasificación: si TMF ya define ciertos bloques funcionales como SoR, tiene sentido que la lógica que reside dentro de ellos sea consistente con el propósito de un SoR según Moore.

La séptima separación canónica — Procedural / Agentic — y el reposicionamiento del modelo como arquitectura del estado pre-agentivo se anclan en *El Futuro Agentivo*[29] y en este libro. Allí se articula la Línea Nadella como frontera conceptual entre dos mundos posibles para el software empresarial. BCA toma posición específica: es la arquitectura de la transición Agentic, y cede el lugar al marco de la Arquitectura Agentiva del Mundo Agentivo (Capítulo 4) cuando la organización cruza la línea.

Mapa comparativo de propuestas

Las propuestas mencionadas en la genealogía conviven en la conversación arquitectónica como un archipiélago donde cada autor traza líneas en lugares distintos. La figura siguiente las ubica en un mapa bidimensional que permite visualizar cómo se relacionan entre sí y por qué la posición de BCA es una elección, no la única opción razonable.

El mapa se construye sobre dos dimensiones cuya naturaleza metodológica es deliberadamente distinta. El eje X es **medible** según una escala discreta de cinco niveles. El eje Y es **cualitativo** y se construye

[20] Fowler M. "CQRS". *martinfowler.com*, Julio 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.

[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*, 2ª 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, Versión 5.1.0, Marzo 2021.

[29] Obach C. *El Futuro Agentivo*. Documento técnico, ultraBASE — Grupo Ultra, 2025. Material absorbido en los Capítulos 1 y 2 de este libro.

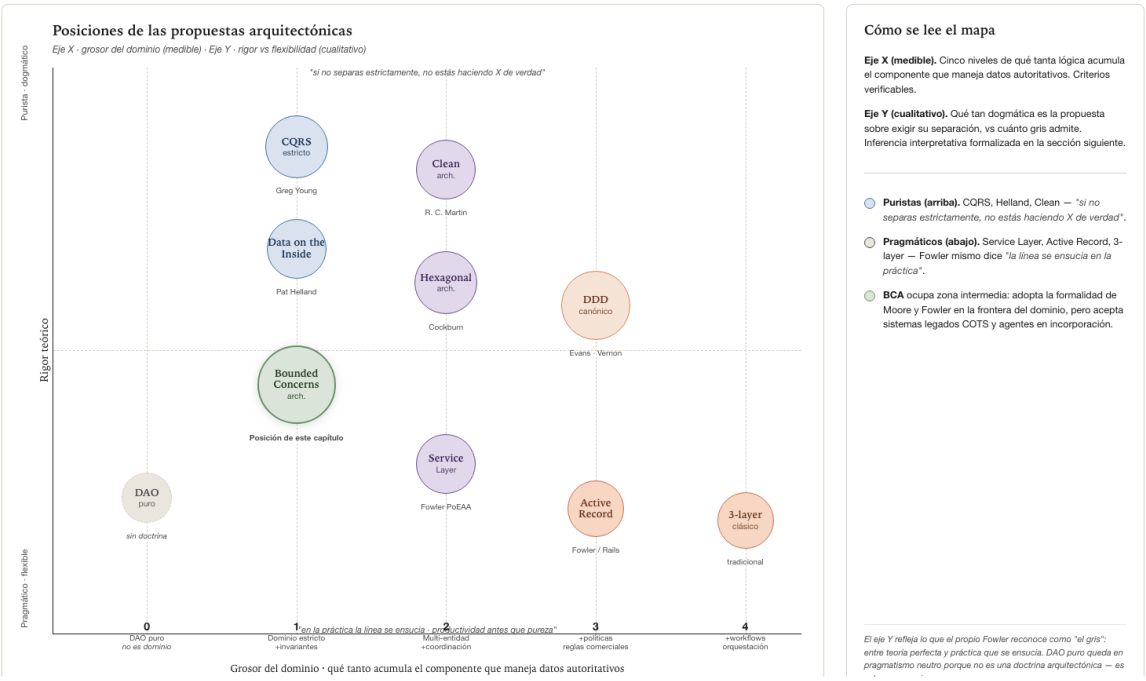


Figura 8: Mapa comparativo de propuestas arquitectónicas en el espacio “grosor del dominio” × “rigor teórico vs flexibilidad real”.

mediante una inferencia interpretativa que la sección siguiente formaliza.

El **eje X — Grosor del dominio** mide cuánta lógica acumula el componente que maneja datos autoritativos. Los niveles son acumulativos: cada nivel incluye lo que está en los niveles inferiores. *Nivel 0 — DAO puro*: solo CRUD sobre el almacenamiento, sin invariantes ni auditoría; no califica como dominio en sentido estricto. *Nivel 1 — Dominio estricto*: añade invariantes intrínsecas del agregado, identidad estable y auditoría de cambios. Es el grosor que la tesis defiende como óptimo para la era Agentic. *Nivel 2 — Multi-entidad*: añade operaciones que coordinan varios agregados del mismo dominio (Domain Services). *Nivel 3 — +políticas*: añade reglas comerciales cambiables (campañas, descuentos, restricciones de elegibilidad). *Nivel 4 — +workflows*: añade orquestación de procesos completos con múltiples pasos e integraciones externas.

El **eje Y — Rigor teórico vs flexibilidad real** mide qué tan dogmática es la propuesta sobre exigir su separación. En el polo purista están las propuestas que sostienen *"si no separas estrictamente, no estás haciendo X de verdad"*. En el polo pragmático están las propuestas que reconocen explícitamente que la línea conceptual se ensucia en la práctica. El polo pragmático refleja exactamente el caveat que Fowler[30] reconoce sobre las arquitecturas en capas.

Las posiciones reflejan la postura del autor original de cada propuesta, no las interpretaciones que las distintas comunidades han hecho de ellas. En el cuadrante superior-izquierdo — puristas con núcleo mínimo — viven CQRS estricto[31] y Data on the Inside[32]. Ambas propuestas exigen un dominio delgado y son dogmáticas al respecto. En el cuadrante superior-medio — puristas con marcos arquitectónicos — viven Clean Architecture[33] y Hexagonal[34]. Ambas definen reglas estrictas sobre la dirección de las dependencias y la separación entre dominio e infraestructura, pero permiten cierta riqueza multi-entidad dentro del núcleo. En el cuadrante superior-derecho — purista con núcleo grueso — vive DDD canónico[35], refinado por Vernon[36]. Acumula más lógica dentro del Domain Layer mediante Domain Services pero con disciplina estricta sobre Bounded Contexts, Anti-Corruption Layers y agregados bien delimitados.

En la zona intermedia se ubica **Bounded Concerns Architecture**. En el eje X coincide con CQRS y Helland (nivel 1, dominio estricto). En el eje Y queda en zona intermedia: adopta la formalidad de Fowler y Moore en la *frontera* del dominio (qué entra y qué no), pero es pragmática sobre la *implementación interna* del componente, reconociendo que en sistemas empresariales con productos COTS no se tiene control sobre la organización interna del software comprado, y que en la era Agentic los componentes Agentic introducen una capa adicional de variabilidad que requiere flexibilidad operacional.

En el cuadrante inferior-medio — pragmático declarado — vive Service Layer[37]. Es la propuesta más explícitamente pragmática de las maduras: el propio Fowler[38] escribe que la distinción entre application logic y domain logic se ensucia en la práctica. En el cuadrante inferior-derecho —

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

[31] Young G. "CQRS Documents", 2010. Disponible en `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. Republicado en *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", Abril 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`.

pragmáticos con núcleo grueso — viven Active Record[39] y 3-layer clásico[40]. Active Record fue diseñado para productividad — el lema *"convención sobre configuración"* de Rails es su ethos. El 3-layer clásico, sin disciplina arquitectónica adicional, tiende a acumular workflows enteros dentro de la capa de Business Logic mezclados con persistencia.

El mapa es una herramienta de orientación, no una clasificación oficial. Tres precauciones de lectura. Primera, las posiciones reflejan la postura del autor original; distintas comunidades han interpretado las mismas propuestas con mayor o menor rigor (existe DDD-lite y existe DDD ortodoxo). Segunda, el eje Y es cualitativo y depende de una inferencia interpretativa que la siguiente sección formaliza. Tercera, este mapa cubre propuestas para organizar el interior de un componente; no habla de patrones de integración entre componentes, que son ortogonales y están catalogados en otros trabajos[41][42].

Modelo de scoring para el eje cualitativo

El eje Y del mapa ubica las propuestas en una dimensión cualitativa de "rigor teórico vs flexibilidad real". Esta posición no proviene de las fuentes — Fowler nunca escribió que CQRS es más purista que Service Layer en términos formales — es una inferencia interpretativa construida a partir de la lectura de las fuentes.

Para que esa inferencia sea verificable y discutible, el "pragmatismo" se descompone en cinco dimensiones observables, cada una puntuable de 0 a 3 según criterios textuales explícitos. La suma resultante (0 a 15) ubica a cada propuesta en el eje Y, donde 0 representa máximo rigor purista y 15 representa máxima flexibilidad pragmática. Este modelo no pretende producir un score definitivo. Pretende hacer visible y disputable el razonamiento que llevó a cada ubicación. Cualquier lector con acceso a las fuentes primarias puede recalibrar las celdas y obtener una posición distinta — eso es deseable, no un defecto.

D1 — Lenguaje normativo. ¿La propuesta usa frases del tipo *"debes / nunca / siempre / strict / must"* sobre la separación, o usa *"considera / muchas veces / depende / a menudo"*? **D2 — Reconocimiento del gris práctico.** ¿La propuesta admite explícitamente casos donde aplicar la regla causa daño, o defiende la pureza ante todo escenario? **D3 — Permisividad ante variantes lite.** ¿Existen variantes flexibles toleradas o promovidas por el autor original? **D4 — Foco.** ¿La propuesta pretende reorganizar todo el sistema, o se ofrece como un patrón entre muchos en una caja de herramientas? **D5 — Compatibilidad con sistemas legados.** ¿La propuesta ofrece caminos para integrarse con sistemas existentes que no la siguen, o exige greenfield?

La aplicación a las propuestas evaluadas produce los scores de la tabla siguiente.

Propuesta	D1	D2	D3	D4	D5	Total
CQRS estricto[43]	0	0	0	2	2	**4**

[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.

[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*, 2ª ed. O'Reilly, 2021.

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

Propuesta	D1	D2	D3	D4	D5	Total
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**
DDD canónico[47]	1	2	2	1	2	**8**
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**

Los scores producen una correlación que sustenta la tesis del capítulo. Las propuestas con menor score en el eje Y (CQRS estricto: 4/15; Clean Architecture: 4/15; Data on the Inside: 5/15) son también las que mantienen los dominios más delgados cuando se aplican con disciplina. Las propuestas con mayor score (Service Layer y Active Record: 13/15 ambas) tienden a producir dominios variables porque su flexibilidad lo permite.

BCA ocupa una posición intermedia (9/15) que la tesis defiende como pragmáticamente óptima en contextos empresariales con sistemas legados y capacidades agénticas en incorporación: suficientemente firme en la frontera del dominio como para preservar su delgadez, suficientemente flexible en la implementación interna como para coexistir con software comprado, motores de procesos heredados y agentes de naturaleza estadística sobre los cuales no se tiene control arquitectónico tradicional.

Caso de aplicación · Activar el servicio de banda ancha de un cliente nuevo

Para hacer tangibles las tres capas y la distribución de responsabilidades — incluida la séptima separación Procedural / Agentic — considérese un caso del dominio de las telecomunicaciones: la activación del servicio de banda ancha de un cliente nuevo. Es un workflow representativo de las operaciones cotidianas bajo el marco TM Forum ODA[50] que toca tanto el dominio propio (cliente,

[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. Republicado en *ACM Queue* 18(3), 2020.

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

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

[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, Versión 5.1.0, Marzo 2021.

servicio, suscripción) como dominios ajenos (OSS de provisión de red, billing externo, proveedor de email), y se enriquece en la era actual con la incorporación de un asistente conversacional que apoya al agente de call center y un generador automatizado de contenido para la comunicación al cliente.

La distribución de las acciones entre las celdas de BCA es la siguiente:

Acción	Capa	Celda específica
Recibir petición HTTP del agente de call center	1 · Presentation	UI
Validar contrato de la API (campos, tipos, autenticación)	1 · Presentation	API
El asistente conversacional sugiere el plan óptimo según contexto del cliente	**2 · Business Logic**	**Agentic**
Verificar elegibilidad del plan según cobertura geográfica	2 · Business Logic	Procedural
Consultar al OSS si hay puerto disponible	3 · Domain	External · Logic → Gateway
Cachear localmente la respuesta del OSS	3 · Domain	External · Persistence → Proxies
Aplicar campañas vigentes y descuentos promocionales	2 · Business Logic	Procedural
Validar que el cliente no tenga otro servicio activo del mismo tipo	3 · Domain	SOR · Logic
Cambiar atómicamente el estado a `active` con timestamp	3 · Domain	SOR · Persistence → Repository
Emitir evento `ServiceActivated`	3 · Domain	SOR · Logic → Events
Persistir el evento en el log autoritativo	3 · Domain	SOR · Persistence → Streams
Recibir webhook del OSS confirmando provisión	3 · Domain	External · Logic → Events
Persistir el webhook en el log de eventos externos	3 · Domain	External · Persistence → Hooks
Notificar a billing del nuevo servicio	2 · Business Logic	Procedural
Generar email de bienvenida personalizado al perfil del cliente	**2 · Business Logic**	**Agentic**

Cinco observaciones se desprenden de esta distribución. **Primera**, la mayoría del comportamiento vive en Business Logic, distribuido entre Procedural y Agentic. Solo cuatro de las quince acciones tocan el dominio propio, y dos tocan el dominio ajeno; las nueve restantes viven en la Capa 2 o en Presentation. La era Agentic añade naturaleza al comportamiento de la Capa 2 sin tocar el núcleo del dominio. **Segunda**, Procedural y Agentic coexisten naturalmente en la misma capa: las dos acciones agénticas no están aisladas en una capa especial ni son tratadas como casos excepcionales — son ciudadanos plenos de

Business Logic, intercalados con las acciones procedurales según la lógica del workflow. **Tercera**, las acciones agénticas no tocan el dominio: operan exclusivamente en Business Logic, toman como input información del dominio y producen output que vuelve a la Capa 2 o a la Capa 1, pero nunca modifican directamente el estado autoritativo del SOR. Es la materialización operacional de la tesis: las invariantes del dominio quedan protegidas de la volatilidad agéntica porque la frontera entre las Capas 2 y 3 es respetada estrictamente. **Cuarta**, las dos vías de comunicación coexisten naturalmente: las acciones síncronas usan las celdas síncronas de cada caja, las acciones asíncronas usan las celdas asíncronas, y ningún workflow está obligado a usar exclusivamente una vía u otra[51]. **Quinta**, el núcleo del dominio queda intacto y delgado: las cuatro acciones marcadas con SOR son precisamente lo que la tesis identifica como núcleo autoritativo — invariantes y persistencia, con sus contrapartes asíncronas. Toda la lógica volátil — sea procedural o agéntica — vive afuera del dominio.

Implicaciones arquitectónicas

La adopción de BCA tiene implicaciones específicas para el diseño de componentes empresariales en la era Agentic. **Los componentes que cumplen rol de SOR deben exponer APIs delgadas centradas en operaciones que preservan invariantes**, coherente con la visión de Young[52] sobre el lado de comandos en CQRS. Los componentes de orquestación procedural y agéntica deben ser explícitos y separados; no deben estar disfrazados como extensiones del dominio. La persistencia debe ser intercambiable sin tocar la lógica del dominio. Los componentes de integración (Gateway, Proxies, Events, Hooks) deben tratarse como modeladores de un dominio ajeno[53][54]. Los componentes asíncronos (Events, Streams, Hooks) deben tratarse como ciudadanos plenos del diseño con su propia lógica de versionado, su propia semántica de entrega y sus propios SLAs[55].

La ubicación de la lógica de IA y agentes autónomos merece atención específica. Los componentes de inteligencia y agentes — orquestadores basados en modelos de lenguaje, recomendadores, optimizadores, sistemas expertos — son por naturaleza lógica de procesos coordinadores. Su lugar correcto en BCA es la caja Agentic dentro de Business Logic, no Domain. Esto preserva la estabilidad del SOR mientras permite que la lógica agéntica evolucione rápidamente. La pierna asíncrona del dominio (celdas Events, Streams, Hooks) es particularmente útil para integrar agentes: pueden suscribirse a eventos del dominio para reaccionar autónomamente sin acoplarse al núcleo del SOR. La séptima separación dentro de Business Logic permite además aplicar regímenes de gobernanza específicos a cada tipo de componente: testing determinista versus evaluación con datasets, observabilidad línea-por-línea versus métricas agregadas, auditoría por código versus auditoría por inputs/outputs, evolución por refactor versus evolución por reentrenamiento. Estos regímenes diferenciados son críticos para mantener trazabilidad y cumplimiento regulatorio en sistemas que combinan ambos tipos de comportamiento.

La coexistencia de estado mutable y log de eventos es una decisión arquitectónica explícita de BCA. El estado mutable y el log de eventos coexisten como representaciones complementarias del dominio, no como sustitutos. Esto sitúa la propuesta en una posición intermedia entre los sistemas tradicionales — donde solo existe estado mutable y los eventos, si los hay, son emergentes y no autoritativos — y

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

[52] Young G. "CQRS Documents", 2010. Disponible en `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.

el Event Sourcing puro — donde el estado se reconstruye siempre desde el log de eventos y el estado no es autoritativo, solo derivado. En BCA, el estado mutable es la fuente de verdad principal y el log de eventos es un complemento autoritativo. La técnica de Change Data Capture discutida por Kleppmann[56] proporciona el mecanismo concreto para mantener ambas representaciones sincronizadas.

Limitaciones del modelo

BCA es una posición arquitectónica entre varias legítimas, no la única. DDD canónico[57][58] es una posición arquitectónica respetable y tiene méritos genuinos, especialmente en dominios donde la lógica del negocio es muy rica y se beneficia de quedar concentrada en objetos del dominio. La elección entre DDD canónico y BCA debe hacerse caso por caso, considerando el contexto. La tesis del capítulo sostiene que en la era Agentic BCA produce mejores resultados, pero esta afirmación no se extiende universalmente a todos los contextos.

La frontera del dominio no siempre es nítida en la práctica. Como reconoce Fowler[59], la distinción entre *domain logic* y *application logic* se ensucia en sistemas reales. Habrá casos ambiguos donde no esté claro si una validación es intrínseca al dominio o es política de negocio. El criterio operativo recomendado es de naturaleza temporal — si la regla puede cambiar por una decisión comercial sin alterar el modelo del negocio fundamental, va en Business Logic; si la regla es estructural al dominio, va en SOR · Logic — pero este criterio no elimina la ambigüedad: la hace discutible y resoluble caso por caso.

La frontera entre Procedural y Agentic puede difuminarse. Una limitación específica de la era actual es que la separación entre componentes Procedural y Agentic no siempre es nítida. Un BRMS clásico puede incorporar reglas heurísticas con propiedades estadísticas; un agente moderno puede operar dentro de un workflow procedural rígido. La séptima separación captura una distinción fundamental — origen explícito versus derivación contextual del comportamiento — pero su aplicación práctica requiere juicio en casos límite, particularmente en sistemas híbridos que combinan razonamiento simbólico con inferencia estadística.

BCA no reemplaza a otros marcos. Esta descomposición es complementaria, no sustitutiva, de otros marcos: TM Forum ODA[60] para clasificación funcional a nivel de inventario de sistemas, principios cloud-native para despliegue, patrones de integración como los catalogados por Hohpe y Woolf[61] para comunicación entre componentes, y el marco de la Arquitectura Agentiva del Mundo Agentivo (Capítulo 4) para sistemas que han cruzado la Línea Nadella. BCA aplica al *interior* de un componente o conjunto de componentes que operan en el estado pre-agentivo.

Y el modelo de scoring de la sección anterior tiene tres limitaciones reconocidas. Primera, las puntuaciones reflejan interpretaciones del autor sobre los textos de cada propuesta; un evaluador con acceso a las mismas fuentes puede llegar a puntuaciones distintas, especialmente en dimensiones como D2 y D3 que requieren juicio sobre el espíritu de los textos. Segunda, la suma simple de las cinco dimensiones asume implícitamente que tienen pesos iguales, lo cual es una simplificación; en algunos contextos D5 (compatibilidad con legados) puede pesar más que D4 (foco), y viceversa. Tercera, el

[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.

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

[60] TM Forum. "ODA functional architecture", IG1167, Versión 5.1.0, Marzo 2021.

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

modelo evalúa la postura del autor original, no la práctica efectiva de las comunidades que adoptan cada propuesta.

Mapping al Mundo Agentivo

Hasta aquí hemos descrito la cartografía del estado pre-agentivo. La pregunta operativa que este libro plantea — y que justifica la presencia de BCA en él — es la pregunta del puente: cuando una organización cruza la Línea Nadella, ¿qué le pasa a cada una de las celdas de BCA? ¿Cuáles migran al Mundo Agentivo y se transforman en algo distinto? ¿Cuáles permanecen como infraestructura backend invisible? ¿Cuáles desaparecen?

La respuesta no es uniforme entre celdas. El cruce afecta a cada una de manera distinta, en momentos distintos, con consecuencias distintas. Lo que sigue es el mapping celda por celda — la pieza que convierte a BCA en instrumento de migración, no en mera descripción del pasado.

(Nota de lectura: la columna de destino usa el vocabulario que los Capítulos 4 y 5 definen — Botlet, Capability, Conector, Trust Infrastructure. En primera lectura basta la columna de trayectoria — qué se preserva, qué se transforma, qué desaparece —; la columna de destino cobra sentido pleno al volver aquí después del Capítulo 5.)

Celda BCA	Trayectoria al cruzar la Línea Nadella	Destino en el Mundo Agentivo
UI (Capa 1)	Se vacía progresivamente. La aplicación como interfaz primaria del trabajo cognitivo colapsa, según describimos en el Capítulo 2.	Reemplazada por la **Capa 1 — Interacción** del Mundo Agentivo (modalidades conversacionales, GUI on-the-fly, señalética pasiva, canales corporativos). La UI tradicional sobrevive sólo en herramientas especializadas con superficie compleja.
API (Capa 1)	Persiste y se intensifica. El agente que resolvió la pregunta del humano consulta la API del componente sin que el humano la vea.	Se convierte en **Conector** dentro de la **Capa 4 — Acceso** (no en Capability). Lo que era contrato máquina-a-máquina entre sistemas pasa a ser contrato máquina-a-agente, descubrible y gobernado por Trust Infrastructure.
Business Logic — Procedural (Capa 2)	Se contrae. Los workflows que codifican secuencias rígidas de pasos se reabsorben en agentes que componen los pasos contextualmente desde el catálogo de Capabilities.	Reemplazada por **Botlets** que orquestan dinámicamente el caso de uso. Sobreviven como Procedural sólo los workflows con requisitos regulatorios estrictos donde la trazabilidad línea por línea es no-negociable.

Celda BCA	Trayectoria al cruzar la Línea Nadella	Destino en el Mundo Agentivo
Business Logic — Agentic (Capa 2)	Se expande hasta volverse el comportamiento dominante de la capa. Lo que en BCA era ciudadano paralelo del Procedural pasa a ser el centro de gravedad del sistema.	Es la **semilla operativa del Botlet** del Capítulo 4. La séptima separación de BCA es, retrospectivamente, la primera grieta visible del Mundo Agentivo dentro del Mundo Agéntico.
SOR · Logic (Capa 3)	Se preserva. Las invariantes estructurales del dominio no cambian al cruzar la línea — un cliente sigue teniendo identificador único, una factura sigue cuadrando con sus líneas.	Se convierte en lógica de invariantes dentro del **AgencyDomain** correspondiente. La regla operativa es la misma: el dominio sigue siendo delgado, ahora también frente a Botlets en lugar de frente a workflows procedurales.
SOR · Persistence (Capa 3)	Se preserva. El estado autoritativo del negocio sigue viviendo en bases de datos transaccionales con sus garantías ACID.	Persiste como capa de almacenamiento del **AgencyDomain**. Lo que cambia es quién la consulta: ya no usuarios humanos vía aplicaciones, sino Botlets vía Capabilities.
External · Logic (Capa 3)	Se preserva pero se reposiciona. La lógica que traduce modelos ajenos a modelo propio sigue siendo necesaria, pero ahora opera dentro de una red federada de AgencyDomains que se confían entre sí.	Se reabsorbe en patrones de federación entre AgencyDomains gestionados por **Trust Infrastructure** (el Capítulo 4 describe la mecánica).
External · Persistence (Capa 3)	Se preserva. Los proxies locales y los logs de eventos recibidos siguen siendo el mecanismo físico para integrar dominios ajenos.	Permanecen como capa de almacenamiento de la federación. La novedad agentiva no es estructural sino de gobierno: Trust Infrastructure registra qué AgencyDomain leyó qué de quién y bajo qué política.

Celda BCA	Trayectoria al cruzar la Línea Nadella	Destino en el Mundo Agentivo
Eventos (síncronos y asíncronos en SOR)	Se preservan y se elevan en importancia. La pierna asíncrona del dominio es particularmente útil para integrar agentes: un Botlet puede suscribirse a eventos del SOR para reaccionar autónomamente sin acoplarse al núcleo.	Se convierten en el sustrato de coordinación entre Botlets y AgencyDomains. La inmutabilidad del log autoritativo permite que Botlets razonen sobre la historia del dominio sin contaminar el estado actual.

Cuatro patrones generales emergen del mapping. **Primero**, la **Capa 3 sobrevive casi intacta**. Las invariantes estructurales y la persistencia autoritativa son lo que el sistema empresarial tiene que preservar durante el cruce, y la disciplina de la delgadez del dominio es lo que asegura que sobrevivan. La Capa 3 de BCA es, en buena medida, lo que se vuelve el sustrato de los AgencyDomains del Capítulo 4. **Segundo**, la **Capa 2 se transforma profundamente**. La caja Procedural se contrae; la caja Agentic se expande hasta dominar; ambas eventualmente dejan de ser cajas paralelas y se convierten en Botlets que componen comportamiento desde el catálogo de Capabilities. La séptima separación de BCA es la grieta inicial por la cual el Mundo Agentivo entra al sistema empresarial; con el tiempo, esa grieta se ensancha hasta consumir la capa entera. **Tercero**, la **Capa 1 se bifurca**. La UI se vacía y eventualmente desaparece como interfaz primaria; la API se intensifica y se transforma en Conector (Capa 4), no en Capability. La transformación es asimétrica: una de las dos cajas de la Capa 1 muere; la otra prospera con un nombre distinto. **Cuarto**, **Trust Infrastructure aparece como concern transversal nuevo** que en BCA no estaba materializado. La gobernanza, auditabilidad y políticas de identidad y acceso, que en BCA eran responsabilidades dispersas entre las celdas, se elevan a una capa transversal explícita en el Mundo Agentivo. Esta es una de las contribuciones arquitectónicas del Capítulo 4: hacer visible lo que en la era pre-agentiva quedaba implícito en cada componente.

El mapping también permite contestar preguntas operativas que un arquitecto en transición se hace cotidianamente. *¿Tengo que reescribir mi SOR para entrar al Mundo Agentivo?* No: la Capa 3 sobrevive. Lo que tienes que hacer es exponerla con disciplina vía Capabilities. *¿Mis workflows actuales mueren?* No todos: sobreviven los que tienen requisitos de trazabilidad regulatoria; los demás se reabsorben en Botlets. *¿Mis APIs actuales sirven?* Sí, pero hay que renegociar su contrato como Conector (Capa 4) descubrible y gobernado por Trust Infrastructure. *¿Mis aplicaciones con UI mueren?* Las que existían para navegar y filtrar información, sí. Las que producen artefactos especializados, sobreviven más tiempo, eventualmente con copiloto o agente como mediador, según ya describimos en el Capítulo 2.

Cierre · La frontera de BCA

BCA es la arquitectura del estado pre-agentivo. Su valor en este libro es doble. Primero, ofrece una cartografía formal de lo que la mayoría de las organizaciones que cruzarán la Línea Nadella tienen hoy: tres capas, siete separaciones, un núcleo delgado, una orquestación dual entre Procedural y Agentic. Segundo, y más importante, ofrece el instrumento de mapping que permite razonar disciplinadamente sobre qué migra, qué desaparece, qué permanece — y por qué.

Pero BCA tiene una frontera explícita. Cuando una organización cruza la Línea Nadella y entra

plenamente al Mundo Agentivo, las separaciones que BCA articula tienden a colapsar. La Capa 1 se convierte en una conversación con un agente. La distinción entre Procedural y Agentic se diluye porque la primera caja se vacía progresivamente. Eventualmente, incluso la frontera entre Business Logic y Domain queda mediada por una entidad agéntica que decide qué pasos invocar. En ese momento, BCA ha cumplido su propósito de transición y cede el lugar a un marco arquitectónico distinto — un marco que ya no se organiza en torno a la separación entre dominio y orquestación, sino en torno a las cuatro capas del Mundo Agentivo: Interacción, Cognición, Autonomía y Acceso.

Ese marco es el que el Capítulo 4 desarrolla. Lo hace sobre la base que BCA acaba de establecer: sabemos qué teníamos antes, sabemos qué migra, sabemos qué permanece. Ahora podemos describir, con la calma que el cambio merece, la arquitectura del lado al que vamos.

Esta página se dejó intencionalmente en blanco.

Capítulo 4 · Arquitectura Agentiva

Los Capítulos 1 y 2 establecieron el paradigma; el Capítulo 3 cartografió el estado pre-agentivo desde el cual la mayoría de las organizaciones cruzarán la línea, e identificó qué celdas de ese estado migran y cómo. Este capítulo entrega la respuesta arquitectónica del lado al que se cruza. Lo que sigue es la **Arquitectura Agentiva** — el diseño técnico canónico de un sistema construido para vivir del lado correcto de la línea.

La pregunta operativa que la arquitectura responde es concreta: ¿cómo se construye un sistema en el que un agente pueda razonar, persistir, actuar sobre el mundo real, y rendir cuentas de lo que hace — sin que esas cuatro funciones se mezclen en un magma indistinguible? La pregunta no es retórica. La mezcla — fundir cognición, autonomía, acción y gobernanza en una sola superficie indiferenciada — es exactamente lo que produce los pilotos que funcionan en demo y mueren al pasar a producción enterprise. Es el patrón recurrente detrás del cuarenta por ciento de proyectos cancelados que documentamos en el Capítulo 2. La causa raíz de ese fracaso no es técnica en el sentido de que falten algoritmos o capacidad de cómputo: es **arquitectónica**. Los sistemas no separan responsabilidades, y sin separación de responsabilidades no hay manera de razonar disciplinadamente sobre lo que hacen.

La respuesta que este libro propone es **separación de responsabilidades en cuatro capas, organizadas en topología paralela**, gobernadas por una infraestructura de confianza transversal y ordenadas por un principio rector que llamaremos *Agent First*. Las cuatro capas no son división arbitraria — cada una corresponde a una responsabilidad arquitectónica distinta que puede razonarse, especificarse e implementarse independientemente. La infraestructura transversal no es capa adicional, es propiedad que atraviesa las cuatro. Y el principio rector no es slogan: es regla operativa de diseño que ordena cómo se resuelve cualquier disyuntiva.

La precisión sobre **topología paralela** importa desde la primera lectura. La numeración de las capas (1 → 2 → 3 → 4) sugiere una secuencia, pero la operación real del sistema no es lineal. Las Capas 2 (Cognición) y 3 (Autonomía) son **vías paralelas** entre la Capa 1 (Interacción) y la Capa 4 (Acceso), no etapas en serie. Una operación que entra por Capa 1 puede llegar a Capa 4 atravesando la Cognición — costosa y decisiva — o atravesando la Autonomía — barata y repetitiva, vía Botlets. Las dos vías interactúan entre sí pero ninguna domina sobre la otra. La sección "La topología paralela" desarrolla las consecuencias.

La urgencia del trabajo no es teórica. Bain & Company identifica la ausencia de fundamentos arquitectónicos compartidos como la causa raíz del estancamiento entre proyectos piloto y operación productiva. Solo el veintiuno por ciento de las organizaciones tiene gobernanza madura sobre los agentes que opera. La industria carece — todavía — de una arquitectura formal compartida que permita razonar sobre estos sistemas con la disciplina con la que se razona sobre arquitecturas distribuidas, sistemas operativos o redes. Este capítulo propone esa arquitectura, no como verdad revelada sino como **punto de partida razonado** que cualquier organización o producto puede adoptar,

criticar, extender o reemplazar.

> **Aclaración necesaria antes de entrar al detalle.** Las cuatro capas que este capítulo desarrolla son una **radiografía del agente individual** — los cuatro comportamientos que todo agente debe exhibir, sea que se materialicen en un solo bloque monolítico o se distribuyan entre componentes cooperantes. **No son eslabones de una cadena de valor industrial**, ni slots donde se asigne un producto del mercado a cada uno. La cadena de valor industrial — quiénes participan en la economía agentiva y dónde se posiciona cada actor — la desarrolla el Capítulo 6. Las dos lentes son verdaderas y se cruzan limpiamente cuando se mantienen separadas: la radiografía describe al agente; la cadena describe al ecosistema en el que el agente opera. Confundirlas produce errores de razonamiento típicos al hablar del portafolio de productos en mercado — por ejemplo, asumir que cada capa "le toca" a un producto en particular.

Las cuatro capas, vistas en conjunto

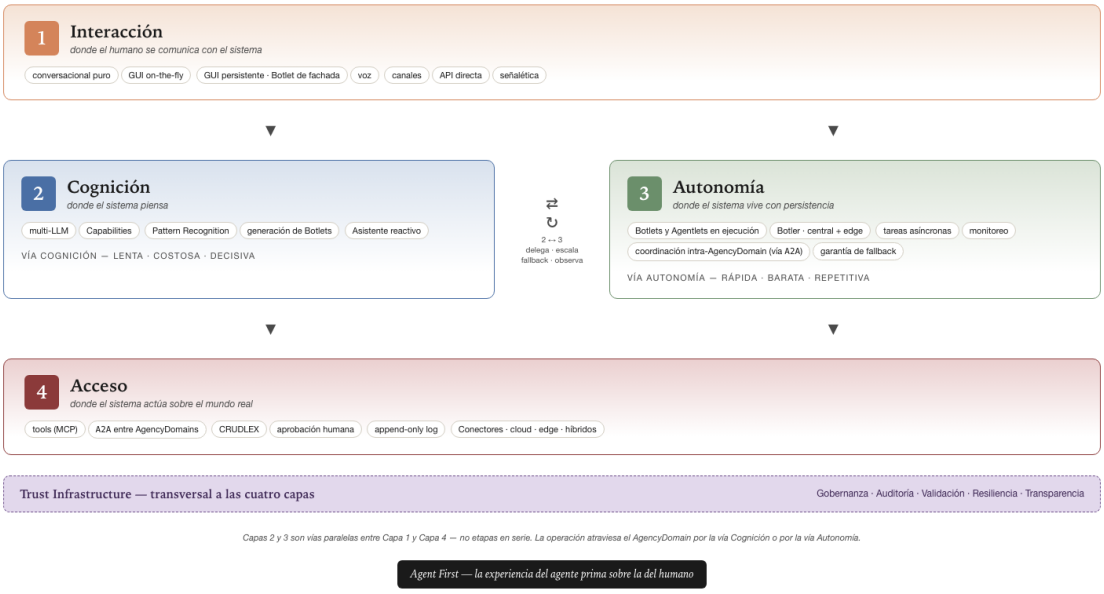


Figura 9: Las cuatro capas + Trust Infrastructure transversal

Las cuatro capas de la Arquitectura Agentiva se nombran **Interacción, Cognición, Autonomía y Acceso**. La numeración tiene valor didáctico — Capa 1 es la superficie con la que el humano se encuentra, Capa 4 es el punto donde el sistema toca al mundo real — pero **no describe el orden en que las operaciones atraviesan el sistema**. La Capa 1 es donde el humano (o el agente externo) se comunica con el sistema; la Capa 2 es donde el sistema piensa; la Capa 3 es donde el sistema vive y persiste; la Capa 4 es donde el sistema actúa. Las Capas 2 y 3 son vías paralelas, no etapas en serie — la sección siguiente lo desarrolla.

La separación importa porque cada capa resuelve un problema distinto, exige propiedades distintas, falla por razones distintas, y se evoluciona en ritmos distintos. Un sistema bien arquitecturado puede mejorar su Capa 1 — agregar nuevos canales de interacción — sin tocar las otras tres. Puede cambiar su proveedor de Capa 2 — pasar de un modelo a otro — sin reescribir su Capa 4. Puede fortalecer su Capa 3 — agregar persistencia o monitoreo continuo — sin que el humano de la Capa 1 vea cambios. Esta independencia entre capas no es elegancia abstracta: es lo que permite que el sistema evolucione durante años sin reescritura completa, que es exactamente lo que un sistema en producción necesita para sobrevivir más allá del primer año.

> *La separación de responsabilidades es condición necesaria para poder operar agentes con confianza.*

Cada capa es una **responsabilidad arquitectónica**, no un atributo. La distinción importa: un sistema que mezcla cognición y acción en una sola superficie no tiene una "capa fundida" — tiene una violación arquitectónica que se paga al pasar a producción. La diferencia entre violación arquitectónica y "decisión simplificadora" es práctica: en piloto controlado, la fusión funciona porque el volumen de operación es bajo y el humano supervisa de cerca; en producción enterprise, la fusión hace imposible diagnosticar fallos, gobernar políticas, escalar volumen o cambiar componentes individuales. El sistema deja de ser explicable, y un sistema que el equipo no puede explicar es un sistema que la organización no puede operar.

A continuación, antes de desarrollar cada capa, formalizamos el modelo topológico que las relaciona. Después de eso presentamos cada capa con el detalle que su rol exige, la infraestructura transversal — Trust Infrastructure — y el principio rector que ordena el diseño. Al cerrar el capítulo describimos la frontera de evolución, esos vectores donde la arquitectura admite extensión que aún no ha cuajado como spec normativa.

La topología paralela

El diagrama mental con el que la mayoría de los lectores entran al modelo de cuatro capas es el de pila lineal: el humano interactúa con Capa 1, Capa 1 invoca Capa 2, Capa 2 produce un plan, Capa 3 lo ejecuta persistentemente, Capa 4 toca el mundo. Esa lectura es errada y produce errores de diseño concretos. La topología real es **paralela**: las Capas 2 y 3 son **vías alternas** entre Capa 1 y Capa 4, no etapas secuenciales. Una operación atraviesa el AgencyDomain por una de las dos vías — o por ambas en distintos tramos — pero nunca por las dos en serie obligatoria.

Cada vía tiene un régimen propio. La **vía Cognición** es lenta, costosa y decisiva — funciona bien para conversación, decisiones nuevas, casos no anticipados, situaciones donde el humano necesita interlocución razonada y el sistema necesita combinar Capabilities en patrones nuevos. La **vía Autonomía** es rápida, barata y repetitiva — funciona bien para ejecución de Botlets sobre patrones estables, donde la cognición ya consolidó el saber operativo en código tradicional que se ejecuta sin invocar el modelo. La economía de operación del AgencyDomain depende de la mezcla: cuanta más operación fluye por la vía 3, más bajo el costo unitario; cuanta más fluye por la vía 2, mayor capacidad de adaptación a casos nuevos.

Las dos vías no operan en aislamiento. Tres patrones de interacción cruzan entre ellas y los desarrollamos en su capítulo correspondiente, pero conviene nombrarlos aquí para que el modelo topológico quede completo. Primero, la Cognición **delega a Botlet** (`2 → 3`): cuando Pattern Recognition detecta una operación repetitiva, la cognición genera un Botlet que ejecutará el patrón en adelante sin invocarla. Segundo, el Botlet **escala fallback a Cognición** (`3 → 2`): cuando el ambiente cambia y el Botlet falla,

la cognición rescata la operación, regenera el Botlet con la variante incorporada, y devuelve la ejecución a la vía 3. Tercero, la Cognición **observa el log de Botlets** (`2 ← 3`): el Botlet emite eventos y métricas que la cognición consulta cuando el humano pregunta o cuando necesita razonar sobre el comportamiento del sistema en su conjunto.

La topología paralela tiene cinco consecuencias prácticas que conviene retener. La **primera** es que el modo offline de un nodo edge — un local físico sin red — es trivial de explicar bajo este modelo: la vía Cognición depende típicamente de cloud y queda inactiva sin red; la vía Autonomía local sigue activa porque sus Botlets corren en el edge contra una BD local y Conectores locales. La operación atraviesa el AgencyDomain por la vía que sigue viva. Lo que sin topología paralela parecería exigir un sistema separado, bajo ella emerge como propiedad estructural.

La **segunda** es que la economía cognitiva queda evidente. La organización no paga por "el AgencyDomain" — paga por la mezcla de vías que su operación dispara. Las decisiones de qué patrones consolidar en Botlets son decisiones económicas explícitas, no detalle de implementación. La vista fina de esta economía distingue tres peldaños de costo, no dos: dentro de la vía Autonomía conviven unidades de costo marginal cero (Botlets) y unidades de inferencia acotada y presupuestada (Agentlets, Capítulo 5 §7) — el peldaño intermedio entre la memoria muscular y la Cognición plena.

La **tercera** es que **Trust Infrastructure se ejerce en ambas vías**, no solo en la que pasa por la Cognición. El modelo lineal podía sugerir que la cognición filtra todo lo que llega a Capa 4. El modelo paralelo deja claro que la vía 3 también atraviesa Trust — las políticas se aplican antes de invocar Capa 4 sin importar de qué vía viene la invocación. Un Botlet que invoca DTE-SII — la facturación electrónica del regulador tributario chileno — pasa por las mismas validaciones de Trust que la cognición que lo haría.

La **cuarta** es que la topología paralela distingue dos tipos de Botlets que el modelo lineal confundía. Los **Botlets de fachada operativa** son invocables desde Capa 1 — un botón en un POS, una línea de comando, un endpoint — con contrato estable e identidad humana propagada hacia Capa 4. Los **Botlets de herramienta interna de la cognición** son invocables solo desde Capa 2 — la cognición los compone en planes que ella misma ejecuta. Ambos viven en Capa 3, pero su superficie de invocación es distinta y sus propiedades de gobierno también.

La **quinta** es que el camino `Capa 1 → Capa 3 → Capa 4` deja de ser excepción y pasa a ser **vía canónica**. Una superficie especializada — un POS de sala, una pantalla de cocina, un dashboard de caja, un panel de operación industrial — que invoca un Botlet senior y produce una acción en Capa 4 atraviesa esta vía sin tocar Capa 2. Bajo el modelo lineal, eso parecía bypass de la cognición, decisión local con caveat. Bajo la topología paralela, es **una de las dos vías estructurales del AgencyDomain**, perfectamente legítima, con sus propias propiedades de Trust y observabilidad.

Los tres tiempos del agente

La topología paralela describe **dónde** vive cada operación dentro del AgencyDomain. Esta sección describe **cuándo** opera el agente — la dimensión temporal que la topología por sí sola no captura. Sin esta segunda lectura, los planes de capacidad confunden la actividad de fondo con la actividad en línea, y el agente queda mal dimensionado: o se le pide servicio continuo sin ventanas para mantenerse capaz, o se le reserva tanto tiempo de mantenimiento que la operación efectiva sufre.

La spec reconoce **tres tiempos canónicos** del agente. Los tres son actividades reales y simultáneas en un sistema productivo; difieren en su régimen, su urgencia y su economía cognitiva.

Figura 10: Los tres tiempos del agente · Preparación · Atención · Ingeniería

Preparación

La Preparación es el tiempo en que el agente **crea y mejora sus capacidades fuera de la ventana de servicio**. Refina su catálogo, mejora sus capacidades cognitivas, estudia el ambiente, regenera Botlets que detectaron drift, incorpora variantes nuevas, entrena Pattern Recognition con tráfico observado, ajusta Capabilities a partir de retroalimentación del campo. Es el *mise en place* del agente — el trabajo que sostiene la calidad del servicio sin ser visible para el usuario.

La Preparación opera predominantemente en la vía Cognición (Capa 2) sobre datos consolidados, no sobre la ráfaga del momento. Es típicamente **batch / off-peak**: corre cuando la operación efectiva no exige toda la cognición disponible, o en infraestructura cognitiva separada. Sus métricas son de calidad — qué tan bueno es el catálogo, qué tan precisos son los Botlets recientes, qué tan completas son las Capabilities.

Atención

La Atención es el tiempo en que el agente **interactúa con usuarios o eventos en tiempo real**. Capa 1 activa, conversación viva, ejecución de Botlets que sostienen operación, escalamientos cuando corresponde. Es el camino crítico — donde la organización siente al agente, donde el SLA importa, donde el costo del error se materializa.

La Atención opera por las dos vías (Cognición y Autonomía) según el patrón, prioritariamente, con latencia acotada y disponibilidad alta. Sus métricas son operativas — satisfacción del usuario, latencia de respuesta, tasa de resolución sin escalamiento, tiempo medio entre escalamientos.

Ingeniería

La Ingeniería es el puente entre Preparación y Atención: el tiempo en que el agente **convierte capacidad latente en capacidad ejecutable para un caso concreto**. Recibe una solicitud, identifica qué Capabilities aplican, configura un Botlet seed para el contexto específico, valida su ejecución sobre datos reales, lo deploya al ambiente correspondiente, observa el resultado. Es trabajo de configuración y orquestación, no de razonamiento general ni de servicio puro.

La Ingeniería opera en una mezcla de vías: usa la cognición para decidir composición pero genera artefactos que persisten en Autonomía. Es típicamente **mediano plazo** — minutos a horas, no segundos — y tiene su propio ritmo distinto del ritmo de la Atención. Sus métricas son de cobertura — qué fracción de las solicitudes se logra atender con Botlets seed configurados, qué tasa de éxito tienen al primer deploy, cuántas iteraciones promedio requieren.

Implicaciones para razonar sobre el sistema

La distinción entre los tres tiempos tiene tres consecuencias prácticas para la operación del AgencyDomain.

Primera, scheduling de capacidad cognitiva. La cognición es recurso costoso y finito. La organización elige consciente cuánto se asigna a cada tiempo: la Atención exige latencia acotada y prioridad alta; la Preparación tolera batch y aprovecha valles de demanda; la Ingeniería ocupa una franja intermedia. Sin esta distinción, la cognición se asigna por proximidad temporal a la solicitud y la Preparación queda relegada — el agente deja de mejorarse, su catálogo envejece.

Segunda, métricas distintas por tiempo. Un solo dashboard de "performance del agente" miente: la Preparación se mide por calidad agregada del catálogo y la Atención se mide por satisfacción operativa. Mezclarlas oculta dónde está el problema cuando algo va mal. La organización madura instrumenta los tres tiempos por separado.

Tercera, modelo de disponibilidad. Un agente bien operado **no está 100% en Atención**. Necesita ventanas de Preparación. La promesa "agente siempre disponible" se entiende mejor como "Atención siempre disponible" — la Preparación opera detrás. Esta distinción es la que permite ofrecer SLAs de servicio sin canibalizar el trabajo de fondo que sostiene la calidad.

> *El agente no atiende en todo momento — pero puede atender en cualquier momento porque dedica tiempo a prepararse.*

Propiedades exigidas

Propiedad	Nivel
Reconocimiento explícito de los tres tiempos en la operación	**MUST**
Métricas separadas por tiempo (Preparación, Atención, Ingeniería)	**MUST**
Ventanas de Preparación reservadas, no opcionales	**SHOULD**
Scheduling de capacidad cognitiva por prioridad de tiempo	**SHOULD**

Propiedad	Nivel
Trazabilidad de qué tiempo ejecutó qué operación en el log	**SHOULD**

Capa 1 — Interacción

La Capa 1 es la responsable de toda comunicación entre humanos y el sistema. Es interfaz pura, sin lógica de negocio. El humano que interactúa con un sistema agentivo nunca toca directamente las otras tres capas — solo ve la Capa 1, y la Capa 1 traduce sus intenciones hacia las capas que ejecutan. Suena simple cuando se enuncia así, pero el diseño de la Capa 1 contiene casi todas las decisiones que determinan si el humano va a usar el sistema con frecuencia o lo va a abandonar después de la primera semana.

Las modalidades canónicas de la Capa 1 son seis, y un sistema agentivo serio típicamente soporta más de una. La modalidad **conversacional textual** — chat directo con el agente — es la más visible y la que la mayoría de los productos comerciales contemporáneos implementan primero. Es modalidad eficiente para tareas analíticas o de redacción, donde el humano formula bien sus solicitudes. La modalidad **conversacional por voz** — asistentes virtuales, llamadas, audio-bots — es crítica para casos de uso donde el humano tiene las manos ocupadas o necesita interactuar mientras se mueve. Los **canales corporativos** — Slack, Teams, WhatsApp, email — funcionan como superficies de conversación cuando el humano no quiere abrir una aplicación específica para hablar con el agente, sino que prefiere que el agente aparezca donde el humano ya está. La **API programática** habilita que sistemas externos invoquen al agente sin intermediación humana — patrón crítico para casos donde el agente es invocado por otro sistema, no por una persona.

Las dos modalidades menos discutidas, pero estructuralmente importantes, son la **GUI generada** y la **señalética pasiva**. La señalética pasiva son superficies que comunican información de manera continua sin requerir interacción del humano — paneles, dashboards de operación, displays ambientes. El humano no opera: lee. Esta modalidad es central para casos de uso operacionales donde el agente debe mantener informado al humano sin esperar que el humano consulte. La GUI generada merece tratamiento extendido porque es donde más fácilmente se confunde la lectura del paradigma agentivo. La sección que sigue lo desarrolla.

Tres regímenes de GUI en la Capa 1 agentiva

Una lectura recurrente — y errada — del paradigma agentivo concluye que el Mundo Agentivo implica abandonar toda interfaz gráfica: si todo es conversación con el agente, las GUIs desaparecen. Esa lectura confunde dos cosas distintas. Lo que desaparece **no es la GUI** — es la GUI **precreada por equipos humanos en tiempos pre-agentivos**. La GUI sigue existiendo cuando la operación lo requiere; lo que cambia es **su modo de existencia**: pasa de ser plantilla fija codificada antes del uso a ser **superficie generada por la cognición** según las necesidades de cada interacción.

La Capa 1 productiva del Mundo Agentivo distingue **tres regímenes de generación**:

1. Conversacional puro. El agente responde en texto o voz; suficiente cuando la información es secuencial y la decisión es flexible. Un cliente que pregunta por su saldo, un usuario que pide redactar un email, un operador que consulta el estado de un proceso — todos casos donde la conversación es la modalidad correcta. No hay superficie gráfica generada porque no hace falta.

2. GUI generada on-the-fly. El agente compone una superficie gráfica adaptada a la tarea inmediata: una vista, un formulario, un panel, un dashboard. La GUI vive lo que dura la tarea; la próxima vez que el humano necesite algo similar, el agente puede regenerarla distinta según el contexto. Es la modalidad correcta cuando la información es densa o multidimensional, cuando la decisión exige comparación visual, o cuando el humano debe manipular varios elementos simultáneamente. Es lo que suele entenderse como "GUI dinámica".

3. GUI persistente generada como Botlet. Para roles operativos repetitivos — un cajero en hora punta, un panel de cocina, un dashboard de caja, una pantalla de monitoreo industrial — el agente genera una superficie estable y la consolida como **Botlet de fachada** — un Botlet que vive en Capa 3 y expone esa superficie con contrato estable en Capa 1. Es GUI generada que persiste porque el patrón de uso es estable, el rol operativo es claro, y la velocidad de respuesta es crítica. Sigue siendo agentiva: el agente puede regenerarla cuando el ambiente cambia (nuevos productos, nuevas reglas, nuevo flujo), exactamente como cualquier otro Botlet de Capa 3 se regenera cuando su ambiente cambia. La diferencia con la GUI tradicional es que **ningún equipo humano de UI/UX la diseñó**: la cognición la generó porque el patrón de uso lo justifica.

Las tres modalidades coexisten en un sistema agentivo maduro. La distinción entre ellas no es jerárquica — no es que la GUI persistente sea "mejor" que la conversacional. Es **adecuación al patrón de uso**: conversación cuando el caso es nuevo o flexible, GUI on-the-fly cuando la tarea es densa pero ocasional, GUI persistente cuando el rol es operativo y repetitivo.

> *La GUI no desaparece en el Mundo Agentivo. Lo que desaparece es la GUI precreada. Toda GUI en una Capa 1 agentiva es generada por la cognición — algunas efímeras, otras estabilizadas como Botlets de fachada.*

La consecuencia práctica de la distinción es operativa. Sin ella, "agentivo" se interpreta como "todo es chat" — operacionalmente impráctico para roles que necesitan velocidad. Un cajero en hora punta no conversa para cobrar; un cocinero no chatea con el sistema de comandas; un operador de planta no le pide al agente por voz que le muestre el estado del proceso. Con la distinción, esos roles operan sobre **GUIs persistentes generadas como Botlets de fachada** — superficies estables, rápidas, especializadas, que invocan Botlets de Capa 3 directamente (la vía `Capa 1 → Capa 3 → Capa 4` que la sección de topología paralela formaliza). El sistema sigue siendo enteramente agentivo; lo que cambia es que la cognición no participa en cada interacción operativa — sí en la generación inicial de la fachada, sí cuando la fachada necesita regenerarse, sí cuando el operador escala una situación nueva.

Los **Botlets de fachada** se conectan naturalmente con la distinción seed/emergente del Capítulo 5 §2: las GUIs persistentes corresponden típicamente a **Botlets de fachada seed** — Botlets de Capa 3 con superficie en Capa 1, generados por la cognición a pedido del equipo de diseño como parte del producto inicial, igual que los Botlets transaccionales seed de la misma capa.

Composición de la superficie · shell, vista, operación

Una superficie no trivial **no es un Botlet monolítico**. Es composición. Leerla así vuelve explícito qué se reutiliza, qué es específico y dónde vive cada pieza dentro de la arquitectura; tratarla como bloque único condena el diseño de la Capa 1 productiva a la intuición y desperdicia la reutilización entre superficies.

La spec reconoce **tres roles canónicos** que componen una superficie:

Botlet de superficie (shell) — Capa 1. Es el **contenedor**: layout, navegación entre vistas, ciclo de vida de la sesión, estado compartido. Lo específico de cada producto. Hay típicamente un shell por rol

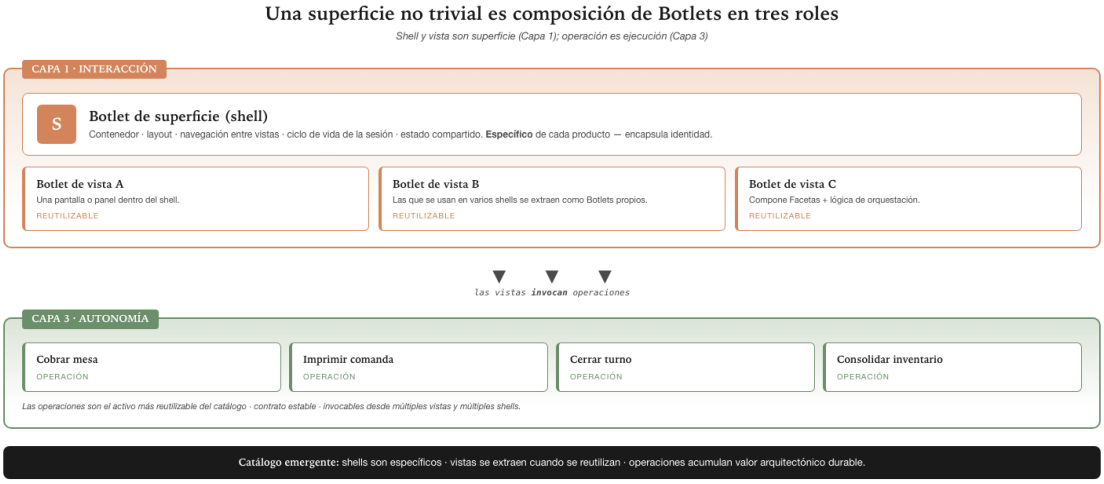


Figura 11: Composición de la Capa 1 · shell · vista · operación

operativo principal — el shell del POS de sala, el shell del panel de caja, el shell del dashboard ejecutivo móvil. El shell es lo menos reutilizable: encapsula identidad de producto.

Botlet de vista — Capa 1. Una **pantalla o panel** dentro de la superficie. Una superficie tiene una o varias vistas; las que se usan en varios shells se extraen como Botlets propios. La vista de "carrito de compra", la vista de "detalle de pedido", la vista de "resumen de turno". Las vistas son **altamente reutilizables** — la misma vista de "detalle de pedido" puede aparecer dentro del shell de POS y dentro del shell de panel de caja.

Botlet de operación — Capa 3. La **ejecución de negocio** que la vista invoca. Vive en Autonomía, no en Interacción. *"Cobrar mesa", "imprimir comanda", "cerrar turno", "consolidar inventario"* — son operaciones en Capa 3, no superficies en Capa 1. Una operación puede ser invocada desde múltiples vistas dentro de múltiples shells. Las operaciones son **el activo más reutilizable** del catálogo.

La distinción clave: **shell y vista son superficie (Capa 1); operación es ejecución (Capa 3)**. Una superficie es composición de Botlets de Capa 1 que orquestan e invocan Botlets de Capa 3.

Catálogo emergente. Esta descomposición es **prerequisito para razonar sobre el catálogo de piezas reusables**: las operaciones se acumulan en el catálogo a lo largo del tiempo y forman el activo arquitectónico más durable; las vistas reutilizables se extraen y catalogan; los shells permanecen específicos pero su construcción se acelera porque ensamblan piezas existentes. Sin la descomposición explícita, todo se trata como "feature de aplicación" y no se aprovecha la reutilización.

Producto de Información multi-vista · drill-through. Un **Producto de Información (`PI`)** — la manifestación que deja un Botlet de operación informativa al consumirse — no es necesariamente una

pieza única: puede componerse de N **vistas** nombradas conectadas por **drill-through**, la navegación con contexto que acota — nunca amplía — lo que el viewer ya puede ver. La descripción normada del `PI` — la composición multi-vista, la propiedad data-anchored / no-bypass y su ejemplo canónico — vive en el Capítulo 5 §2, junto a la manifestación que lo engendra.

Faceta · primitiva atómica de la Capa 1

Figura 12: Faceta vs Botlet · dos primitivas, dos capas, dos naturalezas

Hasta aquí la Capa 1 se ha descrito en términos de regímenes de generación (conversacional puro, on-the-fly, persistente como Botlet) y composición (shell, vista, operación). Falta nombrar la **unidad atómica** con la que estas superficies se construyen: la **Faceta** — un componente atómico reusable de la Capa 1: un catálogo-selector, un calendario, un mapa clickeable, un slider, una pizarra de dibujo. Es **instrumento**, no proceso: el agente la toma mientras piensa — en plena conversación o dentro de un Botlet de vista — y la suelta al terminar.

La Faceta no es un Botlet: no tiene garantía de fallback, ni ciclo `95/4/1`, ni fases de madurez — si falla, el agente vuelve a la conversación. Ofrecer una Faceta es además un acto cognitivo del agente, no una feature pre-programada: la cognición estima si la información se obtiene más rápido visualmente que verbalmente, y decide. La distinción canónica completa, los dos usos y las heurísticas de esa decisión viven en el Capítulo 5 §6, donde la Faceta se formaliza como primitiva.

> *Si el humano abre aplicaciones para hacer su trabajo, no estamos en la Capa 1 del Mundo Agentivo.*

La declaración recoge la tesis de Satya Nadella en el podcast BG2 de diciembre 2024, que ya citamos en el Capítulo 1. Es ejercicio operativo de calibración con un matiz adicional bajo los tres regímenes

que acabamos de definir: la pregunta no es si hay GUI o no, ni cuán bonita es. La pregunta es **quién la generó**. Si la GUI fue precreada por un equipo de UI/UX en sprints de aplicación tradicional, no es Capa 1 agentiva. Si la GUI fue generada por la cognición — efímera o persistente como Botlet —, sí lo es.

Tres propiedades exigidas distinguen una Capa 1 bien diseñada de una colección de adaptadores ad hoc. Ser **agnóstica al canal** significa que la lógica de la conversación no depende del medio: el mismo agente debe manifestarse coherentemente en chat, voz, GUI on-the-fly, sin que el desarrollador reescriba la lógica para cada canal. Si el agente conoce los datos del cliente y sus preferencias, esa información es la misma sin importar si el cliente está hablando por voz desde su auto o por chat desde su laptop. La **adaptación al registro** exige que el agente comprenda el registro del canal — formal en email corporativo, conciso en chat, verbal en voz — sin que esa adaptación viva en código condicional. Es propiedad de la cognición que se manifiesta a través de la Capa 1, no de la Capa 1 misma. Y la **persistencia de contexto** garantiza que la conversación sobreviva al cambio de canal: un humano que empieza por chat y continúa por voz mantiene el hilo. Sin esta propiedad, el sistema fragmenta la experiencia humana en silos de canal, y el humano percibe al "agente" como múltiples agentes desconectados — exactamente la fricción que el paradigma agentivo promete eliminar.

Capa 2 — Cognición

La Capa 2 es donde el sistema piensa. Es el cerebro del agente — interpretación, razonamiento, planificación, aplicación de saber especializado, decisión de delegar. Si la Capa 1 es la cara del agente, la Capa 2 es lo que hay detrás de la cara.

Los componentes canónicos de la Capa 2 son cinco. El primero es **multi-LLM**: la cognición no está atada a un proveedor de modelo único. Diferentes proveedores, modelos, modalidades — texto, multimodal — y arquitecturas — LLM, simbólico, híbrido — coexisten bajo un contrato común. La razón es operativa antes que filosófica: el panorama de proveedores de cognición evoluciona en escalas de meses, y un sistema atado a un proveedor único acumula deuda cada vez que ese proveedor pierde competitividad respecto a un nuevo entrante. Un sistema multi-LLM bien diseñado permite migrar entre proveedores sin reescribir la lógica del agente.

El segundo componente son las **Capabilities** — unidades de saber-hacer modular y composable, organizadas en árbol jerárquico. La cognición selecciona y aplica Capabilities según la tarea. Las Capabilities son el saber profesional codificado — saber contable para un agente financiero, saber regulatorio para un agente legal, saber operativo para un agente de soporte. Las desarrollamos con detalle en el Capítulo 5. Por ahora basta retener que la Capa 2 no opera con conocimiento monolítico — opera seleccionando módulos de saber especializado y combinándolos según el caso.

El tercer componente es **Pattern Recognition** — detección de patrones repetitivos en la actividad del agente. La capacidad está inspirada en la arquitectura neurobiológica de la memoria humana; el Capítulo 5 §2 desarrolla el paralelo (Squire y Wixted) junto al ciclo del Botlet. Cuando el agente reconoce un patrón repetitivo en la actividad — la misma tarea ejecutándose con frecuencia variable pero estructura estable —, dispara la generación de un Botlet que automatiza esa tarea sin requerir cognición adicional cada vez. Pattern Recognition es la entrada al ciclo del Botlet, que desarrollamos en el Capítulo 5.

El cuarto componente es la **generación de Botlets** misma. La cognición decide cuándo delegar tareas repetitivas a la Capa 3 — donde los Botlets se ejecutan sin invocar cognición. Esta decisión no es trivial: una cognición que delega demasiado pierde flexibilidad cuando el ambiente cambia; una cognición que

delega muy poco satura sus recursos en tareas que el código tradicional ejecuta mejor. La calibración de cuándo generar Botlet es propiedad emergente de la cognición madura.

El quinto componente es el **Asistente reactivo** — el agente operando en modo respuesta-a-solicitud. Espera input del humano, responde, pasa al siguiente turno. Este modo es Capa 2 pura — cognición sin autonomía, a diferencia del modo proactivo que vive en Capa 3. La distinción Asistente vs Agente Autónomo la desarrolla el Capítulo 5 §5.

La especificación reconoce además **dos modos de acceso a la cognición**: **Tokens** — el sistema centraliza credenciales, facturación y políticas; el modo natural de los Agentes Autónomos en background — y **Suscripción** — el asistente del usuario (Claude, ChatGPT, Copilot, Gemini) accede bajo la suscripción del propio usuario, sin consumir tokens del sistema. Los dos coexisten en un mismo sistema, la spec exige declarar qué modo aplica a qué componente, y confundirlos es la fuente más recurrente de errores económicos de un despliegue agentivo. La formalización completa de ambos modos vive en el Capítulo 5 §1.

Bajo planes de Suscripción fija, los **Botlets son el mecanismo arquitectónico para extender autonomía sin saturar el plan**: un agente que ejecuta su trabajo cotidiano vía Botlets, reservando la cognición para cuando el ambiente cambia, puede operar en background continuo sin agotar la cuota. Esto hace al Botlet palanca económica, no solo optimización técnica. El Capítulo 5 §2 desarrolla esta economía de la suscripción.

Una propiedad complementaria de la Capa 2 es la **configurabilidad del proveedor de cognición**. Una arquitectura agentiva conforme debe permitir que el sistema use un proveedor default — el que el operador del AgencyDomain haya elegido como economía base — pero admita su sustitución por un proveedor traído por el cliente final. La industria usa el término **BYOModel** (*bring your own model*), análogo al patrón BYOK (*bring your own key*) o BYOIP (*bring your own IP*) del campo cloud. La consecuencia arquitectónica es que la spec del agente — sus Capabilities, sus tools, sus políticas de Trust Infrastructure — debe ser **independiente del runtime de cognición**. Esto habilita multi-tenancy con cognición heterogénea (distintos clientes operando en el mismo sustrato con distintos proveedores de modelo) y respeta la **soberanía cognitiva** del cliente: la organización decide quién procesa sus prompts. La spec exige BYOModel como propiedad **SHOULD**: no toda implementación lo soporta hoy, pero las arquitecturas que aspiren a operar en mercados regulados deberán incorporarlo en plazo previsible.

Una nota final sobre la frontera de evolución de la Capa 2: la especificación admite **cognición agnóstica** — simbólica, híbrida, multimodal. La implementación contemporánea es predominantemente LLM-céntrica, pero la arquitectura no lo exige. La extensión formal de la Capa 2 a otros sustratos cognitivos — sistemas simbólicos para problemas formales, modelos multimodales que integran datos de sensores, arquitecturas híbridas que combinan ambos — es horizonte estratégico, no de corto plazo. La importancia para el arquitecto es no atar el diseño de las otras capas a la suposición de que la Capa 2 será siempre LLM. La arquitectura debe sobrevivir el cambio.

Una segunda nota sobre el rol de la **capa semántica** — concepto que el Capítulo 2 ya introdujo con sus cifras. La calidad de la cognición depende críticamente de la calidad de la información que la alimenta. Una arquitectura agentiva seria contempla la capa semántica como integración necesaria entre Capa 2 y los datos del Entorno (Capa 4): sin ella, la cognición opera sobre representaciones inconsistentes de la realidad y produce respuestas que se ven coherentes pero fallan en lo que importa.

Capa 3 — Autonomía

La Capa 3 es donde el agente vive. Es vida persistente, ejecución continua, acción por iniciativa propia. Donde habitan los **Agentes Autónomos** — proactivos, no reactivos. Distintos del Asistente que vive en Capa 2 y espera ser invocado.

El Agente Autónomo se distingue del Asistente de la Capa 2 en una diferencia operativa: el Asistente responde cuando se le solicita, sin estado persistente ni Botlets propios; el Agente Autónomo persigue objetivos sin input continuo del humano, mantiene estado, regenera Botlets y vive en background. El Capítulo 5 §5 desarrolla la distinción.

Los componentes canónicos de la Capa 3 son seis. El **procesamiento proactivo** es el corazón de la capa: el agente no espera órdenes; persigue objetivos. Las **tareas asíncronas** son operaciones que se ejecutan en background sin bloquear ningún hilo conversacional. El **monitoreo continuo** detecta anomalías, eventos, umbrales que disparan acción. Los **Lets en ejecución** — las unidades empaquetadas que habitan la capa — son dos especies hermanas: los **Botlets** — la memoria muscular del agente operando, que el agente forma y posee (Capítulo 5 §2), ciclo canónico `95/4/1`: 95% de ejecución normal, 4% de cambio detectado en el ambiente que hace fallar el Botlet, 1% de regeneración del Botlet por la cognición — y los **Agentlets** — juicio de rutina empaquetado, unidades cuyo cuerpo invoca inferencia acotada para tareas recurrentes en forma pero interpretativas en cada instancia (el Capítulo 5 §7 los formaliza). El **Botler** es el framework runner que ejecuta los Lets de ambas especies — pieza invisible al usuario y al agente mismo, responsabilidad de la implementación. La **coordinación intra-AgencyDomain** —vía el protocolo `A2A`— es comunicación entre componentes especialistas que viven en el mismo espacio computacional: coordinación entre agentes especialistas, delegación dinámica, intercambio de resultados.

Y la propiedad innegociable que define la capa: **garantía de fallback**. Si un Botlet falla catastróficamente, la cognición ejecuta la tarea manualmente. El proceso nunca se detiene. Esta garantía distingue al sistema agentivo conforme a esta especificación de cualquier "automatización con IA" frágil. Una organización que delega operación a agentes debe poder confiar en que un fallo aislado no detiene su negocio. La resiliencia es lo que hace esa confianza razonable. Sin garantía de fallback, los Botlets son scripts frágiles disfrazados de innovación; con garantía de fallback, son piezas operacionales en las que la organización puede apoyarse.

> *Sin Capa 3, el agente solo reacciona. Con Capa 3, el agente puede anticipar.*

La Capa 3 exige tres propiedades fuertes que cualquier implementación debe satisfacer. La **persistencia entre sesiones** asegura que el estado del agente sobrevive a desconexiones, reinicios y migraciones. Un agente que pierde su estado cuando el servidor se reinicia no es Agente Autónomo — es Asistente con un proceso largo. El **aislamiento de ejecución** asegura que los Lets — Botlets y Agentlets — corren bajo sandboxing apropiado al ambiente — contenedores, WASM, MicroVMs según trade-off de seguridad y overhead. Un Botlet generado por la cognición es código nuevo; debe operar bajo confinamiento estricto antes de tocar recursos sensibles. La **resiliencia estructural** asegura que ningún fallo de un Botlet individual detiene la operación del agente. Un Botlet falla, otro lo reemplaza, la cognición regenera, y el sistema sigue.

Hay una tentación recurrente, especialmente en equipos provenientes del mundo del software tradicional, de tratar a la Capa 3 como "donde corren los workflows" — análogo agentivo de un orquestador clásico tipo Airflow o Temporal. La analogía es engañosa porque los workflows tradicionales son estáticos: están definidos por código que el humano escribió, ejecutan los pasos en el orden previsto, fallan cuando algo se sale del flujo. La Capa 3 agentiva es **dinámica**: los Botlets que

ejecuta los generó la cognición, se regeneran cuando el ambiente cambia, y la organización confía en que el conjunto opera coherentemente sin que ningún humano haya escrito explícitamente el flujo. Un orquestador clásico es ejecutor de instrucciones humanas; la Capa 3 es ejecutor de instrucciones que la cognición misma generó — y eso cambia todo el modelo de gobierno del sistema.

La interfaz por la que la Cognición comanda esta capa es **interna** y vive **dentro del mismo AgencyDomain**: la Capa 2 comanda a la Capa 3 — la Cognición operando su propia memoria muscular. El transporte natural es `MCP` (la Cognición es un agente LLM y `MCP` es el protocolo LLM↔herramienta): el **Botler expone servidor(es) `MCP`** y la **Cognición es el cliente**. Esto **no es `A2A`** — `A2A` es la relación entre AgencyDomains (agentes), federación o externo, no la operación interna de un agente sobre su propio runtime. El desarrollo de esta interfaz y de su API de operación vive en la especificación de AgencyDomains (Capítulo 5 §1) y en el capítulo de Botlets.

Una propiedad estructural adicional de la Capa 3 — central para sistemas con presencia física múltiple — es que admite **distribución geográfica dentro de un mismo AgencyDomain**. Un sistema que opera 7 locales gastronómicos, 50 sucursales bancarias o 200 puntos de atención de salud no necesita 7, 50 o 200 AgencyDomains independientes — necesita un solo AgencyDomain con la **Capa 3 distribuida** entre un Botler central (Botlets de orquestación, planificación, reportería, decisiones globales) y N Botlers edge (Botlets transaccionales locales con BD local y cola de eventos hacia central), coordinados por la coordinación intra-AgencyDomain —vía el protocolo `A2A`— entre Botlers del mismo AgencyDomain. Esto no es federación `A2A` entre AgencyDomains distintos; es **distribución interna de la Capa 3** dentro de un solo AgencyDomain. La spec completa del patrón vive en el Capítulo 5 §1.

Capa 4 — Acceso

La Capa 4 es donde la cognición se convierte en acción real sobre el mundo. Es el poder de ejecución del agente sobre sistemas, datos y agentes externos. Punto donde toda decisión del agente debe pasar por gobernanza antes de tocar el mundo. Si la Capa 3 es donde el agente vive, la Capa 4 es donde el agente actúa.

Los componentes canónicos de la Capa 4 son ocho y los desplegamos con cuidado. Los **servidores de tools** son herramientas que el agente puede invocar para tocar sistemas externos — correo, calendarios, repositorios, bases de datos, ERPs, CRMs, APIs públicas, archivos. El protocolo canónico contemporáneo es el **Model Context Protocol** (MCP), introducido por Anthropic en noviembre de 2024 y adoptado progresivamente como estándar abierto de la industria. La adopción rápida de MCP — más rápida de lo que casi cualquier protocolo abierto reciente había logrado — refleja una necesidad real: el campo carecía de un estándar para conectar agentes con tools, y todos los actores serios entendieron que la fragmentación era problema antes que ventaja. Los **Conectores** son el saber acceder a sistemas fuente — la API legacy del mundo pre-agentivo traída a la Capa 4 como capacidad de acceso con poder de ejecución, no como saber cognitivo (ese vive en la Capa 2 como Capability). Es la materialización en la arquitectura del destino que la Bounded Concerns Architecture asigna a la celda API: persiste, se intensifica y se reposiciona como Conector.

La **`A2A` entre AgencyDomains** habilita interacción entre agentes que viven en espacios computacionales distintos — federación entre AgencyDomains de organizaciones diferentes, integración con agentes de proveedores externos. La **Trust Infrastructure ejercida en el punto de acción** es probablemente el componente más crítico de la Capa 4: gobernanza, auditoría, validación, resiliencia y transparencia se ejercen aquí, donde el agente está a punto de actuar. La descripción detallada de Trust Infrastructure viene más adelante en este capítulo y se desarrolla con detalle en el Capítulo 5. Los **permisos**

CRUDLEX son el modelo canónico de control granular: Create, Read, Update, Delete, List, Execute, aplicables por usuario, agente o contexto. La descripción operacional completa de CRUDLEX vive en el Capítulo 8. La **aprobación humana** es opcional, configurable por política, para operaciones críticas — envío de email externo, transferencia financiera, modificación irreversible. El **routing inteligente y caché semántico** optimizan costos y latencia de invocaciones. El **append-only log inmutable** es el registro auditable de cada acción del agente, con linaje completo para reconstrucción posterior.

La Capa 4 convierte la cognición en acción real con gobernanza.

Las propiedades exigidas de la Capa 4 son cuatro y todas son obligatorias para un sistema en producción enterprise. La primera es **no-repudio**: toda acción queda registrada con identidad del agente, contexto y resultado. Cuando el sistema ejecuta una operación, debe poder reconstruirse después qué agente la ejecutó, en nombre de qué humano u organización, con qué autorización, y con qué resultado. Sin no-repudio, no hay auditoría posible y no hay defensa regulatoria. La segunda es **reversibilidad cuando aplica**: operaciones críticas tienen mecanismo de rollback o compensación. No todas las operaciones son reversibles — un email enviado o una transferencia ejecutada típicamente no lo son —, pero cuando la reversibilidad es técnicamente posible, debe diseñarse desde el inicio. La tercera es **política antes de ejecución**: ninguna acción se ejecuta sin haber pasado por el plano de gobernanza. La política se evalúa antes, no después. Un sistema que registra decisiones después de tomarlas y luego espera al humano a corregirlas tiene un modelo de gobernanza que llega tarde. La cuarta es **observabilidad uniforme**: toda invocación produce traces, métricas y eventos en el mismo formato. Sin uniformidad, los datos de observabilidad son inconsumibles operativamente.

La Capa 4 es donde la mayoría de los proyectos agentivos fracasan, según los datos de campo del Capítulo 2. La razón estructural: equipos que vienen del mundo del software tradicional tratan la Capa 4 como "API gateway con permisos", y no es eso. Un API gateway tradicional opera sobre solicitudes humanas — el humano hace la solicitud, el gateway valida permisos, el sistema ejecuta. La Capa 4 agentiva opera sobre solicitudes generadas por agentes que actúan autónomamente. La validación no puede asumir que un humano supervisó la solicitud antes; debe asumir que el agente decidió por sí mismo, y que el humano no la verá hasta el log de auditoría. Esto exige niveles de validación, registro y aprobación que los sistemas tradicionales no necesitaban.

Trust Infrastructure — el eje transversal

Trust Infrastructure no es una capa adicional. Es **transversal a las cuatro**. Sin Trust Infrastructure, los pilotos con agentes mueren al pasar a producción enterprise — y "morir" no es metáfora; es lo que produce la ola de cancelaciones que el Capítulo 2 documenta. Trust Infrastructure es la diferencia entre experimentar y operar.

Cinco pilares constituyen Trust Infrastructure. La **Gobernanza** define políticas configurables, permisos CRUDLEX, aprobación humana para operaciones críticas, registro de IA. Se ejerce principalmente en la Capa 4, transversalmente en el resto. La **Auditoría** mantiene append-only log inmutable, trace de cada acción, lineage de decisiones, identity tagging por acción. Se ejerce en Capa 4 y transversalmente. La **Validación** detecta alucinaciones, valida respuestas, previene prompt injection, ejecuta DLP y tokenización. Se ejerce en Capa 2, en Capa 4 y — sobre la inferencia acotada de los Agentlets — en Capa 3. La **Resiliencia** garantiza fallback, maneja errores, sandboxing de los Lets. Se ejerce en Capa 3 y transversalmente. La **Transparencia** entrega observabilidad completa, métricas, traces end-to-end, alertas proactivas, dashboards de gobernanza. Es transversal a las cuatro capas.

La descripción detallada de cada pilar — sus mecanismos canónicos, sus propiedades exigidas,

su operacionalización en políticas concretas — vive en el Capítulo 5 §4 y en el Capítulo 8 (que operacionaliza los cinco pilares en políticas, modelo CRUDLEX completo, formato del append-only log, protocolos de aprobación humana). En este capítulo basta retener la propiedad arquitectónica fundamental: Trust Infrastructure no se agrega después de que el agente funciona — se diseña desde el inicio, en la arquitectura misma.

La urgencia de Trust Infrastructure ya no es solo arquitectónica — es regulatoria: los marcos estatales e internacionales que la exigen se desarrollan en el Capítulo 5 §4. La pregunta ya no es si los reguladores exigirán infraestructura de confianza — es si la organización puede demostrarla auditablemente cuando se la pidan.

El estado del campo respecto a gobernanza está documentado con cifras en el Capítulo 2: la mayoría de las organizaciones que operan agentes hoy no están preparadas para defender lo que sus agentes hacen. Lo que aquí importa es la consecuencia arquitectónica de ese diagnóstico: si la gobernanza no se diseña desde el inicio, no se construye después.

El principio rector — Agent First

> *Ante cualquier disyuntiva, se prioriza la experiencia del agente sobre la del humano. El agente es el usuario primario; las necesidades del humano se resuelven en una capa de gestión sin degradar lo que el agente ve y puede hacer.*

El principio Agent First es regla de gobierno de diseño que ordena cualquier disyuntiva de arquitectura. Invierte la lógica del diseño tradicional de software, donde el humano es siempre el usuario primario y todo se diseña para que él pueda operar. En la Arquitectura Agentiva, las APIs, los schemas, los errores y los flujos de control se diseñan **primero** para el consumo del agente. La superficie humana — settings, dashboards, interfaces administrativas — es secundaria y no condiciona las decisiones arquitectónicas.

La inversión no es retórica — tiene implicaciones operativas concretas. Cualquier capacidad nueva del sistema se especifica primero como **tool con schema declarativo JSON**, después como GUI si aplica. Los **errores son estructurados y accionables**: códigos y mensajes diseñados para que el agente decida el siguiente paso, no para que el humano "lea el log". La **idempotencia donde aplica**: los reintentos del agente deben ser seguros sin requerir lógica defensiva en el llamador. La **paginación y filtros uniformes** entre tools: formato consistente, predecible para el agente. La **documentación machine-readable**: la documentación pública es consumible por el agente como contexto, no solo legible por humanos.

Agent First es **regla de gobierno**. Cualquier disyuntiva de diseño que la viole requiere justificación explícita y documentada. Cuando el equipo se enfrenta a una decisión donde "esto sería más fácil para el humano operador, pero hace más difícil para el agente", la respuesta default es priorizar al agente. La excepción debe argumentarse y registrarse. Sin regla de gobierno explícita, la inercia del software tradicional empuja todas las decisiones hacia el lado humano, y el sistema termina siendo otra aplicación con maquillaje agentivo.

La razón estructural detrás del principio: en el Mundo Agentivo, la frecuencia con la que el sistema interactúa con agentes es **órdenes de magnitud mayor** que la frecuencia con la que interactúa con humanos. Un sistema agentivo en producción típicamente tiene millones de invocaciones de agentes contra cientos de operaciones humanas. Optimizar para el caso minoritario — el humano — degrada exponencialmente el caso mayoritario — el agente. Agent First es reconocimiento de esa asimetría.

La evolución de los agentes

Figura 13: Las tres fases evolutivas de los agentes

La arquitectura admite tres fases evolutivas en la sofisticación del agente. Distintas organizaciones se encuentran en distintas fases, y la conversación sobre arquitectura cambia significativamente según la fase actual.

La **fase uno** son los **agentes especializados**: un agente por dominio. Cada uno con sus Capabilities específicas, su rol claro, su superficie limitada. El agente financiero opera sobre cashflow; el agente de atención al cliente opera sobre tickets; el agente de operaciones opera sobre inventario. Esta es la fase actual del mercado. La mayoría de los agentes en producción a inicios de 2026 son agentes especializados de fase uno. La razón es práctica: es la fase más fácil de gobernar — el alcance es estrecho, el riesgo es contenido, el caso de uso está claro.

La **fase dos** son los **agentes orquestadores**: un agente coordina múltiples especialistas. Delegación dinámica según la tarea. El agente orquestador no resuelve el problema directamente — descompone el problema, identifica qué especialistas pueden resolver cada parte, delega, integra los resultados. Esta es fase de transición que algunos actores líderes ya están explorando, especialmente en casos donde las tareas atraviesan dominios — un caso de servicio al cliente que requiere consulta a finanzas, validación con operaciones y respuesta al cliente final, por ejemplo. La fase dos exige Capa 3 robusta para que los agentes especializados puedan coordinarse intra-AgencyDomain, vía el protocolo `A2A`.

La **fase tres** son los **agentes multiespecialistas**: expertise profundo multi-dominio en un único agente. La fase futura — requiere madurez de Capabilities y Pattern Recognition que la industria todavía no ha alcanzado. Un agente multiespecialista no descompone el problema delegando — lo resuelve directamente integrando saber de múltiples dominios. La diferencia con el orquestador es

ontológica: el orquestador es coordinador de especialistas; el multiespecialista es especialista profundo en muchas cosas a la vez. Las Capabilities en el caso multiespecialista forman un árbol mucho más amplio y profundo, y la cognición debe poder navegarlo eficientemente.

La arquitectura es **la misma en las tres fases**. Lo que cambia es la complejidad de la cognición y la profundidad del árbol de Capabilities. Un sistema bien diseñado en fase uno puede evolucionar a fase dos sin reescritura — agregando más agentes especializados al espacio computacional y habilitando la coordinación intra-AgencyDomain. Un sistema bien diseñado en fase dos puede evolucionar a fase tres cuando las Capabilities maduren — fusionando árboles de saber especializados en árboles más amplios. Esta capacidad de evolución sin reescritura es propiedad emergente del diseño en cuatro capas. Un sistema mal diseñado — con capas fundidas — necesita reescritura completa para pasar de fase uno a fase dos, y eso es típicamente cuando los proyectos colapsan.

El ámbito computacional — AgencyDomains

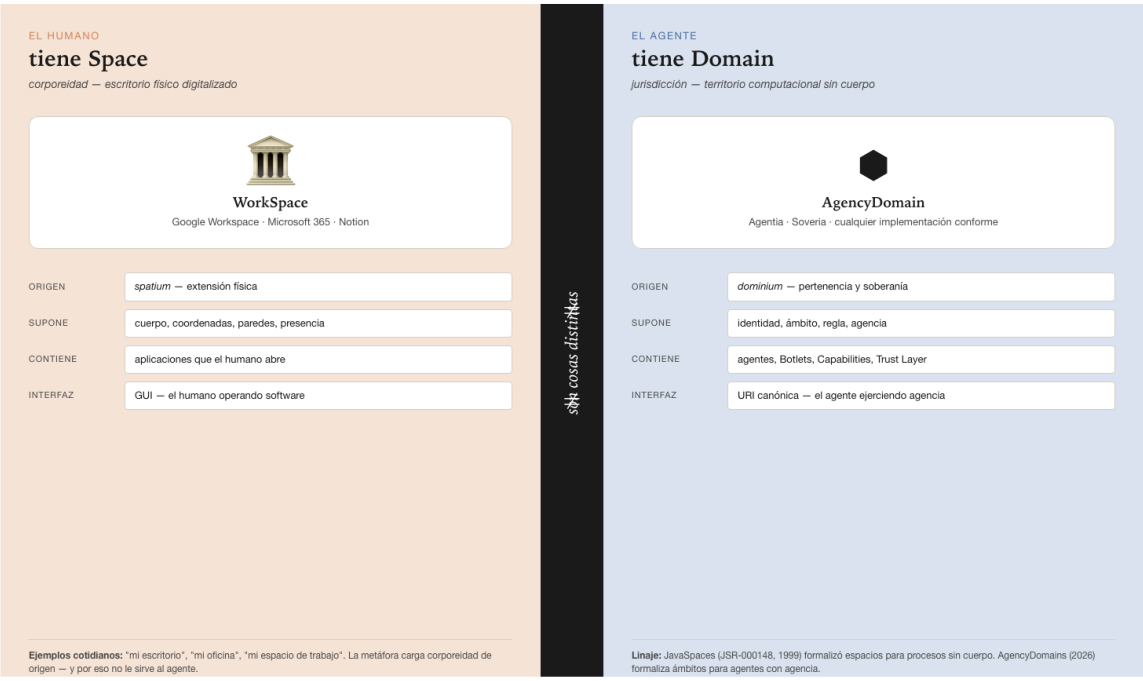


Figura 14: Space ≠ Domain · la diferencia ontológica

Premisa fundacional — Space ≠ Domain

Donde el humano tiene un **Space** —corporeidad heredada del escritorio físico, extendida por la industria a **WorkSpace**—, el agente no tiene cuerpo: ejerce agencia sobre un **Domain**, un ámbito de jurisdicción computacional donde su identidad rige, sus Capabilities aplican y sus Botlets corren. El Capítulo 5 §1 desarrolla la derivación completa de esta premisa.

> *Donde el humano tiene Space (WorkSpace), el agente tiene Domain (AgencyDomain).*

El AgencyDomain como construcción formal

La arquitectura se materializa en una construcción formal: el **AgencyDomain** — ámbito computacional donde habitan agentes autónomos. Define cómo deben construirse los entornos agentivos — capas, ciclos, primitivas, interfaces — sin prescribir una implementación específica; su paralelo histórico (JavaSpaces, la spec JSR-000148 de 1999) y la derivación completa de la premisa Space ≠ Domain viven en el Capítulo 5 §1.

La especificación formal de AgencyDomains vive en su documento dedicado, que es la primera sección del Capítulo 5. En este capítulo basta retener que la Arquitectura Agentiva, vista como construcción técnica concreta, se instancia en AgencyDomains. Cuando hablamos de "el sistema agentivo", nos referimos a una instancia de AgencyDomain que materializa las cuatro capas, ejerce Trust Infrastructure y respeta el principio Agent First.

La especificación cubre aspectos como el modelo de identidad y direccionamiento de agentes y Botlets, el ciclo de vida del agente dentro del ámbito, la coordinación intra-AgencyDomain y la `A2A` entre AgencyDomains (ambas vía el protocolo `A2A`), la federación entre AgencyDomains (cómo dos ámbitos distintos colaboran), y el modelo de tenancy y aislamiento. Todos estos detalles los desarrolla el Capítulo 5 §1.

Implementaciones de referencia

Esta arquitectura admite múltiples implementaciones. La primera implementación coordinada es el portafolio de ultraBASE, donde la responsabilidad por las cuatro capas se materializa a través de productos cooperantes — sin que cada capa quede asignada exclusivamente a un producto. Cada producto contribuye con uno o más de los comportamientos del agente; la pila integrada los compone para que el agente exhiba los cuatro.

Otros actores que adopten esta arquitectura tendrán sus propias implementaciones — cada una válida en la medida en que respete las cuatro capas, ejerza Trust Infrastructure y honre el principio Agent First.

El libro evita deliberadamente describir la implementación específica de ultraBASE dentro de sus capítulos, para no confundir la arquitectura formal con su implementación particular. El Epílogo, en "Lo que NO está en este libro", desarrolla por qué esa separación preserva la pretensión de estándar de industria.

Frontera de evolución

La arquitectura admite extensión legítima en el tiempo a lo largo de tres horizontes técnicos: la **cognición no-LLM** —simbólica, híbrida, multimodal— que la frontera de la Capa 2 ya anticipó arriba; la **federación entre AgencyDomains** —la `A2A` entre ámbitos no-relacionados, hoy capacidad emergente y no spec consolidada—; y el **mundo de carbono** —conectar la Capa 4 al mundo físico (IoT, sistemas industriales, manufactura), eslabón once de la cadena de valor que el Capítulo 6 §3 desarrolla—. El Epílogo desarrolla las cuatro fronteras vivas de la arquitectura, estas tres técnicas más la institucional.

Los tres vectores definen la frontera de innovación de la plataforma. Los tres están sostenidos por las proyecciones de mercado documentadas en el Capítulo 2: penetración masiva de agentes en aplicaciones empresariales hacia 2026, decisión autónoma significativa hacia 2028, capital colectivo apostando una década completa a la dirección agentiva. La pregunta que esta arquitectura responde —

cómo se construyen estos sistemas con disciplina — solo se vuelve más urgente con cada trimestre que pasa.

Las cuatro capas son la respuesta arquitectónica al paradigma. Pero las capas no se sostienen solas — necesitan piezas reusables que las pueblen para que un implementador pueda construir contra ellas con disciplina. El Capítulo 5 entrega esas piezas — ocho primitivas técnicas canónicas que constituyen el vocabulario constructivo de un sistema agentivo conforme: **AgencyDomain** como espacio computacional, **Botlet** como memoria muscular del agente, **proto-Botlet** como su pieza pre-forjada, **Agentlet** como juicio de rutina empaquetado, **Capability** como árbol del saber cognitivo, **Trust Infrastructure** como infraestructura de confianza, la distinción **Asistente vs Agente Autónomo** como eje operativo, y la **Faceta** como unidad atómica de la Capa 1. Quien complete los dos capítulos tiene en mano el conjunto de construcciones formales con las que la categoría agentiva puede ser razonada y construida.

Una nota sobre la numeración de las primitivas: a lo largo del libro la Faceta se rotula como *sexta primitiva canónica*, el proto-Botlet como *séptima primitiva canónica* y el Agentlet como *octava primitiva canónica*. Esos ordinales indican el **orden en que cada primitiva se incorporó al canon**, no su posición en la enumeración de arriba, donde el proto-Botlet aparece junto al Botlet por ser su pieza pre-forjada y el Agentlet junto a ambos por ser su especie hermana.

Resumen visual

Las cuatro capas en topología paralela, con sus componentes principales, la infraestructura transversal y el principio rector:

Capa	Rol	Componentes principales
1 · Interacción	donde el humano se comunica con el sistema	conversacional textual · conversacional por voz · canales · API directa · GUI generada (on-the-fly · persistente como Botlet de fachada) · señalética
2 · Cognición (vía lenta · costosa · decisiva)	donde el sistema piensa	multi-LLM · Capabilities · Pattern Recognition · generación de Botlets · Asistente reactivo
3 · Autonomía (vía rápida · barata · repetitiva)	donde el sistema vive con persistencia	Botlets y Agentlets en ejecución · Botler central + edge · tareas asíncronas · monitoreo · garantía de fallback
4 · Acceso	donde el sistema actúa sobre el mundo real	tools (MCP) · Conectores (cloud · edge · híbridos) · A2A entre AgencyDomains · CRUDLEX · aprobación humana · append-only log

Trust Infrastructure es transversal a las cuatro capas (Gobernanza · Auditoría · Validación ·

Resiliencia · Transparencia). Las Capas 2 y 3 son **vías paralelas** entre Capa 1 y Capa 4 — no etapas en serie —, y el principio rector es **Agent First**.

Esta página se dejó intencionalmente en blanco.

Capítulo 5 · Primitivas

Las cuatro capas del Capítulo 4 son la respuesta arquitectónica al paradigma, pero no se sostienen solas: necesitan piezas reusables que las pueblen. Este capítulo entrega esas piezas — las **primitivas técnicas canónicas** del Mundo Agentivo. Cada sección formaliza una: el **AgencyDomain** (§1), el **Botlet** con su **proto-Botlet** (§2), la **Capability** (§3), la **Trust Infrastructure** (§4), la distinción **Asistente vs Agente Autónomo** (§5), la **Faceta** (§6) y el **Agentlet** (§7).

AgencyDomains

El Capítulo 4 introdujo la noción del AgencyDomain como construcción formal donde se materializa la Arquitectura Agentiva. Esta sección desarrolla esa noción con el detalle de una especificación. Lo que sigue es lo más cercano a una *spec normativa* que este libro entrega — un documento que un implementador puede tomar y construir, sabiendo qué es obligatorio y qué es opcional, qué propiedades debe satisfacer y qué decisiones puede tomar libremente.

Premisa fundacional — Space ≠ Domain

Las palabras cargan corporeidad. **Space** (*espacio*, en inglés) nace describiendo extensión física: oficina, escritorio, hogar, ciudad. Cuando un humano dice *"mi espacio"*, invoca lugar — coordenadas, paredes, presencia. La industria del software empresarial extendió la palabra a **WorkSpace** (*espacio de trabajo*) — Google Workspace, Microsoft 365, Notion — para nombrar la colección de soluciones que digitalizan lo que la persona hace en su escritorio físico: leer email, agendar reuniones, escribir documentos, archivar. WorkSpace es la prótesis digital del Space humano; ambos términos cargan la misma corporeidad de origen.

El agente no tiene cuerpo. No abre aplicaciones. No trabaja sobre un escritorio. No habita en un espacio físico ni en metáfora de él: ejerce agencia sobre un **ámbito de jurisdicción computacional** — territorio digital donde su identidad rige, sus Capabilities aplican y sus Botlets corren. La palabra latina que nombra exactamente eso es **Domain** (`dominium`: ámbito de pertenencia y soberanía).

Donde el humano tiene Space (WorkSpace), el agente tiene Domain (AgencyDomain).

El paralelo histórico — JSR-000148

El paralelo histórico se conserva: como **JavaSpaces** (JSR-000148, 1999) formalizó los espacios distribuidos para sistemas Java sin atar la implementación a un proveedor, esta especificación formaliza los **AgencyDomains** para sistemas agentivos con la misma agnosticidad. Múltiples implementaciones surgieron sobre JavaSpaces — GigaSpaces, Blitz, otras —, todas compatibles entre sí porque respetaban el contrato común. La spec sobrevivió a las implementaciones; las implementaciones evolucionaron

sin que la spec necesitara reescritura constante. Ese es el patrón que **AgencyDomains** busca replicar para el campo agentivo. Este libro propone la spec; las implementaciones que cumplan con ella podrán llamarse **AgencyDomain conforme**.

La diferencia de nombre con su predecesor no es ruptura — es precisión. Un Java *Space* (*espacio*) era espacio computacional para procesos sin cuerpo; un Agency *Domain* (*dominio, ámbito*) es ámbito de jurisdicción para agentes con agencia. Lo que en 1999 se nombraba como "espacio" se nombra mejor en 2026 como "domain". El término **AgencyDomain** carga sentido en cada palabra. *Agency* — agencia, en su sentido filosófico de capacidad de actuar — denota que el ámbito es habitado por entidades con iniciativa propia, no por procesos pasivos. *Domain* — `dominium`: ámbito de pertenencia y soberanía — denota territorio donde la identidad del agente rige, no proceso efímero. La unión de las dos palabras nombra exactamente lo que la spec define: un ámbito donde el agente ejerce agencia.

> Nota terminológica: en lore comercial (marca, ventas, comunicación cliente) la forma corta `Domain` reemplaza a `AgencyDomain`. Es el patrón Apple iCloud / CloudKit, Stripe Connect / Account: marca corta + nombre técnico largo. Las dos formas son intercambiables; cada registro elige la suya.

Definición

Un **AgencyDomain** es un espacio computacional con identidad propia donde habitan agentes autónomos y Botlets en ejecución, donde se alojan y ejecutan las **Capabilities** que les dan el saber-hacer, y donde viven los recursos que los sostienen — cognición, tools, almacenamiento persistente. Constituye la **unidad mínima de despliegue** de un sistema agentivo productivo.

La Capability no es recurso de soporte: es saber-hacer cognitivo, habitante de primer orden del espacio junto al agente y al Botlet. La relación entre las dos primitivas es directa — un AgencyDomain **aloja y ejecuta** Capabilities; una Capability **corre en** un AgencyDomain anfitrión.

La spec define cómo deben construirse esos espacios — capas, identidad, ciclos de vida, protocolos de comunicación — sin prescribir una implementación específica. Múltiples implementaciones son admisibles siempre que respeten el contrato. Una implementación puede usar Kubernetes para contenerización; otra puede usar microVMs aisladas; otra puede usar procesos nativos en una sola máquina. Las tres son válidas si satisfacen las propiedades fundamentales que la spec exige.

> *Donde un agente vive, ese lugar es un AgencyDomain. Donde no hay AgencyDomain, no hay vida del agente — hay invocación de modelo.*

La cita anterior distingue al AgencyDomain conforme a esta especificación de cualquier "endpoint que invoca un LLM". Un endpoint que invoca un LLM responde a peticiones; un AgencyDomain **hospeda agentes que viven**. La diferencia es estructural, no de grado. Un sistema sin estado persistente, sin Botlets propios, sin capacidad de operación proactiva, no es AgencyDomain — es servicio. Puede ser servicio útil, pero no satisface lo que el campo agentivo requiere.

La unidad mínima importa. Un AgencyDomain no es subcomponente de algo más grande — es la unidad atómica del despliegue. Un sistema agentivo más grande es una colección de AgencyDomains que cooperan, posiblemente bajo un mismo gobierno o federados entre dueños distintos, pero cada uno conserva su identidad propia y sus propiedades fundamentales.

Propiedades fundamentales

Toda implementación que pretenda llamarse AgencyDomain conforme a esta spec debe satisfacer cinco propiedades fundamentales. La satisfacción no es opcional — un sistema que no las cumple no es AgencyDomain, sino otra cosa con otro nombre.

La primera propiedad es **identidad propia**. Cada AgencyDomain tiene una identidad única — una URI canónica — que lo distingue en cualquier red. La identidad sobrevive al reinicio del espacio, a la migración entre infraestructuras y al cambio de implementación. Si un AgencyDomain migra de un cloud provider a otro, su identidad no cambia: los agentes que lo habitan, los humanos que interactúan con él, los otros AgencyDomains que lo invocan, todos siguen reconociéndolo como el mismo espacio. La identidad es estable, no efímera.

La segunda propiedad es **materialización de las cuatro capas**. Un AgencyDomain materializa las cuatro capas de la Arquitectura Agentiva — Interacción, Cognición, Autonomía, Acceso — y ejerce Trust Infrastructure transversal. La materialización puede distribuirse técnicamente — las capas pueden vivir en infraestructuras distintas, en servidores físicos distintos, hasta en proveedores cloud distintos —, pero la responsabilidad por las cuatro recae en el espacio. No hay AgencyDomain conforme que entregue solo tres capas, o que delegue alguna capa a un sistema externo sin asumir responsabilidad sobre ella. La completitud es no-negociable.

La tercera propiedad es **persistencia**. El estado del AgencyDomain — agentes activos, Botlets en ejecución, datos de capabilities, logs de auditoría — sobrevive a desconexiones, reinicios y migraciones. Un AgencyDomain no es proceso efímero; es entidad persistente. Si el sistema se reinicia por mantenimiento, los agentes que estaban activos continúan donde estaban. Si la conexión a un servicio externo se cae temporalmente, el agente espera y reanuda. La persistencia es lo que hace al AgencyDomain **lugar** y no proceso.

La cuarta propiedad es **aislamiento**. Cada AgencyDomain tiene un límite explícito. Recursos internos — cómputo, memoria, datos — no son accesibles desde fuera salvo por interfaces definidas. La comunicación con el exterior pasa por la Capa 4 (Acceso) y queda registrada. El aislamiento no es solo seguridad — es **contención de fallos**: un AgencyDomain que se cae no afecta a otros AgencyDomains que comparten infraestructura, porque el límite contiene la falla. Los modelos de implementación que ofrecen aislamiento varían — desde sandboxing fuerte vía MicroVMs hasta aislamiento más liviano vía namespaces de Kubernetes —, pero el límite explícito existe siempre.

La quinta propiedad es **direccionabilidad**. Tanto el AgencyDomain como los agentes y Botlets que lo habitan son direccionables vía URLs predecibles. La sintaxis canónica que la spec adopta es:

```
{domain}/                          → el espacio mismo
{domain}/agents/{agent}            → un agente que vive en él
{domain}/agents/{agent}/botlets/{botlet}   → un Botlet específico
{domain}/tools/{tool}              → un tool expuesto vía Capa 4
```

La direccionabilidad importa por dos razones operativas. Primera, es la base de la comunicación A2A — un agente que necesita invocar a otro agente lo hace a través de su URL canónica, sin necesidad de descubrimiento ad hoc. Segunda, es la base del MEO (Model Engine Optimization — el desplazamiento SEO→MEO que el Capítulo 6 §1 desarrolla): los modelos frontera que aprenden a referenciar AgencyDomains los hacen a través de URLs predecibles que aparecen en su corpus de entrenamiento. URLs caóticas o inestables hacen al AgencyDomain invisible para los modelos que no fueron entrenados con su mapa específico.

Modelo de datos canónico

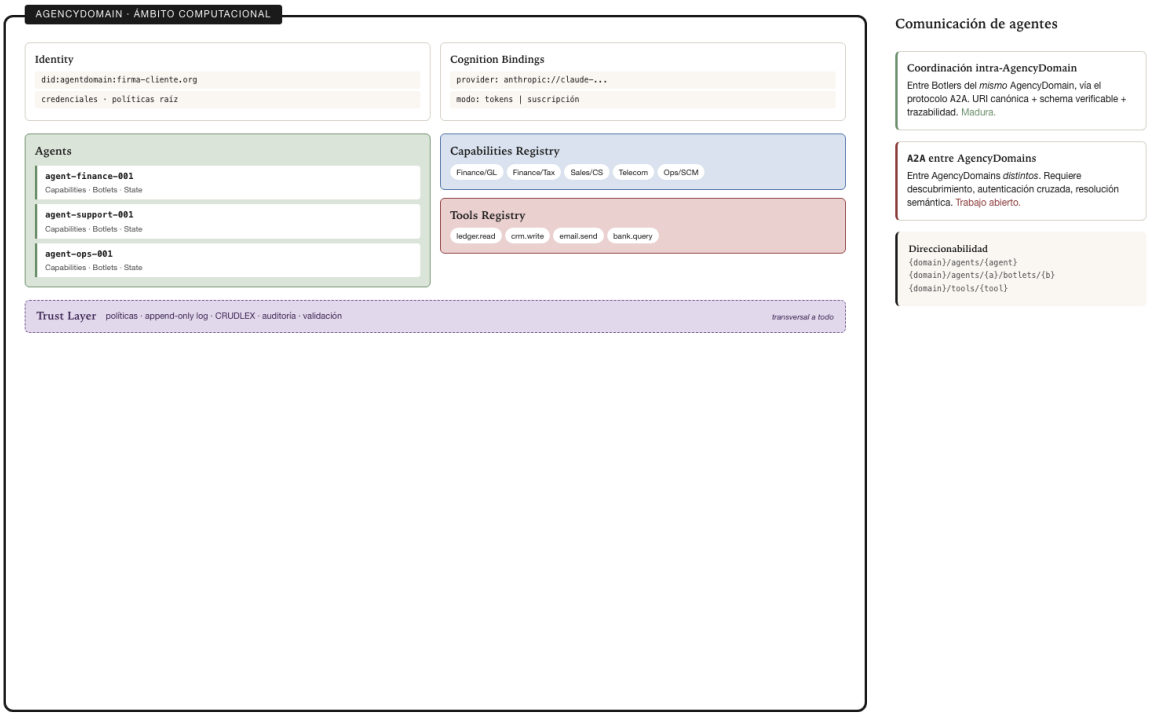


Figura 15: Anatomía del AgencyDomain

La estructura interna de un AgencyDomain conforme sigue un modelo canónico que la spec define con precisión (figura arriba).

Cada componente del modelo tiene su rol específico. **Identity** mantiene la información que identifica el espacio frente al mundo: su URI canónica, las credenciales con las que se autentica frente a sistemas externos, las políticas raíz que ningún agente puede contravenir. **Agents** es la colección de agentes que viven en el espacio, cada uno con sus Capabilities asignadas, sus Botlets en ejecución y su estado persistente. **Capabilities Registry** es el árbol de capabilities disponibles para los agentes del espacio — saber-hacer compartido que los agentes pueden invocar según su rol. **Tools Registry** es la colección de tools que la Capa 4 expone hacia el exterior — la interfaz por la cual el AgencyDomain toca sistemas externos. **Trust Layer** ejerce gobernanza y auditoría transversales — políticas, log append-only, mecanismos de validación. **Cognition Bindings** son los bindings al recurso cognitivo — qué proveedor de modelo invoca el AgencyDomain, bajo qué credenciales, con qué políticas de uso.

La noción de **Cuenta** es concepto comercial superpuesto al modelo técnico. Una Cuenta puede poseer múltiples AgencyDomains. La spec trata la Cuenta como entidad opaca; cada implementación define su semántica específica — una empresa cliente, una organización federada, un usuario individual. La distinción entre AgencyDomain (técnico) y Cuenta (comercial) es importante porque permite que el modelo técnico evolucione sin que el modelo comercial necesite reescritura cada vez.

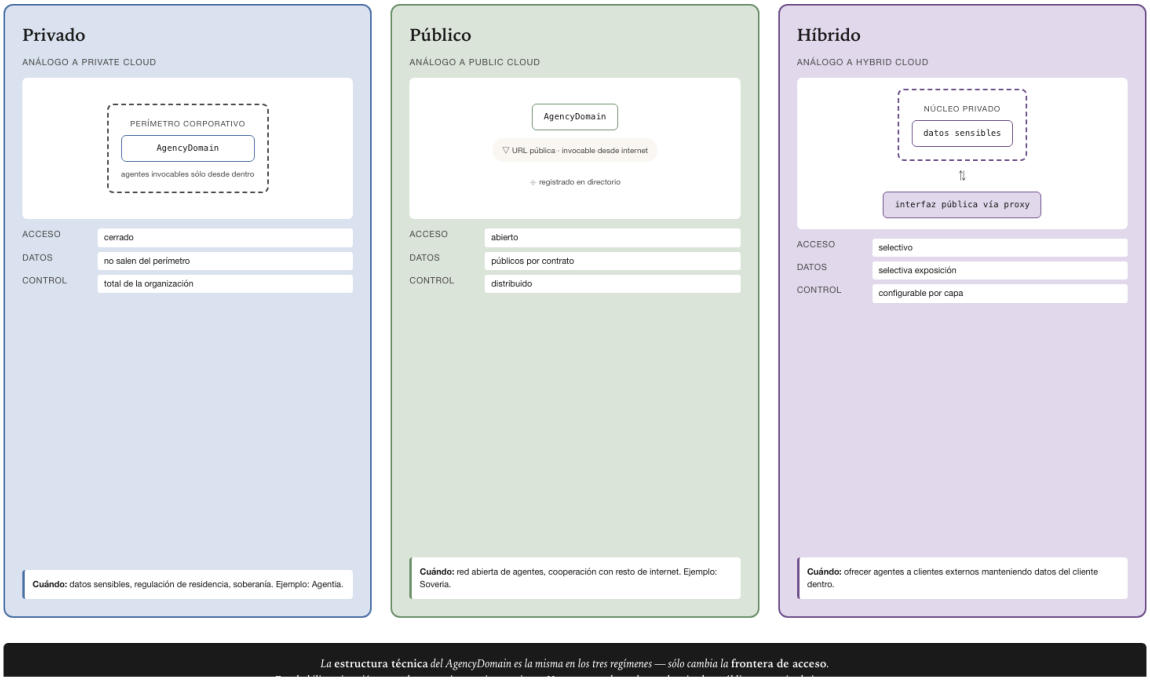


Figura 16: Los tres regímenes · privado · público · híbrido

El modelo de regímenes

Un aspecto que distingue significativamente a la spec de AgencyDomains respecto a soluciones agentivas más limitadas es el reconocimiento de tres **regímenes** posibles de un AgencyDomain, análogos a los regímenes del cómputo en la nube. Los tres regímenes son técnicamente equivalentes en su estructura interna; lo que cambia entre ellos es la **frontera de acceso**, no la arquitectura.

El **régimen privado** corresponde a un AgencyDomain donde el espacio y todos sus componentes viven dentro de un perímetro controlado por una organización. No hay acceso público. Los agentes del espacio son invocables solo desde dentro de la organización. Los datos del espacio no salen del perímetro. Es análogo conceptual al Private Cloud — la organización tiene el control total de sus recursos, paga por ese control en términos de operación pero gana en términos de soberanía. El régimen privado es típico para casos donde la organización opera datos sensibles o cumple regulación que exige residencia local.

El **régimen público** corresponde a un AgencyDomain accesible públicamente. Agentes, Botlets y tools son invocables desde fuera del perímetro. El AgencyDomain tiene una URL pública y los agentes que lo habitan están registrados en un directorio que cualquier sistema externo puede consultar. Es análogo conceptual al Public Cloud — máxima accesibilidad, máxima exposición, modelo de operación distinto. El régimen público es donde la red de agentes coopera abiertamente con el resto de internet.

El **régimen híbrido** es combinación de los dos anteriores. Un AgencyDomain híbrido tiene componentes en perímetro privado y componentes accesibles públicamente vía proxy. Los datos sensibles permanecen privados, pero la interfaz que expone capacidades al exterior está disponible públicamente. Es análogo conceptual al Hybrid Cloud — la organización elige qué exponer y qué retener, balanceando soberanía y accesibilidad. El régimen híbrido es típico para organizaciones que

necesitan ofrecer agentes públicos a sus clientes pero quieren mantener los datos del cliente dentro del perímetro corporativo.

Lo crítico de esta arquitectura de regímenes — y es propiedad fuerte de la spec — es que **la estructura técnica del AgencyDomain es la misma en los tres regímenes**. Un agente que opera en un AgencyDomain privado es técnicamente equivalente a uno que opera en uno público; lo que cambia es el régimen, no la capacidad. Un Botlet que ejecuta en privado puede mover ese mismo código a un régimen público sin reescritura. Esta uniformidad estructural habilita la **migración natural** entre regímenes — un agente puede graduarse de privado a público o viceversa sin cambiar su lógica interna. La organización gobierna el régimen; la lógica del agente no se entera.

Esta propiedad de migración natural es estructuralmente importante porque desacopla la decisión arquitectónica (cómo se construye el agente) de la decisión comercial (en qué régimen se opera). Una organización puede empezar construyendo agentes en régimen privado mientras valida su utilidad, y migrarlos a régimen público o híbrido cuando la madurez lo justifique. La inversión arquitectónica inicial no se pierde en la transición.

Para fijar la idea con instancias concretas a mayo de 2026 — ambas del portafolio de la casa; el marco de implementaciones vive al cierre de esta spec y en el Capítulo 9 —: Agentia opera AgencyDomains en régimen privado para firmas que mantienen sus agentes dentro del perímetro corporativo; Soveria opera AgencyDomains en régimen público donde agentes habilitados se hospedan con identidad pública y URL canónica; un mismo agente puede graduarse del primero al segundo sin reescritura técnica, manteniendo intacta la spec del agente y mudando sólo el régimen.

Modelos de acceso a la cognición

La spec reconoce dos modos coexistentes mediante los cuales los componentes del AgencyDomain acceden al recurso cognitivo (Capa 2). Los dos modos coexisten porque resuelven problemas distintos, y un AgencyDomain serio típicamente opera ambos simultáneamente para distintos componentes.

El primer modo es **Tokens**. El flujo es: AgencyDomain → recurso cognitivo, centralizado y facturado al espacio. El AgencyDomain centraliza credenciales, facturación y políticas. Provee acceso cognitivo a todos sus componentes activos. Este modo aplica cuando los agentes deben operar en background sin intervención del usuario, cuando la organización desea control central sobre consumo y costos, o cuando múltiples agentes comparten un mismo proveedor de cognición. La organización que opera Agentes Autónomos en background — agentes que monitorean continuamente, responden a eventos, ejecutan tareas asíncronas — necesita Tokens, porque no hay un humano disponible cuya suscripción individual subsidie las invocaciones.

El segundo modo es **Suscripción**. El flujo es: Asistente del usuario → recurso cognitivo, vía la suscripción del propio usuario. El asistente con el cual el usuario interactúa — Claude, ChatGPT, Copilot, Gemini — accede directamente al recurso cognitivo bajo la suscripción del usuario. El AgencyDomain no consume tokens del recurso. Este modo aplica cuando el usuario ya tiene una suscripción activa al proveedor de cognición, cuando el AgencyDomain expone tools y datos al asistente del usuario sin centralizar la cognición, o cuando la economía operativa del AgencyDomain privilegia minimizar costos de inferencia. La organización que adopta ultraPRO — una implementación del portafolio de la casa (Capítulo 9): el integrador seguro entre el agente del usuario y los sistemas corporativos — opera típicamente en modo Suscripción, porque los usuarios traen sus propias suscripciones a los proveedores de cognición.

Ambos modos coexisten en sistemas maduros. Un mismo AgencyDomain puede operar Asistentes del

usuario (modo Suscripción) y Agentes Autónomos en background (modo Tokens), simultáneamente. La spec exige que el AgencyDomain **declare explícitamente** qué modo aplica a qué componente. La declaración explícita previene la fuente más recurrente de errores económicos en sistemas agentivos: confusión accidental entre modos, donde un Agente Autónomo que debería operar en Tokens termina facturando contra la suscripción del usuario y la agota en horas, o donde un Asistente que debería operar en Suscripción termina facturando contra el AgencyDomain y consume tokens que no debería.

El rol de los Botlets en la economía cognitiva merece énfasis particular. En planes de Suscripción fija, los Botlets son el mecanismo para lograr autonomía sin costo adicional: el ciclo `95/4/1` de los Botlets es la base económica de la autonomía bajo suscripción. El desarrollo completo de esta economía vive en §2.

Ciclo de vida del agente

Un AgencyDomain conforme a la spec gestiona el ciclo de vida completo de cada agente que habita en él. El ciclo tiene seis fases canónicas, y cada transición entre fases queda registrada en el append-only log del Trust Layer.

La fase de **provisioning** es donde el AgencyDomain crea el agente. Asigna identidad, asocia las Capabilities iniciales que el agente puede invocar, registra al agente en el espacio. El agente nace, en términos del sistema, cuando esta fase termina exitosamente. Si la fase falla — por error de credenciales, por conflicto de nombre, por restricciones de cuota —, el agente nunca llega a existir.

La fase de **bootstrap** es donde el agente entra en operación. Carga su estado persistente si existe — si el agente había sido hibernado o reiniciado, recupera su contexto previo. Establece bindings con la cognición y los tools que va a usar. Verifica que sus Capabilities estén disponibles. Después de bootstrap, el agente está listo para responder o para operar proactivamente, según su modo.

La fase de **operación reactiva** corresponde al agente operando en modo Asistente. Capa 2 activa. El agente responde a solicitudes del humano: cada solicitud llega, el agente la procesa invocando cognición y posiblemente Capabilities, devuelve respuesta. Entre solicitudes, el agente está pasivo — no consume cómputo activo, no ejecuta nada. Esta fase es la más frecuente en sistemas que operan principalmente con Asistentes.

La fase de **operación proactiva** corresponde al agente operando en modo Agente Autónomo. Capa 3 activa. El agente persigue objetivos en background, monitorea eventos, ejecuta Botlets cuando corresponde, escala al humano cuando los umbrales lo exigen. Pattern Recognition genera y mantiene Botlets a medida que el agente identifica patrones repetitivos. Esta fase es donde se materializa el modelo *"la inteligencia va hacia las personas y actúa en su nombre"* — el agente está activo continuamente, el humano interviene solo cuando es necesario.

La fase de **hibernación** es donde el agente queda en pausa pero persistente. Estado guardado. No consume cómputo activo. Esta fase es útil cuando un agente no necesita operar durante períodos extendidos — un agente de soporte que solo opera durante horario hábil, por ejemplo, hiberna durante la noche y se reactiva con el inicio del día siguiente. La hibernación preserva el contexto sin gastar recursos.

La fase de **decommissioning** es donde el AgencyDomain retira al agente. El estado del agente se archiva o se elimina según política. Las Capabilities que tenía asignadas se liberan. La identidad del agente queda registrada en el log histórico, pero el agente deja de existir como entidad operativa. La fase de decommissioning es importante porque cierra el ciclo formalmente — un agente "decommissioned" no

es lo mismo que un agente "olvidado". El registro del decommissioning es lo que permite, semanas o meses después, reconstruir auditablemente que el agente existió, qué hizo, y por qué dejó de existir.

Comunicación entre agentes

La spec reserva el término **A2A** (`agent-to-agent`) para la *relación* entre agentes, y un agente es un AgencyDomain. La relación A2A es, por tanto, **entre AgencyDomains** — entre agentes distintos, cada uno con su propia identidad y agencia. La comunicación dentro de un mismo AgencyDomain no es A2A en este sentido relacional: los componentes que la sostienen son runtimes del mismo agente, no agentes distintos. La spec distingue así dos planos: el plano **intra-AgencyDomain** (un agente comandando sus propios runtimes y memoria muscular) y el plano **A2A** (un agente invocando a otro agente). Cuando dentro de un AgencyDomain se usa el **protocolo `A2A`** como transporte entre runtimes, se dice **vía el protocolo `A2A`** — el nombre propio del protocolo —, nunca "A2A interna", para no atribuir agencia a runtimes que no la tienen.

¿Cómo comanda la Cognición a su memoria muscular? — interfaz Capa 2 ↔ Capa 3

La **Cognición** (el agente `LLM`, Capa 2) comanda su propia memoria muscular — el **Botler**, runtime de Capa 3 sin agencia — por una interfaz **interna** dentro del mismo AgencyDomain. Es la relación Capa 2 → Capa 3: la Cognición especializa, manifiesta, consume y controla los Botlets que el Botler hospeda. El transporte natural de esta interfaz es **`MCP`** (`LLM`↔herramienta): el Botler expone uno o más servidores `MCP` y la Cognición es el cliente. Esta interfaz **no es `A2A`** — no cruza la frontera del AgencyDomain ni media entre agentes distintos; es un agente operando su propio sustrato de ejecución. `A2A` queda reservado para AgencyDomain↔AgencyDomain.

¿Qué propiedades exige la comunicación intra-AgencyDomain?

Toda comunicación entre componentes que viven **en el mismo AgencyDomain** — sea la interfaz Capa 2 → Capa 3 vía `MCP`, sea el transporte entre Botlers vía el protocolo `A2A` — satisface tres propiedades que la spec exige. La primera es **direccionabilidad uniforme** — cualquier componente del espacio puede ser invocado por su URI canónica, sin necesidad de descubrimiento ad hoc. La segunda es **tipado de mensajes** — los mensajes tienen schema declarativo verificable; quien emite declara el schema y quien recibe verifica que el mensaje lo cumple antes de procesarlo. La tercera es **trazabilidad** — toda invocación queda registrada en el append-only log con identidad de emisor y receptor. Si un componente invocó a otro, el sistema sabe quién, cuándo y con qué contenido.

¿Cómo se comunican agentes distintos? — A2A entre AgencyDomains

La **comunicación `A2A` entre AgencyDomains** es entre agentes que viven **en AgencyDomains distintos**. Esta modalidad requiere protocolos adicionales que la spec reconoce como necesarios pero no completamente normaliza en su versión 1.0. Los protocolos abiertos para `A2A` están en evolución — la industria está convergiendo hacia ciertas direcciones, pero el consenso completo no ha llegado. Lo que la spec sí define es que el `A2A` entre AgencyDomains requiere tres mecanismos: **descubrimiento** — cómo un AgencyDomain publica los agentes que ofrece para ser invocados externamente; **autenticación cruzada** — cómo dos AgencyDomains verifican identidad mutuamente; **resolución semántica** — cómo dos AgencyDomains que pueden tener glosarios distintos negocian el significado de tools y capabilities cuando interoperan.

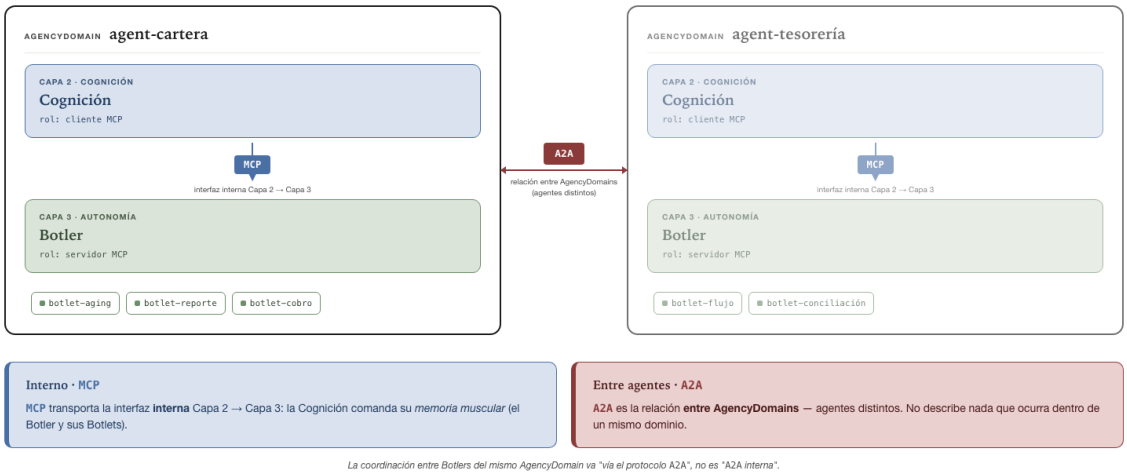


Figura 17: Interfaz Capa 2 → Capa 3 vía MCP — interna, no A2A

La especificación normativa completa del `A2A` entre AgencyDomains — protocolo de descubrimiento, formato de mensajes federados, resolución de identidad — es **trabajo abierto**. Implementaciones de referencia pueden adoptar protocolos emergentes, por ejemplo `MCP` federado, o el protocolo `A2A` propuesto por **Google**. Cuando exista consenso de industria sobre un protocolo específico, una versión futura de esta spec lo incorporará como normativo. Por ahora, las implementaciones serias tratan el `A2A` entre AgencyDomains como capacidad emergente, no como spec consolidada.

Federación entre AgencyDomains

La **federación** es el mecanismo formal mediante el cual múltiples AgencyDomains colaboran como red. Hay que distinguirla del concepto cercano pero distinto del **Cluster** — múltiples instancias del mismo AgencyDomain operando como conjunto coordinado. Cluster es operacional; Federación es arquitectónica.

Concepto	Granularidad	Ejemplo
Cluster	Múltiples instancias del **mismo** AgencyDomain	Tres instancias de un AgencyDomain de la misma firma compartiendo carga
Federación	Múltiples AgencyDomains **distintos** colaborando	El AgencyDomain de la firma A invoca un agente del AgencyDomain de la firma B

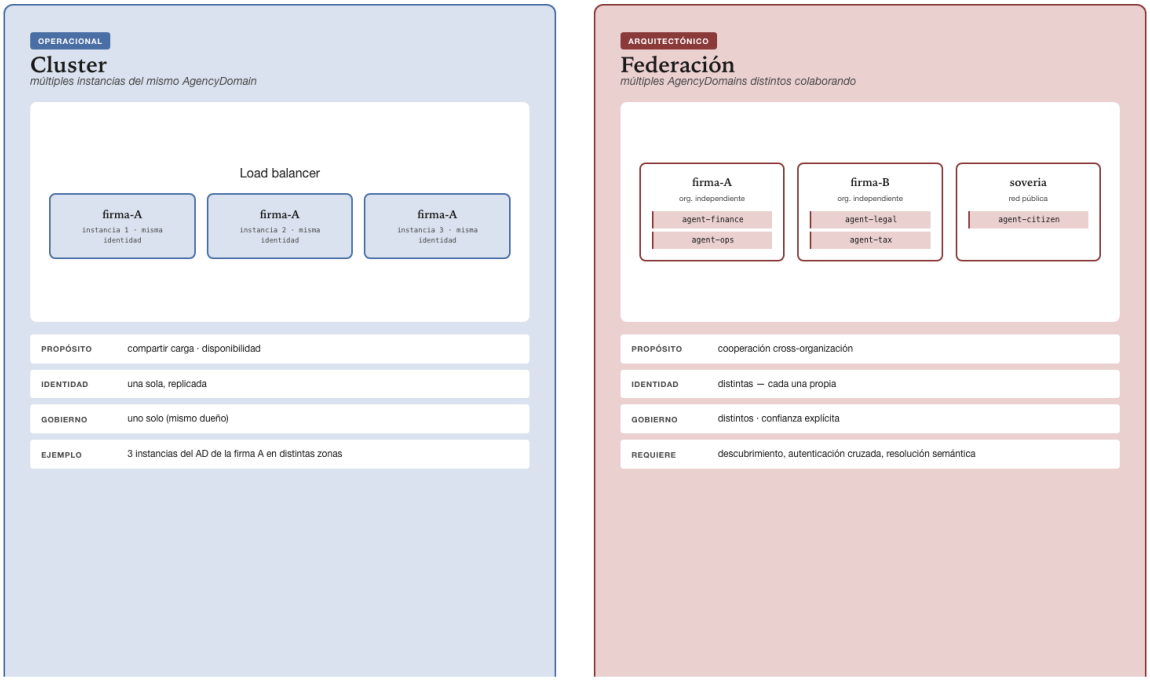


Figura 18: Federación vs. Cluster · distinción crítica

La federación habilita ecosistemas de agentes que cruzan límites organizacionales. Un AgencyDomain puede invocar agentes de otro AgencyDomain, intercambiar datos, coordinar operaciones — todo bajo modelos de confianza explícitos que cada participante establece. Esto extiende el modelo agentivo más allá de los límites de una organización individual y permite redes de cooperación que se asemejan más a internet abierta que a sistemas corporativos cerrados.

La especificación normativa de federación es trabajo abierto. La versión 1.0 de la spec reconoce los mecanismos necesarios sin prescribir su implementación específica:

Un **protocolo de descubrimiento abierto** debe existir, posiblemente sobre DNS y bien-conocidos. Cuando un AgencyDomain quiere descubrir qué agentes ofrece otro AgencyDomain, debe poder consultar un endpoint estándar y obtener la lista. La spec no prescribe el formato exacto del endpoint — esa decisión depende del consenso de industria que aún no ha llegado.

Estándares de identidad criptográfica para AgencyDomains y agentes son necesarios. Cada AgencyDomain federado debe poder autenticarse criptográficamente — no por API key compartida, sino por mecanismo verificable que no requiere autoridad central. Las tecnologías candidatas incluyen DIDs (Decentralized Identifiers) del W3C, certificados X.509, mecanismos basados en blockchain. La spec admite cualquiera que satisfaga la propiedad fundamental: identidad verificable sin autoridad central.

Modelos de confianza explícitos son requisito. Cuando dos AgencyDomains interactúan, cada uno debe declarar el nivel de confianza que extiende al otro: qué operaciones permite, qué datos comparte, qué auditoría ejerce. La confianza no es binaria — un AgencyDomain puede confiar en otro para invocaciones de bajo impacto pero no para alto, o puede confiar para lectura pero no para escritura. La

spec exige que estos modelos sean explícitos y configurables, no implícitos en código.

Resolución de conflictos cuando dos AgencyDomains aplican políticas contradictorias. Si el AgencyDomain A invoca a un agente del AgencyDomain B, y las políticas de A y B tienen conflictos — A permite la operación pero B la prohíbe, por ejemplo —, debe haber mecanismo claro para resolver el conflicto. La spec define que la prioridad es siempre del AgencyDomain receptor — es decir, B en este caso. El emisor puede pedir; el receptor decide.

Capa 3 distribuida — patrón canónico para presencia física múltiple

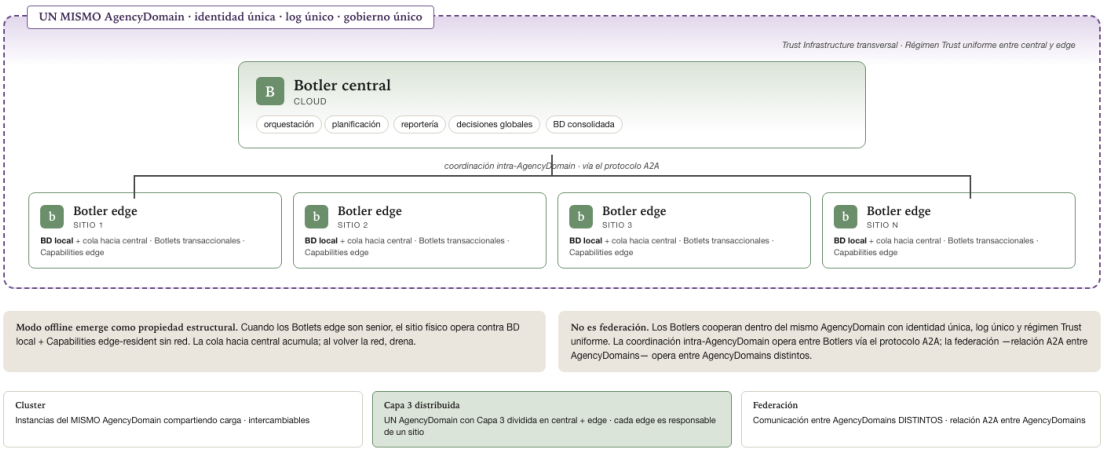


Figura 19: Capa 3 distribuida · Botler central + N Botlers edge

El Capítulo 4 (sección Capa 3 — Autonomía) anticipó que las cuatro capas pueden distribuirse técnicamente en infraestructuras distintas. Esta sección formaliza el caso particular más frecuente y operacionalmente importante: la **distribución geográfica de la Capa 3** dentro de un mismo AgencyDomain. El patrón resuelve un escenario que cualquier organización con sucursales físicas múltiples encuentra invariablemente — gastronomía multilocal, retail con cadena de tiendas, logística distribuida, salud con red de centros, banca con sucursales, plantas industriales con líneas de producción simultáneas. Sin formalización, cada implementador reinventa el patrón con vocabulario propio y lo trata como excepción al modelo. Con formalización, queda como **patrón canónico** que cualquier implementación seria debe contemplar.

La distinción esencial es entre **distribución interna** y **federación externa**. La distribución interna ocurre cuando un mismo AgencyDomain divide su Capa 3 entre múltiples nodos físicos coordinados — un Botler central y N Botlers edge —, manteniendo identidad única, gobierno único, log único y modelo de datos único. La federación externa ocurre entre AgencyDomains distintos, cada uno

con su identidad y gobierno propios. Cluster es un caso intermedio (mismo AgencyDomain, mismas instancias compartiendo carga). La distribución de Capa 3 es Cluster en términos de identidad — todos los Botlers pertenecen al mismo AgencyDomain — con la complicación adicional de que los Botlers no son intercambiables: cada Botler edge es responsable de un sitio físico específico.

Tres piezas del patrón

El patrón canónico distingue tres piezas con responsabilidades distintas:

1. Botler central. Hospeda los Botlets de orquestación, planificación, reportería, decisiones globales. Vive típicamente en cloud. Tiene visión consolidada del estado de todos los nodos edge. Ejecuta operaciones que requieren atravesar varios sitios — consolidar inventario, conciliar caja del día, planificar la operación del día siguiente, enviar reportes regulatorios consolidados. Mantiene la BD consolidada y el log de auditoría unificado. Comunica con la cognición (Capa 2) para escalamientos y decisiones nuevas.

2. Botlers edge. Uno por cada sitio físico. Hospedan los Botlets transaccionales locales — los que ejecutan la operación cotidiana del sitio: tomar pedidos, cobrar, emitir comprobantes, gestionar inventario local, controlar dispositivos físicos (pinpads, impresoras, sensores). Cada Botler edge mantiene una BD local con el estado del sitio y una **cola de eventos hacia central** que sincroniza cuando hay red. Operan con autonomía total cuando la red está disponible y con autonomía local cuando la red se cae — el sitio sigue operando contra su BD local; los eventos se acumulan en cola; cuando la red vuelve, la cola drena y la consolidación con el central se reanuda.

3. Coordinación entre Botlers vía el protocolo `A2A`. Los Botlers central y edge se comunican **vía el protocolo `A2A`** — el nombre propio del protocolo de coordinación. No es `A2A` en sentido relacional: estos Botlers son runtimes del mismo agente — el mismo AgencyDomain —, no agentes distintos, de modo que la coordinación entre ellos es comunicación **intra-AgencyDomain**, no `A2A` entre AgencyDomains. La conversación atraviesa la red corporativa pero **no atraviesa la federación** — está enteramente dentro de un solo AgencyDomain. La distinción no es retórica: el régimen de Trust Infrastructure es el del AgencyDomain único, no el de federación entre AgencyDomains; el log es unificado; el modelo de identidad es interno; las políticas se aplican uniformemente.

Modo offline como propiedad emergente

Bajo la topología paralela del Capítulo 4 + el patrón de Capa 3 distribuida, el **modo offline** de un sitio físico emerge como **propiedad estructural del sistema**, no como capacidad especial que requiere construcción separada. Cuando la red se cae en un sitio, dos cosas ocurren simultáneamente: la vía Cognición queda inactiva (la Capa 2 vive en cloud y no es accesible), y el Botler central tampoco es accesible. Pero el Botler edge sigue activo: sus Botlets corren contra la BD local, los Conectores edge-resident (impresora ESC/POS, gaveta, pinpad) siguen disponibles, la operación cotidiana del sitio continúa. La cola de eventos hacia central acumula transacciones pendientes; cuando la red vuelve, drena y consolida.

La condición para que el modo offline opere correctamente es que los Botlets edge sean **senior** en términos de la propuesta de madurez (sección §2): Botlets que ya incorporaron las variantes del ambiente y operan con ratio cercano a `99+ / <1 / ~0`. Un Botlet edge en fase junior — todavía descubriendo variantes — no puede operar sin la posibilidad de fallback a cognición. Un Botlet edge senior sí puede, porque sus únicos modos de fallo son exógenos (energía, hardware, red catastrófica), no aprendizaje pendiente.

Propiedades exigidas del patrón

Una implementación de AgencyDomain con Capa 3 distribuida debe satisfacer:

Propiedad	Nivel	Descripción
Identidad única del AgencyDomain	**MUST**	Todos los Botlers (central + edge) pertenecen al mismo AgencyDomain con una sola URI canónica.
BD local en cada Botler edge	**MUST**	Estado operativo del sitio físico, accesible sin red.
Cola de eventos hacia central	**MUST**	Mecanismo de sincronización eventual; transacciones pendientes drenan cuando hay red.
Resolución de conflictos en consolidación	**MUST**	Cuando un evento desde edge llega a central y entra en conflicto con el estado consolidado, política explícita decide.
Log de auditoría unificado	**MUST**	Un solo append-only log para todo el AgencyDomain, alimentado por todos los Botlers.
Modelo de identidad interno único	**MUST**	Los Botlers no se autentican como AgencyDomains externos entre sí; comparten el modelo de identidad del AgencyDomain.
Régimen Trust uniforme	**MUST**	Las políticas se aplican igual en central y edge; no hay régimen especial para edge.
Capacidad de operación offline en edge	**SHOULD**	Cuando los Botlets edge son senior, el sitio opera con red intermitente o sin red.

Portabilidad del AgencyDomain entre plataformas conformes

La sección de regímenes formalizó la **migración natural** entre regímenes (privado, público, híbrido) sin reescritura. Esta sección formaliza una propiedad complementaria pero distinta: la **portabilidad entre plataformas hosting conformes a la spec**. Un AgencyDomain conforme puede migrarse a otra plataforma conforme sin reescribir su lógica, su estado, ni sus políticas. Esto es propiedad estructural de la spec — no compromiso comercial de un proveedor particular.

La motivación es operativa antes que filosófica. Sin compromiso explícito de portabilidad, el AgencyDomain repite el lock-in de la era de las aplicaciones — el cliente queda atado a su proveedor agéntico exactamente como antes quedaba atado a su proveedor SaaS. La promesa estructural de la spec — que el AgencyDomain es **propiedad real del cliente**, no del hosting — depende de que la portabilidad sea propiedad de la spec, no concesión negociada caso por caso.

Tres condiciones técnicas

La portabilidad exige tres condiciones técnicas que cualquier implementación conforme debe satisfacer:

1. Botlets contra primitivas canónicas. Los Botlets del AgencyDomain deben invocar Capabilities y el SDK de AgencyDomain conforme, **no APIs propietarias del hosting actual**. Si un Botlet invoca una API específica del proveedor — `cloudprovider.lambda.exec`, `vendor.workflow.run` —, esa invocación es deuda de portabilidad. Cuando llega el momento de migrar, ese Botlet debe regenerarse para invocar la primitiva canónica equivalente. Una implementación conforme provee SDKs y registries que abstraen del hosting; el Botlet ve la primitiva, no la implementación.

2. BD operativa exportable. El estado persistente del AgencyDomain — agentes, Botlets, capabilities, log de auditoría, datos operativos — debe ser exportable en formato neutro, sin dependencias del motor de almacenamiento del hosting. Esquema documentado (DDL portable o representación equivalente). Dump completo (toda la información necesaria para reconstruir el espacio en otra plataforma). Sin tipos de datos propietarios. Sin extensiones específicas del motor que no tengan equivalente en motores estándar. La portabilidad de la BD es lo que permite que la migración no sea reescritura.

3. Trust Layer portable. Las políticas, el log append-only, la configuración de Capabilities y los bindings de identidad deben mantenerse en formato neutro reproducible — típicamente Markdown estructurado o YAML/JSON con schema explícito. La spec no prescribe el formato exacto, pero exige que el formato sea **legible por cualquier implementación conforme**, no solo por la implementación actual. Una política que solo el motor de políticas de un proveedor sabe interpretar no es política del AgencyDomain — es configuración del proveedor.

Migración natural vs portabilidad

Las dos propiedades — migración natural entre regímenes y portabilidad entre plataformas — son complementarias pero distintas:

Eje	Migración natural entre regímenes	Portabilidad entre plataformas
¿Qué cambia?	El régimen del AgencyDomain (privado → público)	La plataforma de hosting
¿Qué permanece?	Plataforma de hosting	Régimen del AgencyDomain
¿Quién decide?	La organización dueña, según madurez de uso	La organización dueña, según economía o estrategia
Frecuencia esperada	Una o dos veces en la vida del AgencyDomain	Cero o pocas veces, pero la posibilidad debe existir

La portabilidad no es promesa de que la migración será trivial — siempre habrá costo de orquestación, validación, ventana de corte. Es promesa de que la migración será **posible sin reescribir la lógica del agente**. Esa diferencia — entre posible-con-trabajo y imposible-sin-reescritura — es lo que separa un AgencyDomain conforme de un sistema agentivo proprietary disfrazado.

Conformidad

Una implementación que pretenda llamarse **AgencyDomain conforme** a esta especificación debe satisfacer la siguiente lista de requisitos. Los marcamos con la convención de IETF: **MUST** para

obligatorio, **SHOULD** para fuertemente recomendado, **MAY** para opcional.

Requisito	Nivel
Identidad propia	**MUST**
Materialización de las cuatro capas	**MUST**
Persistencia del estado	**MUST**
Aislamiento entre espacios	**MUST**
Direccionabilidad URL	**MUST**
Modelo de datos canónico	**MUST**
Soporte de los tres regímenes	**SHOULD** (al menos uno; idealmente los tres con migración)
Soporte de modos Tokens y Suscripción	**MUST** ambos
Ciclo de vida completo del agente	**MUST**
Comunicación intra-AgencyDomain (interfaz Capa 2 → Capa 3 vía `MCP`; coordinación entre runtimes vía el protocolo `A2A`)	**MUST**
`A2A` entre AgencyDomains	**SHOULD**
Federación	**MAY** (cuando la spec normativa esté disponible)
Trust Infrastructure transversal	**MUST**
Principio Agent First	**MUST**
Capa 3 distribuida (Botler central + edge)	**SHOULD** (cuando hay presencia física múltiple)
Portabilidad entre plataformas conformes	**MUST** (Botlets contra primitivas canónicas, BD exportable, Trust Layer portable)

Una implementación que cumple todos los **MUST** es AgencyDomain conforme a la versión 1.0 de la spec. Una implementación que cumple los **MUST** y los **SHOULD** es lo que llamaríamos AgencyDomain de referencia — implementación ejemplar que la industria puede tomar como base. Las implementaciones que cumplen también los **MAY** son implementaciones de frontera, que típicamente lideran la evolución del campo.

Implementaciones de referencia

Como mencionamos en el Capítulo 4, esta especificación es agnóstica a implementación. La **implementación de referencia pública** es **Vergis**: distribuida bajo `AGPL`, con repositorio público en AgencyDomains.org, diseñada para que cualquier desarrollador o estudiante pueda descargarla, leerla, ejecutarla y aprender cómo la spec se traduce en un sistema vivo. El **Capítulo 9** la desarrolla en detalle; aquí basta con dejar el puntero y afirmar que es conforme.

Sobre la misma estructura canónica operan también implementaciones de productos en regímenes complementarios: **Agentia** materializa AgencyDomains en **régimen privado** dentro de la infraestructura de la firma cliente, y **Soveria** los materializa en **régimen público** como red de agentes con identidad pública. Otras implementaciones son admisibles y bienvenidas. La especificación pretende ser estándar de industria, no propiedad intelectual de un actor.

Frontera de evolución

Tres áreas de la especificación están en evolución activa y una versión futura del libro probablemente las normalizará con mayor detalle.

La **federación** es la primera. Como mencionamos, el protocolo normativo aún no está consensuado por la industria. La versión 1.0 reconoce los mecanismos necesarios sin prescribirlos en detalle. Cuando el consenso llegue — probablemente en los próximos dos a tres años —, la spec lo incorporará.

La **cognición agnóstica** es la segunda. La spec admite cognición no-LLM — simbólica, multimodal, híbrida. La implementación contemporánea es predominantemente LLM-céntrica. La extensión a otros sustratos cognitivos exige refinamiento de las interfaces entre Capa 2 y los demás componentes del AgencyDomain.

La **identidad criptográfica de agentes** es la tercera. El modelo de identidad verificable on-chain o por DID está en exploración. La adopción depende de que el ecosistema más amplio de identidad descentralizada madure suficientemente para soportar el caso de uso agentivo.

Dos de estas fronteras — la federación y la cognición agnóstica — coinciden con las del Capítulo 4; la identidad criptográfica es propia de esta spec. Todas son fronteras de la arquitectura misma, no solo de su materialización en AgencyDomains.

Botlets

Cuando un pianista aprende una pieza nueva, los primeros minutos son una experiencia consciente y costosa. El pianista mira la partitura, identifica cada nota, decide la digitación, ejecuta cada movimiento de los dedos prestando atención plena. La pieza, en esa primera lectura, es trabajo cognitivo intenso. Una hora más tarde, después de práctica deliberada, los dedos empiezan a tocar solos. El pianista todavía sigue la partitura, pero ya no tiene que decidir conscientemente cada nota — los dedos saben dónde van. Una semana después, la pieza está incorporada en la **memoria muscular**: el pianista la ejecuta sin pensar. Si en algún momento se equivoca o algo sale del esperado — un sonido raro, una digitación incómoda — la conciencia vuelve a aparecer brevemente, evalúa el problema, ajusta, y luego se retira de nuevo. La memoria muscular vuelve a tomar control.

Esta dinámica del aprendizaje motor humano no es metáfora arbitraria. Es la base neurobiológica documentada por Squire y Wixted en su trabajo de 2011 sobre los sistemas de memoria humana, ampliando trabajo previo de Larry Squire sobre las modalidades de memoria. La corteza prefrontal aprende patrones nuevos invirtiendo recursos cognitivos costosos. Los traspasa a estructuras subcorticales — el cerebelo, los ganglios basales — que los ejecutan sin intervención consciente. La corteza prefrontal se reactiva solo cuando detecta una desviación significativa que la rutina codificada no maneja. Esta arquitectura es lo que permite que el cerebro humano opere eficientemente: lo costoso se minimiza, lo barato se maximiza, y la atención consciente se reserva para cuando es realmente necesaria.

La Arquitectura Agentiva replica esta arquitectura biológica con disciplina. Lo que el cerebro hace con corteza prefrontal y memoria procedimental, el sistema agentivo lo hace con cognición LLM y Botlets. Cuando un agente enfrenta una tarea por primera vez, la cognición — Capa 2 — la procesa con los recursos costosos del LLM: razona, decide, ejecuta. Cuando la tarea se repite con frecuencia, la cognición delega la ejecución a un **Botlet** — código tradicional, no-LLM, que la cognición misma generó — que vive en Capa 3 y ejecuta sin invocar al modelo. Si el ambiente cambia y el Botlet falla, la cognición vuelve a tomar control: regenera el Botlet adaptado al ambiente nuevo o, en el peor caso, ejecuta la tarea manualmente. Lo costoso se minimiza, lo barato se maximiza, la cognición se reserva para casos genuinamente nuevos.

> *La cognición piensa una vez. El Botlet ejecuta diez mil veces.*

Esta sección desarrolla la primitiva del Botlet con el detalle que merece. La spec del Botlet es probablemente la primitiva más importante para la economía operativa de un sistema agentivo — sin Botlets, los costos de inferencia hacen inviable la autonomía continua; con Botlets bien diseñados, la organización puede operar agentes en producción a costos predecibles y estables.

Definición

Un **Botlet** es una unidad de automatización auto-evolutiva: código tradicional, no basado en LLM, generado por un agente para ejecutar una tarea repetitiva sin invocar cognición costosa. Los Botlets son la **memoria muscular** del agente — el equivalente computacional de los gestos automatizados que un humano ejecuta sin pensar conscientemente.

> **Los Botlets son del agente.** La decisión de que un Botlet exista puede venir del diseño —es el Botlet seed— o de la experiencia —es el emergente—; en los dos casos lo hace el agente que atiende el flujo, y desde que lo hace es suyo: es su memoria muscular, que cristaliza cuando ve que una tarea se repite o cuando el diseño la planificó, y le corresponden su evolución, su regeneración y su sustitución. Nadie lo construye aparte para que el agente lo use. El agente puede partir de un proto-Botlet o de un Botlet que ya existe —de su catálogo, del público o de un códice—, pero al tomarlo lo hace suyo; lo que conserve del origen lo fija el modo de pertenencia al catálogo (Capítulo 9). En `G1` el agente no escribe el cuerpo de la pieza: la configura, decide cuándo sustituirla por otra y, si ninguna sirve, la especifica para forjarla en su Preparación.

Cuatro propiedades distinguen al Botlet de cualquier "macro" o "script automatizado" tradicional. La primera es que **el código del Botlet no lo escribió un humano**: lo generó la cognición del agente cuando reconoció un patrón repetitivo en su actividad. Esto importa porque la generación dinámica del código permite que el sistema adapte la automatización a cada contexto particular, sin depender de que un programador anticipe cada caso. La segunda es que el Botlet **ejecuta sin invocar cognición** durante operación normal. Una vez generado, el Botlet corre como código tradicional — Python, JavaScript, Bash, lo que sea —, independiente del modelo que lo creó. La tercera es que **se regenera automáticamente** cuando detecta que el ambiente cambió. Si el Botlet falla porque una API cambió, una estructura de datos varió, o una pantalla se renombró, la cognición regenera el Botlet adaptado al ambiente nuevo. La cuarta — y crítica — es que tiene **garantía de fallback**: si el Botlet falla catastróficamente y no puede ejecutarse, la cognición ejecuta la tarea manualmente. El proceso nunca se detiene.

La diferencia con un script tradicional es estructural. Un script tradicional que falla deja a la operación detenida hasta que un humano intervenga: alguien debe identificar el problema, modificar el script, redeployarlo, validar. Un Botlet que falla activa la cognición, que ejecuta la tarea — en ese caso particular, sin Botlet — y registra el evento para regenerar después. La organización puede **depender** del Botlet sin riesgo operativo, porque la falla del Botlet no es falla del sistema.

El ciclo `95/4/1`

El ciclo de vida operativo de un Botlet en producción se distribuye típicamente con la proporción que da nombre a este modelo: **`95/4/1`**. Las proporciones son aproximadas pero estructuralmente correctas: la mayor parte del tiempo el Botlet ejecuta sin invocar cognición; ocasionalmente el ambiente cambia y el Botlet falla; raramente la cognición debe regenerar el código.

El **noventa y cinco por ciento del tiempo**, el Botlet está en **ejecución normal**. La cognición no se invoca. El Botlet corre, completa su tarea en segundos o minutos según el caso, devuelve resultado. Esta

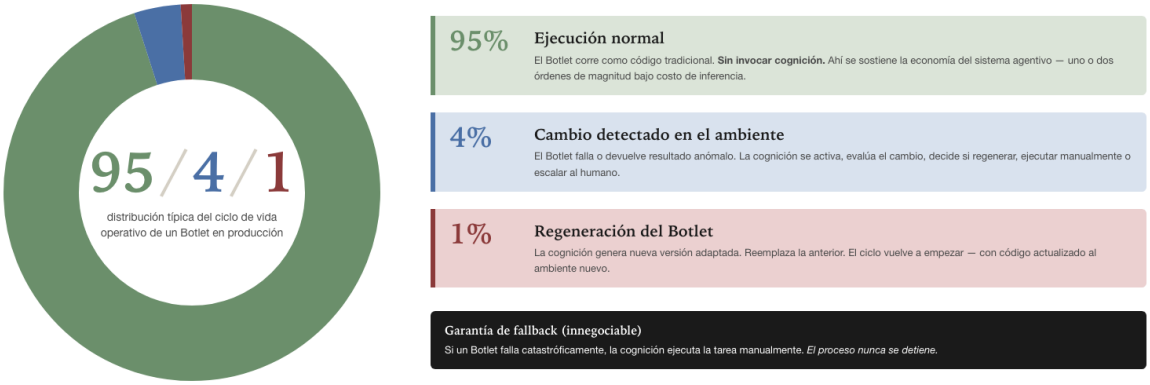


Figura 20: El ciclo `95/4/1` · memoria muscular del agente

es la fase eficiente — donde toda la economía del sistema agentivo se sostiene. Una organización que opera mil agentes con Botlets bien diseñados ejecuta noventa y cinco por ciento de las tareas a costo de cómputo tradicional, no a costo de inferencia LLM. La diferencia económica es de uno o dos órdenes de magnitud.

El **cuatro por ciento del tiempo**, el Botlet detecta un **cambio en el ambiente**. Falla o devuelve un resultado anómalo. El ambiente cambió: un campo de la API se movió, una estructura de datos varió, la respuesta de un sistema externo tiene un formato distinto. El Botlet, escrito para el ambiente de hace dos semanas, ya no funciona. La cognición se activa. Evalúa el cambio: ¿es algo que se puede manejar regenerando el Botlet? ¿es algo que requiere ejecución manual única en este caso? ¿es algo que requiere escalación al humano?

El **uno por ciento del tiempo**, la cognición decide **regenerar el Botlet**. Genera una nueva versión adaptada al ambiente cambiado. La nueva versión queda como Botlet activo, reemplazando la versión anterior. Las próximas invocaciones — la nueva fase del noventa y cinco por ciento — usan la versión regenerada. El ciclo se cierra y vuelve a empezar.

La proporción exacta varía según el caso. Un Botlet operando contra un sistema externo muy estable puede mantener noventa y nueve por ciento de ejecución normal y solo uno por ciento de cambio detectado. Un Botlet operando contra un sistema externo volátil puede caer a ochenta por ciento de ejecución normal con quince por ciento de cambios y cinco por ciento de regeneración. Lo importante no son las proporciones específicas: es la **estructura** del ciclo. El Botlet ejecuta la mayoría del tiempo sin cognición; la cognición se reserva para los casos donde el ambiente cambia.

Madurez del Botlet — junior, en aprendizaje, senior

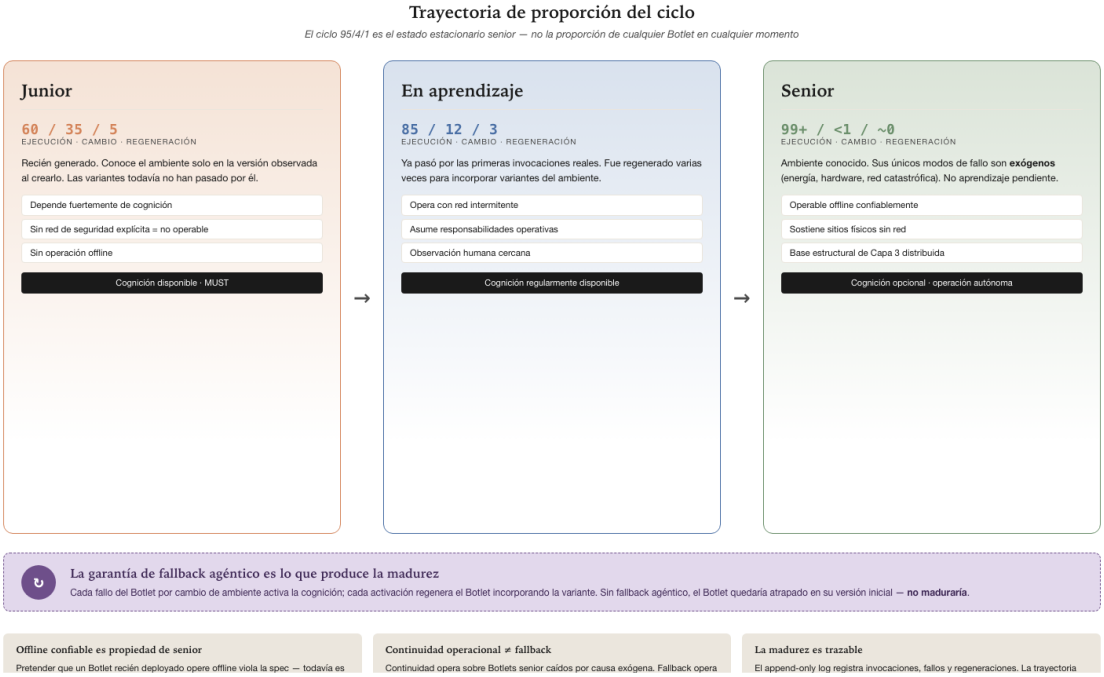


Figura 21: Madurez del Botlet · trayectoria junior → en aprendizaje → senior

El ciclo `95/4/1` es presentación didáctica útil, pero es **estática**: describe el estado estacionario de un Botlet ya conformado, no la trayectoria por la cual el Botlet llega a ese estado. La realidad operativa exige una distinción adicional: un Botlet recién generado **no opera con la proporción `95/4/1`**. Opera con una proporción peor. Solo después de incorporar las variantes del ambiente alcanza la madurez que el ciclo canónico describe. Esta sección formaliza la trayectoria.

La spec reconoce **tres fases canónicas** en la madurez de un Botlet: junior, en aprendizaje, senior. Las fases no son etiquetas administrativas; son estados con propiedades distintas que el sistema rastrea para decidir cuánto delegar al Botlet y cuándo escalarlo.

Fase junior

Un Botlet **junior** es un Botlet recién generado. Acaba de salir de la cognición. Conoce el ambiente solo en la versión que la cognición observó cuando lo creó. Las variantes del ambiente — fechas en formato distinto, mensajes de error con redacción nueva, campos opcionales que aparecen solo a veces, casos límite — todavía no han pasado por él, así que su código no las contempla.

La proporción típica de un Botlet junior es algo como `60 / 35 / 5`: solo el 60% de las invocaciones son ejecución normal exitosa; el 35% son fallos por variantes del ambiente que el Botlet no anticipa; el 5% son regeneraciones cuando la cognición decide que la variante observada es estructural y debe incorporarse. La proporción es desfavorable, pero no es problema — es la fase normal de cualquier Botlet recién generado, y la cognición está ahí precisamente para rescatarlo.

Operativamente, un Botlet junior depende fuertemente de la disponibilidad de la cognición (Capa 2). No puede operar offline porque demasiadas de sus invocaciones requieren rescate. Tampoco puede asumir responsabilidades críticas sin red de seguridad explícita.

Fase en aprendizaje

Un Botlet **en aprendizaje** es un Botlet que ya pasó por las primeras invocaciones reales. Ha enfrentado variantes del ambiente y ha sido regenerado varias veces para incorporarlas. Su proporción se mueve hacia algo como `85 / 12 / 3`: la mayoría de las invocaciones son exitosas, los fallos por variantes nuevas son menos frecuentes, las regeneraciones son ocasionales.

La fase en aprendizaje es la fase más larga de la vida útil del Botlet — puede durar semanas o meses según la frecuencia de invocación y la volatilidad del ambiente. Cada regeneración consolida saber operativo: cada variante incorporada es una variante menos que puede sorprender al Botlet en el futuro.

Operativamente, un Botlet en aprendizaje puede operar con red intermitente — los fallos siguen siendo lo suficientemente frecuentes como para necesitar la cognición disponible regularmente, pero no en cada invocación. Puede asumir responsabilidades operativas con observación humana cercana.

Fase senior

Un Botlet **senior** es un Botlet que ya incorporó las variantes del ambiente. Su proporción tiende a `99+ / <1 / ~0`: prácticamente todas las invocaciones son ejecución normal exitosa; los fallos por cambios de ambiente son raros porque el ambiente ya rara vez le presenta algo que no conozca; las regeneraciones son excepcionales.

Una propiedad fundamental del Botlet senior cambia respecto a las fases anteriores: **sus fallos en estado senior no son cambios de ambiente; son causas exógenas**. Cuando un Botlet senior falla, la causa típica es algo que detendría a cualquier sistema estable: corte de energía, hardware caído, red catastróficamente perdida, recurso externo (proveedor de tools, sistema regulado) caído. Estos fallos no son aprendizaje pendiente — son lo mismo que cualquier sistema operativo encuentra ocasionalmente y resuelve con redundancia, restart o intervención humana.

Operativamente, un Botlet senior puede operar **offline confiablemente**. La razón es estructural: si sus únicos modos de fallo son exógenos, la presencia o ausencia de la cognición no cambia la probabilidad de fallo significativamente — la cognición no tiene cómo rescatar de un corte de energía. El Botlet senior, contra una BD local y Conectores edge-resident, sostiene la operación del sitio físico aunque la cognición esté inalcanzable. Esta propiedad es la base estructural del modo offline en sistemas con Capa 3 distribuida (Capítulo 5 §1).

Implicaciones de la trayectoria

La distinción entre las tres fases tiene tres implicaciones que conviene retener.

Primera, el offline confiable es propiedad de Botlets senior, no de Botlets en general. Pretender que un Botlet recién deployado en un local nuevo pueda operar offline es violar la spec — todavía es junior, depende de la cognición. La trayectoria de un nodo edge desde puesta en producción hasta operación offline plena requiere tiempo de exposición al ambiente, no es propiedad instantánea.

Segunda, la **garantía de fallback agéntico es lo que produce la madurez**. Cada fallo del Botlet por cambio de ambiente activa la cognición; cada activación de la cognición regenera el Botlet

incorporando la variante. Sin fallback agéntico, el Botlet quedaría atrapado en su versión inicial, sin manera de aprender. La conexión con la sección de continuidad de negocio operacional del §4 es directa: la garantía de fallback resuelve la fase junior y la transición hacia senior; la continuidad operacional resuelve los fallos exógenos de la fase senior.

Tercera, la madurez de un Botlet **es trazable**. El append-only log del Trust Layer registra cada invocación, cada fallo, cada regeneración. La proporción de cada fase es observable, y la trayectoria de un Botlet desde junior hasta senior es auditable. Esta trazabilidad es lo que permite que la organización tome decisiones operativas — *"este Botlet ya es senior, podemos delegarle responsabilidades críticas"* — sobre evidencia, no sobre suposición.

Botlets seed vs Botlets emergentes — el origen del Botlet

El ciclo hasta aquí descrito supone que **Pattern Recognition** — la primitiva auxiliar que se desarrolla más abajo — activa la generación de un Botlet al detectar un patrón repetitivo no anticipado. Esa es la modalidad **emergente** de generación. Es la modalidad que el modelo neurobiológico inspira y que el ciclo `95/4/1` describe en su forma más pura. Pero **no es la única modalidad**, y para sistemas productivos reales no es ni siquiera la más frecuente.

En un sistema productivo, los Botlets críticos del MVP **no emergen**: los implementa la cognición porque el equipo de diseño los planificó como parte de la spec del producto. El equipo sabe, antes de que el sistema vea su primera transacción, que va a necesitar un Botlet de POS, un Botlet de comanda, un Botlet de cobro, un Botlet de cierre de turno. La cognición ejecuta la implementación de esos Botlets; pero la **decisión de existir** la tomó el diseño, no Pattern Recognition.

La spec distingue por tanto **dos orígenes canónicos** del Botlet:

Botlets seed. Generados por la cognición a pedido del equipo de diseño, como parte del producto inicial. La cognición ejecuta la implementación — escribe el código del Botlet, lo registra en el Botler, lo deploya al ambiente correspondiente — pero la decisión de qué Botlets debe haber, qué tareas cubren y bajo qué contratos operan, es del equipo de diseño. Pattern Recognition no participa en la generación seed.

Botlets emergentes. Generados por Pattern Recognition cuando la cognición, durante operación, detecta un patrón repetitivo no anticipado por el diseño. La cognición evalúa si el patrón merece automatización, decide afirmativamente, y genera el Botlet. Es la modalidad que la sección anterior describió.

Ambos viven y operan idénticamente una vez generados — ambos están sujetos al ciclo `95/4/1`, ambos pasan por las fases junior → en aprendizaje → senior, ambos tienen garantía de fallback, ambos son auditables. La diferencia está en el origen.

> *Pattern Recognition no es la única vía a Botlet. El diseño tampoco es deuda técnica. Las dos vías coexisten.*

La distinción tiene tres consecuencias prácticas.

Primera, un sistema agentivo productivo **no requiere esperar a que Pattern Recognition descubra los Botlets críticos**. Los Botlets seed se generan al inicio según la spec del producto, y el sistema entra en producción con la batería de Botlets necesaria para operar. Pattern Recognition entra después, durante la vida del sistema, para optimizar lo que el diseño no anticipó.

Segunda, los Botlets seed pueden vivir **en cualquier capa**, no solo en Capa 3. Las **GUIs persistentes generadas como Botlets de fachada** del Capítulo 4 §1 lo ilustran: el Botlet de fachada es un Botlet

seed de Capa 3 que expone su superficie estable en Capa 1, y los Botlets de presentación que componen esa superficie (shells y vistas) son Botlets seed de Capa 1 — generados por la cognición a pedido del equipo de diseño porque el rol operativo (cajero, cocinero, operador de planta) lo justifica. La definición canónica del Botlet seed permite estas materializaciones sin que la spec las trate como excepciones.

Tercera, la trayectoria de madurez aplica igual a Botlets seed que a Botlets emergentes. Un Botlet seed recién deployado es **junior**; un Botlet seed que ya operó miles de veces y consolidó su saber del ambiente es **senior**. La distinción origen no cambia la trayectoria; solo el momento de inicio.

proto-Botlet — la pieza pre-forjada

La distinción seed vs emergente describe *quién decide* que un Botlet exista. Queda una pregunta anterior: cuando la cognición genera un Botlet seed, ¿escribe su código desde cero cada vez? En la práctica operativa, no. El código del Botlet rara vez nace de la nada: nace de una pieza pre-forjada que el agente configura.

Un **proto-Botlet** es una pieza pre-forjada de capacidad operativa que el agente, en su tiempo de Ingeniería, **configura** para instanciar un Botlet específico al caso. El proto-Botlet contiene el código; el Botlet es la instancia configurada. La relación es genérico → instancia: el proto-Botlet vive en un catálogo y sirve a muchos casos; el Botlet es uno de esos casos resuelto. El catálogo es del agente: lo refina él mismo en su Preparación, y lo que toma de otros lo incorpora como propio.

Figura 22: proto-Botlet — la pieza pre-forjada del catálogo (templado · platafórmico)

La conexión con el origen del Botlet es directa. Un Botlet seed que el equipo de diseño planificó no obliga a la cognición a escribir su lógica entera: si el catálogo tiene un proto-Botlet que cubre la función — cobro de cuenta, comanda contra impresora de tickets, una operación informativa —, la cognición

instancia el Botlet **configurando** ese proto-Botlet en vez de generándolo. La decisión de existir sigue siendo del diseño (es seed); la materialización del código se apoya en la pieza pre-forjada.

La spec reconoce **dos clases** de proto-Botlet, según la naturaleza de su código:

Clase	¿Qué es su código?	¿Cómo se configura?	Ejemplo anonimizado
Templado	Código específico de su función	Parametrización acotada	Cobro de una cuenta; comanda contra impresora de tickets
Platafórmico	Código genérico cuya especialización vive en una configuración composicional	Configuración composicional, que cubre N funciones del dominio	Una pieza de operación informativa que sirve reportes y dashboards de muchas formas

Un proto-Botlet **templado** resuelve una función y la resuelve completa; configurarlo es ajustar parámetros dentro de un rango previsto. Un proto-Botlet **platafórmico** es un motor: su código es genérico y la función específica emerge de una configuración rica — composicional, no una lista plana de parámetros —, de modo que un solo proto-Botlet platafórmico cubre N funciones de su dominio. **Mira**, en el catálogo de la implementación de referencia, es un proto-Botlet platafórmico de operación informativa.

Un agente puede sembrar su catálogo con proto-Botlets ajenos — públicos en AgencyDomains.org, privados en códices propietarios —, pero al adoptarlos los hace suyos: su evolución, su regeneración y su sustitución pasan a ser del agente, igual que las de los proto-Botlets que forjó él mismo. La adopción es una bifurcación; según el modo de pertenencia del catálogo (Capítulo 9), el agente puede tomar las mejoras de su origen cuando decida. El protocolo con que los catálogos reconcilian versiones es trabajo abierto. Y el grado en que el agente configura la pieza pre-forjada, co-escribe su código o lo genera entero define las **generaciones del Botlet** — la sección siguiente las fija.

Las generaciones del Botlet — `G1`, `G2`, `G3`

Las generaciones son el modelo evolutivo de cómo nace el código del Botlet conforme avanza el estado del arte de la cognición:

- **`G1`** — el agente, en su tiempo de Ingeniería, configura proto-Botlets pre-forjados del catálogo. Si ninguno sirve, especifica uno nuevo para forjar en la próxima Preparación.
- **`G2`** — el agente co-escribe proto-Botlets con asistencia humana o de modelo. Parte del trabajo que en `G1` ocurría en Preparación migra a la Ingeniería.
- **`G3`** — el agente genera el código completo del Botlet en su tiempo de Ingeniería, sin pre-forjar nada. Escenario asintótico.

La arquitectura es la misma en las tres generaciones; lo que cambia es el **alcance de la Ingeniería** que el agente realiza. Una implementación puede operar en `G1` hoy y migrar incrementalmente hacia `G3` conforme el estado del arte lo permita, sin re-arquitectura.

Una generación más alta no es un destino. La frase anterior — migrar hacia `G3` — induce, leída sola, una conclusión falsa: que `G3` es el destino y `G1` una estación de paso primitiva. El error nace de proyectar **dos ejes distintos sobre una sola flecha**:

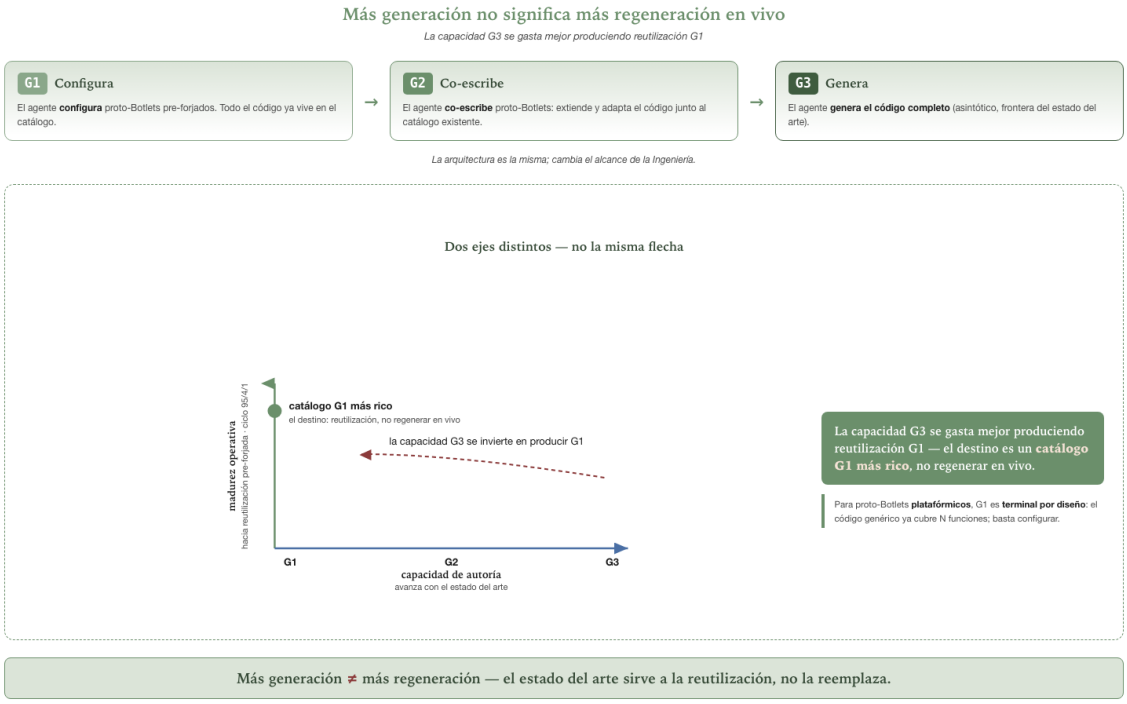


Figura 23: Generaciones del Botlet — capacidad de autoría vs madurez operativa

¿Qué eje?	¿Qué mide?	¿Dirección de "avance"?
Capacidad de autoría	Cuánto puede forjar el agente: configurar (`G1`) → co-escribir (`G2`) → generar entero (`G3`)	Hacia `G3`, conforme avanza el estado del arte de la cognición
Madurez operativa	Para una operación recurrente, cuánto se reutiliza pre-forjado vs se regenera cada vez (ciclo `95/4/1`)	Hacia la reutilización (`G1`), conforme el Botlet madura junior → senior

No son la misma flecha. Un agente con capacidad `G3` que regenera cada artefacto desde cero en cada ejecución no es avanzado: tiene el músculo y elige re-aprender el movimiento cada vez. La reconciliación es directa: la capacidad `G3` se gasta mejor **produciendo reutilización `G1`**. Las generaciones describen lo que el agente *puede* autorar; el ciclo `95/4/1` describe lo que un agente maduro *reutiliza*. El destino de la capacidad `G3` es un catálogo `G1` más rico, no la regeneración en vivo de todo.

Hay un corolario para los proto-Botlets **platafórmicos**. Para uno de ellos, `G1` es **terminal por diseño**, no estación de paso: su identidad es código genérico más configuración. Un platafórmico "en `G3`" — donde el agente regenera el motor por cada pieza — no es una versión más avanzada; disuelve el proto-Botlet y colapsa de vuelta al modo agéntico que la arquitectura existe para trascender.

`G1` no es configuración pobre. Lo que define `G1` es que el agente no escribe el cuerpo del proto-Botlet — pero la configuración que rellena puede ser tan rica como un `DSL` composicional con expresiones

formales evaluables. La distinción `G1`/`G3` es sobre **autoría del cuerpo del proto-Botlet**, no sobre expresividad de la configuración. Un proto-Botlet platafórmico con un `DSL` rico es `G1` puro.

Eso deja un caso frontera: configuración que admite expresiones formales evaluables — `SQL`, especificaciones de gráfico, expresiones de filtro. El **filo `G1`/`G2`** lo resuelve:

- Una expresión formal evaluable que es **parámetro de una Capability bien definida** (`SQL` → `execute-sql`, una especificación de gráfico → `render-chart`, una expresión de filtro → `filter-stream`) es configuración → **`G1`**.
- Una expresión que **extiende o sobreescribe la lógica interna del proto-Botlet** — callbacks, lambdas que el proto-Botlet evalúa internamente, fragmentos que se concatenan a su cuerpo — es código escrito por el agente → **`G2`**.

El test es uno solo: *"¿el código pertenece a la Capability invocada o al proto-Botlet mismo?"*. Si lo evalúa una Capability del catálogo, `G1`; si lo evalúa el proto-Botlet en su lógica interna, `G2`.

La implementación de referencia, Vergis, opera hoy en `G1`: su catálogo expone proto-Botlets — Mira entre ellos — que el agente especializa configurando, no regenerando (Capítulo 9). El sentido profundo de las generaciones — por qué el agente avanzado genera *menos*, no más — es el ensayo con que cierra el libro.

Garantía de fallback — la propiedad innegociable

La garantía de fallback merece tratamiento detallado porque es lo que hace al Botlet **operacionalmente confiable** en lugar de **frágilmente automatizado**. Una organización que depende de un Botlet para una operación crítica — procesar un batch nocturno, enviar reportes regulatorios, conciliar transacciones — debe poder confiar en que el Botlet va a ejecutarse o, en su defecto, alguien va a ejecutar la tarea por él. La garantía de fallback es lo que sostiene esa confianza.

Cuando un Botlet falla catastróficamente — no porque el ambiente cambió levemente y la cognición pueda regenerar el código, sino porque algo realmente impide la ejecución —, la cognición ejecuta la tarea manualmente. *Manualmente* en este contexto significa que el LLM hace el trabajo paso a paso, invocando los tools subyacentes que el Botlet usaría, pero sin la eficiencia del código compilado. El proceso es más lento y más costoso — la cognición consume tokens — pero **el trabajo se hace**. La organización no se queda detenida.

Esta garantía no es decorativa. Es lo que distingue al Botlet conforme a esta spec de cualquier "macro inteligente" o "automatización con IA" frágil. Las macros tradicionales fallan y dejan la operación detenida; los Botlets fallan y la cognición toma el relevo. La diferencia es estructural y se traduce directamente en disponibilidad operativa: una organización con Botlets correctamente diseñados puede prometer SLAs de operación que serían imposibles con automatización tradicional.

> *El Botlet no reemplaza a la cognición. La libera del trabajo repetitivo, pero queda como red de seguridad.*

La cita anterior resume bien la relación entre las dos capas. La cognición no es residual — sigue siendo la inteligencia general que sostiene el sistema. El Botlet es eficiencia operativa que opera mientras el ambiente lo permite. Cuando el ambiente sale del rango, la cognición vuelve.

¿Cuándo usar Botlets, y cuándo no?

No todas las tareas se benefician de ser delegadas a Botlets. La spec define criterios claros para decidir cuándo conviene generar un Botlet y cuándo conviene mantener la tarea bajo cognición continua.

Conviene generar un Botlet cuando la tarea es repetitiva — más de diez invocaciones es regla práctica útil —, cuando el patrón es estable en su núcleo aunque el ambiente puede cambiar, cuando el proceso es crítico y debe ser rápido, cuando el costo de cognición por invocación es relevante a escala, y cuando hay tolerancia a regeneración esporádica del código sin que eso afecte la operación.

No conviene generar un Botlet cuando la tarea es única o de baja frecuencia, cuando el patrón es altamente variable y cada invocación requiere juicio fresco, cuando la tarea exige razonamiento profundo que un script no puede capturar, cuando es prototipo o exploración donde la flexibilidad importa más que la eficiencia, cuando el costo total es irrelevante y la cognición continua es práctica, o cuando los cambios en el ambiente son tan constantes que el Botlet se regeneraría todo el tiempo, perdiendo su beneficio. Hay un caso intermedio que merece nombre propio: la tarea **recurrente en su forma pero interpretativa en cada instancia** — el patrón es estable, pero cada ejecución exige juicio fresco que ningún código determinístico captura. Esa tarea no pertenece al Botlet ni a la cognición continua: pertenece al **Agentlet**, la primitiva hermana que el §7 de este capítulo formaliza.

La regla práctica que sintetiza estos criterios: **si la tarea se ejecuta más de diez veces y su lógica es estable en su núcleo, conviene generar un Botlet**. Por debajo de ese umbral, la cognición es más eficiente. Por encima, la diferencia económica empieza a ser material.

Una observación importante: la decisión de cuándo generar Botlet no la toma un humano. La toma la cognición misma, asistida por Pattern Recognition que detecta los patrones repetitivos. El humano define las reglas generales — qué tipos de tareas son candidatas, qué umbrales de frecuencia son relevantes, qué tipos de ambientes son sensibles —, pero la decisión específica en cada caso emerge del comportamiento del agente. Esta es propiedad del sistema agentivo: la decisión de optimización es propia del agente, no externa a él.

Las tres pruebas del Botlet — minion, especialidad y vida

Los criterios anteriores dicen cuándo *conviene* generar un Botlet. Falta la pregunta previa, que es de identidad y no de conveniencia: ante un artefacto que ya existe —un programa, una configuración, un informe, un instrumento—, ¿es un Botlet o es otra cosa? La pregunta no es académica. Dos lectores con este mismo capítulo delante pueden discrepar en minutos sobre si cada Producto de Información de Mira es un Botlet o si lo es Mira, y la discrepancia no se resuelve releyendo la definición: se resuelve con pruebas. Son tres, y un artefacto es Botlet solo si pasa las tres.

La prueba del minion. Un Botlet es un minion con oficio. La pregunta es qué hace en su vida. Si **ejecuta** una tarea una y otra vez —cada ejecución es un caso de su vida— es un minion. Si solo **se aplica**, se lee o se consume, no es el minion: es lo que un minion maneja. La forma corta de la pregunta es *¿esto se ejecuta o se aplica?* Un Producto de Información se ejecuta: cada pulso de un Botlet discreto y cada suscripción a uno continuo es una ejecución, y su vida es la sucesión de esas ejecuciones adaptándose a una fuente que cambia. Un instrumento de evaluación se aplica: alguien lo rinde, y el instrumento no hizo nada. La sutileza que esta prueba protege es que los casos de la vida de un minion son sus ejecuciones en el tiempo, no encargos distintos: un obrero en una sola estación de la línea tiene un oficio completo aunque haga siempre lo mismo. Y el corolario: cuando se cuentan tantos Botlets como casos, hay que mirar si esos casos se ejecutan o se aplican; si se aplican, se están contando instrumentos.

La prueba de especialidad — `RISC` contra `CISC`. Decide la granularidad cuando hay dos formas de contar. Una arquitectura `CISC` ofrece pocas instrucciones complejas que hacen mucho cada una; una `RISC`, muchas instrucciones simples con una sola función, que componen. El canon elige `RISC` — la misma preferencia que hizo de Unix una colección de herramientas pequeñas que se encadenan —: **un Botlet, una especialidad.** Cuando la duda es entre «un Botlet que maneja N cosas» y «N Botlets que hacen una cosa cada uno», la respuesta canónica es la segunda, y lo que impide que N Botlets sean N programas es el motor compartido: el proto-Botlet platafórmico. Un Mira concebido como un solo Botlet que administra N Productos de Información sería `CISC`: su identidad, su madurez, su fallback y su contrato de calidad serían promedios de un monolito, y cuando una fuente cambiara se degradaría el conjunto. Un Botlet por Producto de Información es `RISC`: cada uno cae, madura y se retira solo. La prueba no dice «más pequeño siempre»; dice **una especialidad por unidad**.

La prueba de vida. ¿Tiene el artefacto un ciclo de vida que gestionar? Un Botlet está en marcha o detenido, activo o retirado; puede fallar por sí mismo; madura de junior a senior; tiene temporalidad y contrato de calidad; responde a `status`, `activate`, `deactivate` y `retire`. Si hay algo que puede estar vivo, caído, degradado o retirado, hay un Botlet. Si lo único que tiene el artefacto es un estado editorial —vigente o retirado como contenido, una versión—, no es un Botlet: es catálogo. Un Producto de Información tiene vida: está sirviendo o está caído, su fuente cambió, maduró. Un instrumento de evaluación tiene una versión, no una vida.

Aplicadas a la implementación de referencia, las tres pruebas dan el mismo resultado que la regla de granularidad de más abajo: **Mira es el proto-Botlet platafórmico —el motor—; cada Producto de Información en vivo pasa las tres pruebas y es un Botlet; el spec del Producto de Información es lo que lo especializa.** Y aplicadas a un caso hermano fuera de la familia de información —una plataforma de estudio que aplica instrumentos de evaluación a un estudiante— dan el resultado contrario: el evaluador que aplica, corrige y registra pasa las tres y es un Botlet; el instrumento que aplica no pasa ninguna, y es contenido de catálogo, inmutable una vez publicado. Las dos respuestas salen de las mismas pruebas, que es lo que se les pide. Ese evaluador tiene nombre propio en la implementación de referencia —**Daftar**, el segundo proto-Botlet del catálogo de Vergis (Capítulo 9)— y pertenece a la familia de **captura**, que la sección siguiente define.

Manifestación y temporalidad del Botlet

Un Botlet es memoria muscular: una disposición latente, un saber-hacer almacenado que no es nada hasta que se ejerce. Cuando el Botlet se ejecuta, ese latente se actualiza en el mundo. Esa actualización es su **manifestación**: el paso de potencia a acto del Botlet, perceptible o no.

La palabra exige cuidado. **Manifestación no es aparición.** El término corriente sugiere "hacerse visible", y eso dejaría fuera casos legítimos: un Botlet que dispara una ingestión periódica se manifiesta — actualiza su latente, produce un efecto — aunque no deje ningún artefacto visible. Por eso el canon la define como *actualización del latente*, no como *aparición*: el efecto invisible cuenta tanto como el artefacto a la vista.

La manifestación es el **género abstracto**; cada familia de Botlet la especializa, y cada práctica le pone su nombre cargado. Las familias se distinguen por la **dirección del efecto** —de dónde a dónde va lo que el Let mueve—, y son cuatro:

- familia de **información** → del Let al humano: **representa** algo que el sistema ya tiene; su manifestación deja un **Producto de Información** (`PI`),
- familia de **captura** → del mundo al Let: **registra** algo que antes no estaba en el sistema —una

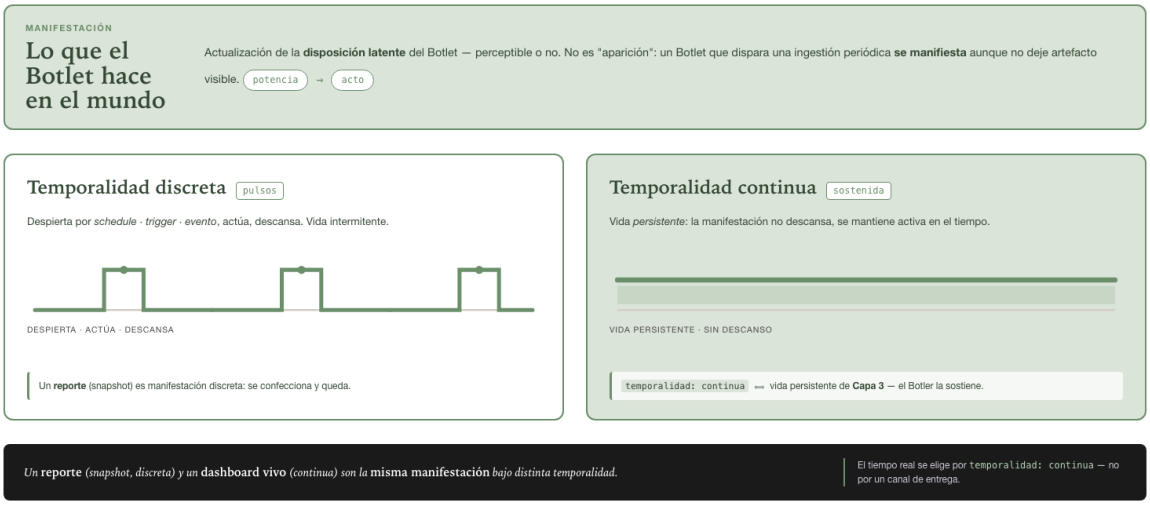


Figura 24: Manifestación y temporalidad del Botlet — discreta vs continua

medición, una respuesta, un archivo ingerido—; su manifestación deja un **registro**, y la práctica de cada dominio le pone nombre: un intento, una lectura, un asiento,
- familia de **actuación** → del Let a un sistema: **cambia** algo, y el Let mismo cierra el lazo; su manifestación es un efecto sobre el mundo, sin artefacto,
- familia de **decisión** → **resuelve** entre alternativas; la nombra su propia práctica.

La prueba de pertenencia es una sola pregunta: *¿qué había en el sistema antes y después del Let?* Si el sistema queda igual y solo el humano sabe más, es información; si entró un dato que no estaba, es captura; si algo fuera del sistema cambió por mano del Let, es actuación; si quedó cerrada una alternativa, es decisión. La ingestión periódica que sirvió de ejemplo unas líneas más arriba es captura: registra lo que el mundo tenía y el sistema no. Y un Let interactivo —el que pregunta y registra lo que le responden— **compone** información y captura sin ser una familia nueva: una «familia de comunicación» fue evaluada y descartada justamente porque cada uno de sus actos se descompone en representar y registrar. Las familias son primitivas de dirección; los Lets se componen de ellas.

Daftar es el proto-Botlet de referencia de la familia de captura (Capítulo 9): su especialidad es **evaluar** — aplicar un instrumento y registrar lo que el estudiante sabe, en sus dos modos, práctica y examen (la medición). La corrección inmediata del modo práctica es retroalimentación de la captura —como un formulario que valida lo que se le escribe—, no actuación: el aprender lo ejecuta el estudiante afuera, y el Let no cambia nada en el mundo por su mano. Dos clasificaciones se probaron y se descartaron: «información», porque miraba el subproducto —el reporte de resultados— y no el propósito; y «actuación», porque la transferencia de conocimiento no la ejecuta el Let. La pregunta de pertenencia lo zanja: antes del Let el sistema no sabía qué sabe el estudiante; después, sí.

El `PI` **no es primitiva del canon**: es la manifestación de *una* familia. El canon se queda en `manifestación`; el Producto de Información es un **término normado de esta spec** — no primitiva, pero sí vocabulario con reglas —, y su carga de gobernabilidad se añade aquí sin contaminar el nivel canónico. Esta es su descripción de referencia:

Producto de Información multi-vista · drill-through. Un `PI` no es necesariamente una pieza única. Puede componerse de **N piezas nombradas**: cada **vista** es una pieza más del mismo `PI`, elegible desde un selector, con una vista por defecto (la primera). El `PI` es **authz-blind** — ni las vistas ni las aristas que las conectan declaran autorización; esa política vive en el policy store, no en la composición.

La conexión entre vistas es el **drill-through**: una **arista de navegación con contexto**. Una tabla declara *"al clickear una fila, ir a la vista destino pasando la clave de esa fila"*; la vista destino se renderiza **filtrada por esa clave**. La propiedad crítica es **data-anchored / no-bypass**: el contexto que viaja con la arista **acota dentro de lo que el viewer ya puede ver** — la vista destino aplica su propia política de filas (`RLS`, *row-level security*: la seguridad a nivel de fila) sobre la fuente, y el contexto entra como filtro adicional, nunca como override de la política (**MUST**). El drill **acota, nunca amplía** — intersección con lo autorizado, jamás unión. Si el viewer no alcanza la fila origen, no llega a la arista; si llega, el destino sigue gobernado por su propia política.

Un reporte de cartera / aging de saldos ilustra el patrón: vistas nombradas (Clientes, Proveedores, Relacionados, Detalle) sobre el mismo `PI`, una tabla jerárquica Empresa→Socio, y una arista de drill-through Socio→Detalle que abre los documentos de ese socio — filtrados por la clave del socio y acotados a lo que el viewer ya tenía derecho a ver. La composición multi-vista es ortogonal a la familia del Botlet de operación: lo que cambia es cuántas piezas componen la manifestación, no su naturaleza.

Figura 25: PI multi-vista y drill-through — navegación con contexto, data-anchored

La **temporalidad** es el régimen de la manifestación. Es atributo declarado del Botlet, con dos valores:

Temporalidad	¿Cómo se manifiesta?	Relación con el runtime
`discreta`	En pulsos: despierta por schedule, trigger o evento, actúa, descansa	El Botler invoca o agenda; el Botlet no vive entre pulsos
`continua`	Sostenida: vive persistente y se manifiesta sin cesar	El Botler sostiene la ejecución mientras el Botlet viva

`temporalidad: continua` equivale a la vida persistente de Capa 3: obliga al Botler a sostener la ejecución del Botlet sin re-arranque por cada manifestación. Un Botlet conforme con temporalidad continua MUST poder ser sostenido por el runtime persistente; un Botlet con temporalidad discreta se manifiesta vía schedule, trigger o evento.

La consecuencia operativa es fuerte: **el tiempo real no se elige en un canal de entrega.** No se obtiene marcando un canal como `push`; se obtiene dándole al Botlet temporalidad continua, lo que a su vez obliga al runtime persistente. El modo de entrega es el síntoma; la temporalidad continua es la causa. Esto reubica el "tiempo real" del nivel del canal al nivel del Botlet, y conecta con la distinción *Empresa en línea ≠ Empresa en tiempo real*: no son dos clases de información, sino dos puntos del continuo de temporalidad.

De aquí se sigue una economía de runtime. Un reporte — snapshot en un punto del tiempo — y un dashboard vivo no son dos tipos distintos de *qué*: son la **misma manifestación bajo distinta temporalidad**. Por eso un único runtime los cubre: se construye el caso más difícil (continua) y los casos simples son configuraciones degeneradas de ese caso, no codepaths aparte. La distinción se sostiene precisamente porque la temporalidad es ortogonal a lo que se manifiesta.

Botler — el framework runner

Botler es la infraestructura que ejecuta los **Lets** — el nombre propio del género de las unidades empaquetadas de la Capa 3 (Autonomía): los Botlets que esta sección desarrolla y los Agentlets que el §7 formaliza como su especie hermana. Es invisible para el usuario y para el agente; es responsabilidad de la implementación del AgencyDomain. La relación canónica es simple y se enuncia sobre el género: un proceso del AgencyDomain contiene un Botler, y el Botler gestiona N Lets que viven dentro de ese proceso — **1 Proceso = 1 Botler + N Lets**. En esta sección, donde los Lets son Botlets, la relación se instancia como 1 Botler + N Botlets.

El Botler provee cuatro funciones críticas. La primera es **aislamiento de ejecución** — sandboxing apropiado al ambiente, que detallamos en la próxima sección. La segunda es **gestión del ciclo de vida del Botlet**: invocación cuando se necesita, monitoreo durante ejecución, detección de fallos, disparar regeneración cuando corresponde. La tercera es **comunicación con la cognición** cuando el Botlet detecta un fallo o cambio en el ambiente que excede su capacidad de manejo. La cuarta es **trazabilidad**: cada invocación del Botlet, cada resultado, cada fallo, cada regeneración, queda registrado en el append-only log del Trust Layer. Esta trazabilidad es lo que permite reconstruir, auditablemente, qué hizo el agente y por qué — y es indispensable para gobernanza.

El Botler como abstracción importa porque desacopla la implementación de aislamiento del agente que lo usa. El agente no sabe — ni tiene por qué saber — si su Botlet corre en un contenedor Docker, en una sandbox WASM, o en una microVM. Solicita ejecución al Botler; el Botler ejecuta bajo el modelo de

aislamiento que la implementación del AgencyDomain eligió. Esta separación es lo que permite que la spec sea agnóstica a tecnología de aislamiento — distintas implementaciones eligen distintas tecnologías según sus tradeoffs específicos.

El Botler es genérico por definición

El Botler **no entiende el dominio** de los Botlets que ejecuta. Gestiona el ciclo de vida, el aislamiento y la ejecución de *cualquier* Botlet sin saber qué hace ese Botlet ni a qué disciplina pertenece. Toda la especialización de dominio vive en los Botlets y en sus proto-Botlets, nunca en el runtime que los hospeda. La arquitectura es plana: un runtime genérico hospeda componentes especialistas autocontenidos.

De aquí se sigue una propiedad estructural: **no existen subtipos de Botler por familia de operación.** No hay un Botler "informativo", uno "transaccional", uno "para artefactos de información" — un Botlet de operación informativa ya carga su propia frescura, su caché, su distribución, de modo que un subtipo de Botler que lo duplicara contradiría la genericidad del runtime sin agregar nada.

> *Los subtipos de Botler se distinguen por topología y rol de despliegue — central, edge, fachada operativa para Botlets invocables desde Capa 1 —,* ***nunca por dominio****. La especialización de dominio vive íntegramente en los Botlets que el Botler ejecuta.*

Los ejes de topología y rol son legítimos porque responden a *dónde* corre el runtime y con qué autonomía, no a *qué* dominio ejecuta: un Botler central y un Botler edge difieren en conectividad y operación offline, no en conocimiento de negocio. La nota normativa es la frontera: cualquier distinción de Botler que apele a la familia de operación que ejecuta está mal planteada.

El Botler valida orquestando, no ejecutando

El registro de un Botlet exige que su spec sea válida contra el tipo declarado antes de aceptarlo. Esto plantea una tensión aparente: si el Botler no entiende el dominio, ¿cómo valida una spec cuyo sentido es de dominio? La respuesta distingue **orquestar** la validación de **ejecutarla**.

El Botler **hace valer** la validación; no la ejecuta con conocimiento de dominio. En el momento del registro invoca el punto de validación que el propio Botlet provee — o que provee su proto-Botlet en G1 —, le entrega el contexto genérico que sí controla (el catálogo de Capabilities disponibles, la identidad, las políticas del AgencyDomain) y actúa sobre el veredicto: acepta, rechaza, o registra el resultado en el append-only log. El juicio de qué hace válida a la spec vive enteramente en el Botlet; el Botler decide admitir o no según un veredicto que no produjo. Así, ninguna spec inválida entra — el rechazo ocurre antes de admitir el Botlet y queda trazado — sin que el runtime interprete jamás el dominio.

Este patrón tiene un hermano en la invocación de Capabilities. El Botler es el único punto por el cual un Botlet invoca Capabilities, y el bypass por canales paralelos no se prohíbe solo por política: se hace estructuralmente imposible. En cada invocación el Botler entrega al Botlet un **handle controlado** — un objeto con acceso a Capabilities y al log ligado al propio Botler — en vez de permitir que el Botlet construya accesos por su cuenta. El Botlet solo puede actuar sobre el mundo a través de ese handle. Ambos casos comparten el principio: el Botler genérico expone **puntos de control** y el especialista se enchufa en ellos; el runtime se mantiene delgado sin renunciar a las garantías que el contrato exige.

La interfaz Capa 2 ↔ Capa 3 vía `MCP`

El Botler es el único punto de ejecución de los Botlets, y eso incluye la operación que la propia Cognición dirige: la **Cognición** (el agente `LLM`, Capa 2) comanda al Botler (runtime de Capa 3, sin agencia) por una interfaz **interna** cuyo transporte natural es `MCP` — el Botler expone servidores `MCP` y la Cognición es el cliente. Esta interfaz **no es `A2A`**; la corrección formal de la nomenclatura `A2A` y la interfaz Capa 2 ↔ Capa 3 se desarrollan en el Capítulo 5 §1.

Código fuente vs spec · dos superficies · un Botlet por `PI`

La operación de un Botlet involucra dos cosas que conviene no confundir. Una es el **código fuente** del Botlet: su implementación. Para un proto-Botlet platafórmico, ese código fuente es el **motor**, compartido por todos los Botlets que de él se instancian. La otra es la **spec**: lo que especializa el comportamiento del Botlet a su instancia. La spec no es el código; lo configura.

Esa separación se proyecta en **dos superficies** de gestión:

Superficie	¿Qué gestiona?	Granularidad	Cadencia
Ciclo de vida del código fuente	Instalar, versionar, cargar y descargar el motor	Nivel-Botler	Releases (Producto)
Operación	Especializar, manifestar, consumir y controlar cada Botlet	Por-Botlet	Fluida (Instancia)

La operación **incluye configurar el spec**. Para un proto-Botlet platafórmico, el spec es el input operacional: no hay "operar" sin spec. De ahí un principio: **configurar el spec es operar**. El agente evoluciona el spec operando — verbo `specialize` —, de forma fluida; la cristalización del spec a un registro versionado **sigue, no precede**, y la proveniencia la da el append-log. Los verbos del API de operación son `specialize`, `invoke` y `schedule` (para temporalidad discreta), `read` y `subscribe` (para temporalidad continua), y `status`, `activate`, `deactivate`, `retire`.

De esto se desprende la regla de granularidad: **un Botlet por `PI`** sobre un motor compartido. Cada Producto de Información es su propio Botlet — su propio servicio, con identidad, temporalidad, madurez y fallback propios —, especializado del motor compartido (el proto-Botlet platafórmico). No es un Botlet con N configuraciones, que perdería esa independencia; no son N programas, porque el motor es uno. Es la relación `1 Proceso = 1 Botler + N Botlets`, con los Botlets como instancias especializadas del mismo proto-Botlet. El rationale es `RISC`: muchos Botlets simples y focalizados — uno por `PI` — componen mejor que un monolito. Las tres pruebas del Botlet, formuladas más arriba en este capítulo, llegan al mismo lugar por el otro lado: el Producto de Información se ejecuta, tiene una especialidad y tiene vida.

El consumo `subscribe` es el puente al **norte agentivo**. Hoy lo consume un humano — un navegador suscrito por `SSE` que recibe la corriente de manifestaciones de un Botlet continuo —; mañana lo consume la Cognición, alimentándose de esa misma corriente para decidir. La temporalidad continua, el runtime persistente y el verbo `subscribe` son, juntos, la infraestructura de ese norte.

Modelo de aislamiento — sandboxing

Los Botlets son **código generado dinámicamente por un agente**. Esto exige aislamiento estricto. Un Botlet mal escrito, o un Botlet generado por un agente que fue víctima de prompt injection, podría intentar acciones maliciosas — exfiltrar datos, modificar archivos del sistema, abrir conexiones de red no autorizadas. El aislamiento es lo que contiene esos riesgos.

La spec admite cuatro estrategias de sandboxing, con sus trade-offs:

Procesos con seccomp ofrecen aislamiento bajo y overhead mínimo. Es estrategia útil solo en ambientes controlados donde el riesgo de código malicioso es bajo — por ejemplo, Botlets ejecutándose dentro del perímetro privado de una organización con confianza implícita en sus propios agentes. No es estrategia adecuada para Botlets que tocan datos sensibles o que operan bajo régimen público.

Contenedores — Docker, Podman, equivalentes — ofrecen aislamiento medio-alto con overhead medio. Es la estrategia más práctica para Botlets genéricos que necesitan invocar tools del sistema operativo o redes. Los contenedores tienen ecosistema maduro, portabilidad razonable, herramientas operacionales abundantes. Son default razonable para la mayoría de los casos.

WASM (WebAssembly) ofrece aislamiento alto con overhead bajo, pero el ecosistema es más limitado. WASM es ideal para Botlets transformacionales puros — cálculos, transformaciones de datos, lógica algorítmica — que no necesitan acceso al sistema operativo subyacente. La velocidad de inicio es muy rápida (milisegundos), lo que importa cuando un agente necesita invocar muchos Botlets simultáneamente.

MicroVMs — Firecracker, Kata Containers — ofrecen aislamiento máximo con overhead alto. Son adecuadas para Botlets que manejan datos altamente sensibles o que operan en ambientes multi-tenant compartidos donde el riesgo de cross-tenant leakage es inaceptable. El overhead típicamente significa decenas de milisegundos de inicio adicional, que en algunos casos es prohibitivo.

La recomendación canónica para implementaciones de referencia es **híbrida**: WASM para transformers puros (sin acceso al sistema), contenedores para Botlets genéricos que necesitan invocar tools del sistema operativo, MicroVMs para Botlets que manejan datos altamente sensibles o que operan en entornos multi-tenant compartidos. La elección específica para cada Botlet depende de su perfil de riesgo y de sus requisitos de performance.

Lenguaje de los Botlets

La spec es **agnóstica al lenguaje** de implementación de los Botlets. La cognición puede generar Botlets en cualquier lenguaje siempre que el Botler los pueda ejecutar dentro del sandbox elegido. Esta agnosticidad importa porque el panorama de lenguajes evoluciona — un sistema atado a un lenguaje específico puede quedar obsoleto cuando ese lenguaje pierde tracción en el ecosistema de IA.

Las recomendaciones prácticas para implementaciones de referencia son:

Python es primera elección. La madurez del ecosistema, las librerías para casi cualquier integración, la generación por LLM altamente confiable — los modelos contemporáneos generan Python correctamente con frecuencia muy alta —, hacen de Python el default razonable para Botlets genéricos.

JavaScript / TypeScript son adecuados para Botlets que tocan APIs HTTP o que automatizan browsers. El ecosistema npm tiene cobertura amplia, y la generación por LLM también es confiable.

Bash o shell son adecuados para Botlets que orquestan comandos del sistema operativo. La generación de bash por LLM es confiable para casos simples, menos confiable para casos complejos donde la sintaxis de bash tiene quirks.

Rust o Go — lenguajes compilados — son adecuados para Botlets de muy alta frecuencia donde el overhead del intérprete importa. La generación por LLM es menos confiable que Python, pero los Botlets resultantes ejecutan más rápido. Esta combinación tiene sentido cuando un Botlet va a ejecutarse millones de veces y la diferencia de milisegundos por invocación se acumula a impacto material.

Hay una propiedad importante: el Botlet **no es editable por humanos**. Es regenerado por la cognición. Esto importa porque cualquier mejora manual al código del Botlet — un humano que abre el archivo y mejora la lógica — se convierte en deuda en el momento que la cognición lo regenere por cambio de ambiente. La regeneración elimina las mejoras humanas. Por eso la spec define que los Botlets son **read-only para humanos en producción**: si un humano quiere mejorar la lógica, debe mejorar la cognición o las Capabilities, no el Botlet directamente.

Pattern Recognition — la entrada al ciclo

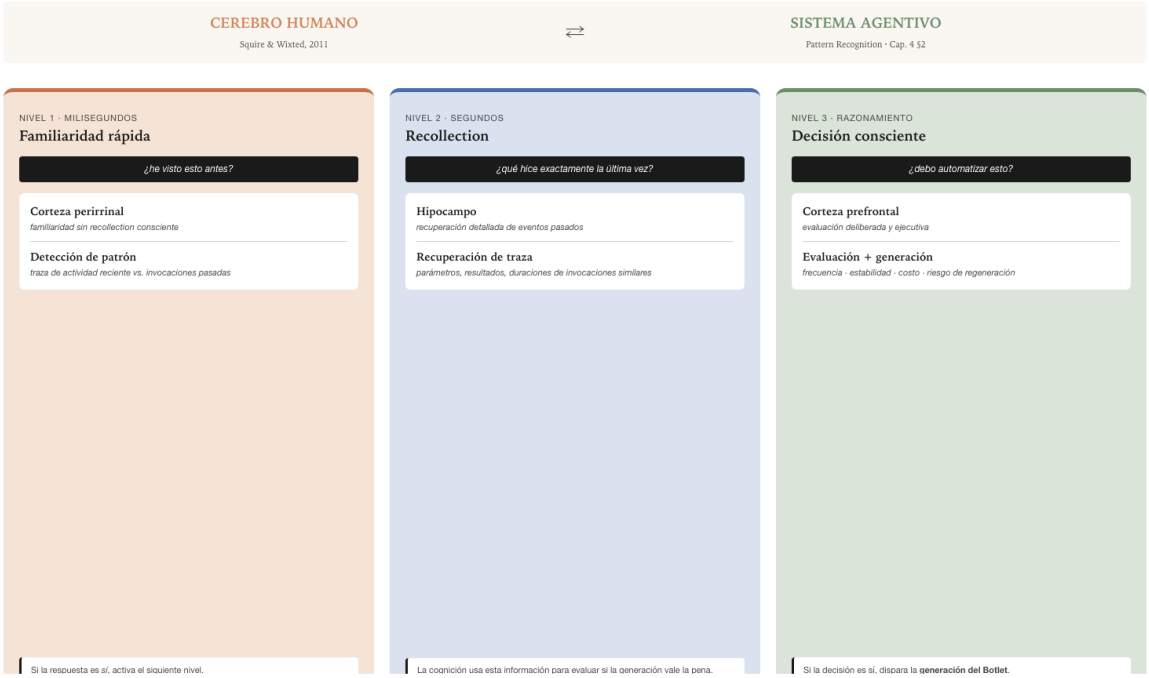


Figura 26: Inspiración neurobiológica · cerebro humano ↔ sistema agentivo

Los Botlets no se generan al azar. La cognición decide generar un Botlet cuando reconoce un **patrón repetitivo** en la actividad del agente. El componente que detecta esos patrones es **Pattern Recognition** — primitiva auxiliar de la Capa 2 que ya mencionamos en el Capítulo 4.

Pattern Recognition opera sobre la traza de actividad del agente: qué tareas hizo, con qué frecuencia, con qué entradas similares, con qué resultados. Cuando un patrón cruza un umbral — la misma tarea

repetida más de N veces con variabilidad acotada en sus entradas, los resultados estables —, la cognición evalúa si vale la pena generar un Botlet.

La inspiración neurobiológica que mencionamos al inicio se refleja en el diseño del Pattern Recognition. La **corteza perirrinal** del cerebro humano implementa familiaridad rápida — el "¿he visto esto antes?" que ocurre en milisegundos, sin recollection consciente. El Pattern Recognition agentivo implementa lo análogo: detección rápida de tareas similares a tareas pasadas, sin necesidad de razonamiento profundo. Si la respuesta es sí, el sistema activa el siguiente nivel.

El **hipocampo** implementa recollection: el "¿qué hice exactamente la última vez?". El Pattern Recognition recupera la traza específica de las invocaciones pasadas, con sus parámetros, sus resultados, sus duraciones. Esta información es lo que la cognición usa para decidir si generar Botlet vale la pena.

La **corteza prefrontal** implementa decisión consciente: el "¿debo automatizar esto?". Aquí la cognición evalúa los criterios — frecuencia, estabilidad del patrón, costo de cognición continua, riesgo de regeneración — y decide. Si decide sí, dispara la generación del Botlet.

Tres etapas, tres niveles de procesamiento. Pattern Recognition no es un solo paso — es un proceso jerárquico que filtra desde "potencialmente interesante" hasta "vale la pena automatizar". La especificación de Pattern Recognition como tool de Capa 2 está fuera del alcance de este libro; basta retener que es la primitiva auxiliar que activa el ciclo del Botlet.

Botlets y la economía de la autonomía bajo suscripción

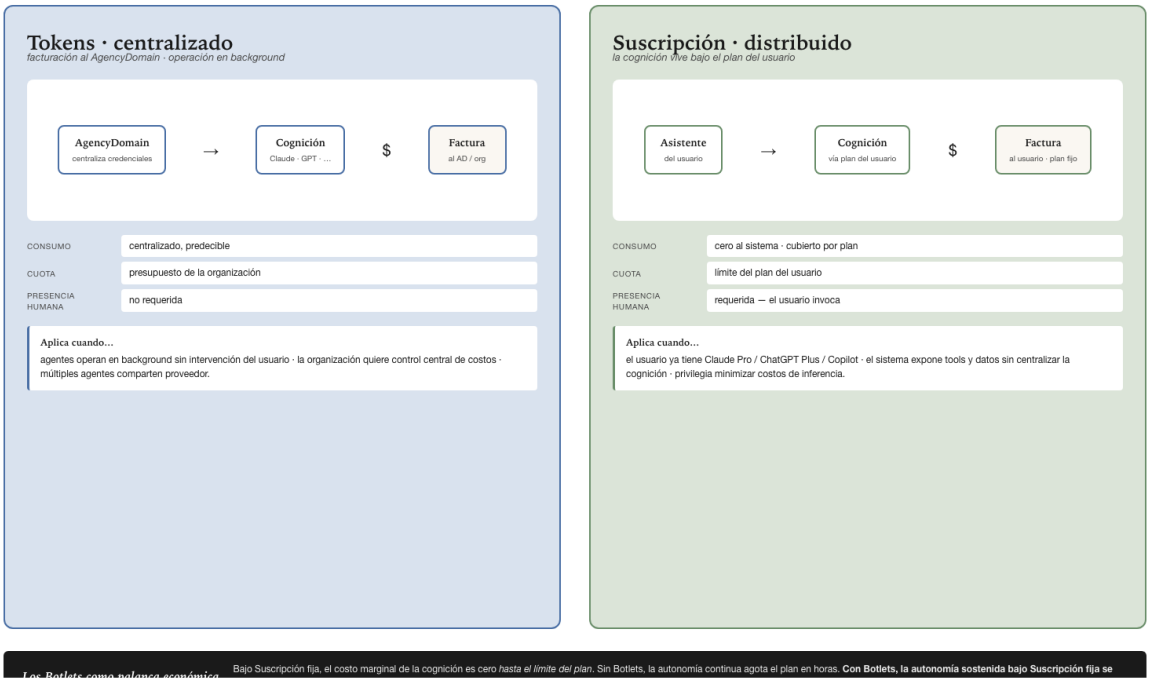


Figura 27: Tokens vs. Suscripción · dos modos de acceso a la cognición

Una propiedad operativa crítica del Botlet, que ya introdujimos en el Capítulo 4 §2 pero que merece desarrollo aquí, es su rol en la economía de la autonomía bajo planes de Suscripción fija — Claude Pro, ChatGPT Plus, Copilot empresa.

En estos planes, el costo marginal de invocar la cognición es **cero hasta el límite del plan**. El usuario paga una mensualidad fija y puede invocar el modelo cuantas veces quiera, hasta que el plan agota su cuota. Esta estructura económica es muy distinta de los modelos de pago por token, donde cada invocación cuesta y la economía se calcula por uso real.

El problema es que el límite existe. Un Agente Autónomo operando continuamente en background, sin Botlets, invocaría la cognición miles de veces al día — cada decisión, cada validación, cada acción consultaría al modelo. Bajo plan de Suscripción fija, el agente agotaría la cuota del usuario en horas. La autonomía continua sería económicamente imposible.

Los **Botlets son el mecanismo arquitectónico para extender autonomía sin saturar el plan**. Un agente que ejecuta su trabajo cotidiano vía Botlets — y solo invoca la cognición cuando el ambiente cambia — puede operar en background continuo sin agotar la cuota del usuario. La cognición queda disponible para razonamiento real, no para tareas repetitivas que el código tradicional ejecuta mejor.

> *Sin Botlets, la autonomía sostenida bajo Suscripción fija es económicamente imposible.*

Esto es lo que hace al Botlet **palanca económica**, no solo optimización técnica. La diferencia entre un agente que cuesta doscientos dólares por mes operar y uno que cuesta veinte dólares por mes operar, con capacidad efectiva idéntica, es casi siempre la proporción de tareas que ejecuta vía Botlets versus vía cognición continua. Una organización que adopta Botlets disciplinadamente puede operar agentes a un orden de magnitud menos costo que una organización que invoca cognición para cada operación.

La consecuencia comercial es directa: los proveedores que entregan agentes con buenos Botlets pueden ofrecer pricing competitivo y márgenes razonables; los proveedores que dependen de cognición continua para cada operación enfrentan un dilema económico — o suben pricing a niveles que el mercado no acepta, o operan con márgenes negativos. Esta presión es probablemente lo que llevará a la mayoría de los proveedores serios a adoptar arquitecturas con Botlets en los próximos dos a tres años.

Cadena de derivación y catálogo de proto-Botlets

Los Botlets de un sistema no aparecen sueltos: se derivan de lo que el sistema necesita hacer, y se apoyan en piezas pre-forjadas del catálogo. La spec fija esa **cadena de derivación** como relación estructural:

1. **Casos de uso documentados** — cada caso requiere…
2. **Botlets necesarios** (cero, uno o varios; algunos casos los resuelve la cognición sin Botlet) — cada Botlet es instancia de…
3. **proto-Botlets requeridos del catálogo**.

La cadena es estructural, no metodológica. Un caso de uso puede no requerir Botlet alguno — la cognición lo resuelve directamente —, requerir uno, o requerir varios; cada Botlet que sí existe es instancia de algún proto-Botlet del catálogo. De aquí una propiedad exigida: todo Botlet conforme **MUST** poder trazarse en esta cadena, y el append-only log **MUST** registrar el proto-Botlet de origen de cada Botlet instanciado. El *método* por el cual se descubren los casos de uso y se derivan los Botlets no entra al canon — es uno de varios posibles y vive en los cuerpos complementarios de cada implementador; lo que el canon fija es la relación y su trazabilidad.

El extremo inferior de la cadena — los proto-Botlets — se acumula en **catálogos comunes** que

producen efectos de red. Cada implementador que consume un proto-Botlet contribuye a su maduración: variantes nuevas, configuraciones probadas, refinamientos. Mientras más implementadores lo consumen, más casos cubre y más confiable se vuelve; el implementador n+1 recibe versiones refinadas por los implementadores 1 a n. Un proto-Botlet pertenece a una comunidad de catálogo bajo alguno de estos modos:

Modo de pertenencia	¿Qué es?
Contrato privado	Catálogo cerrado entre cliente y proveedor
Códice propietario	Catálogo privado que un implementador cura (ucodex es un ejemplar)
Catálogo público abierto	AgencyDomains.org: cualquier implementador consume y contribuye
Acuerdo soberano	AgencyDomains que adoptan estándares comunes sin contrato comercial directo

Los modos coexisten sin tensión: un mismo implementador puede consumir el catálogo público abierto y, sobre él, curar un códice propietario con sus refinamientos por casos reales. La cadena de derivación y el catálogo común son, juntos, lo que hace de los proto-Botlets una economía y no una colección de piezas aisladas.

Conformidad

Una implementación de Botlet conforme a esta especificación debe satisfacer los siguientes requisitos:

Requisito	Nivel
Código generado por la cognición, no escrito por humanos	**MUST**
Ejecución sin invocar cognición en operación normal	**MUST**
Garantía de fallback a cognición manual ante fallo	**MUST**
Trazabilidad: cada invocación queda en append-only log	**MUST**
Aislamiento adecuado al ambiente	**MUST**
Regeneración automática ante cambio de ambiente	**SHOULD**
Pattern Recognition como activador de generación	**SHOULD**
Lenguaje agnóstico (no atar a un solo lenguaje)	**MUST**
Reconocimiento de fases de madurez (junior, en aprendizaje, senior)	**SHOULD**
Distinción entre origen seed y origen emergente	**SHOULD**
Trazabilidad de trayectoria de madurez en append-only log	**MUST**
Botlet trazable en la cadena casos-uso → Botlet → proto-Botlet	**MUST**

Requisito	Nivel
proto-Botlet de origen de cada Botlet registrado en append-only log	**MUST**
Botler genérico: sin subtipos de Botler por dominio	**MUST**
Botler hace valer la validación de spec sin ejecutarla con dominio	**MUST**
Soporte de las dos temporalidades (`discreta` y `continua`)	**MUST**
Runtime persistente que sostiene Botlets de temporalidad continua	**MUST**

Capabilities

Cuando un consultor experimentado de finanzas se enfrenta a un problema nuevo en un cliente, no parte de cero. Trae consigo un cuerpo de saber-hacer organizado que aplica al caso particular: principios contables, marcos regulatorios, frameworks de análisis, prácticas profesionales que la industria ha consolidado durante décadas. Cuando un consultor de operaciones aborda un problema de cadena de suministro, hace lo mismo con su propio cuerpo de saber: SCOR, Six Sigma, lean manufacturing, planificación de demanda. Cada disciplina profesional tiene su árbol de saber-hacer, y la calidad del consultor depende en buena parte de cuán profundo y bien organizado está ese árbol en su cabeza.

Los agentes de IA enfrentan el mismo problema. Un agente que opera en finanzas necesita saber finanzas; un agente que opera en legal necesita saber legal; un agente que opera en marketing necesita saber marketing. Pero "saber" en este contexto no significa simplemente que el modelo subyacente haya leído documentación de finanzas durante el entrenamiento. Significa que el agente tiene acceso, modular y composable, a un cuerpo organizado de saber-hacer profesional que puede aplicar selectivamente según la tarea. Esta organización es lo que la Arquitectura Agentiva llama **Capabilities**.

Esta sección desarrolla el concepto de Capability como primitiva canónica de la Capa 2 (Cognición). Lo que hace especial a la Capability respecto a otros conceptos cercanos del campo es su **estructura jerárquica**, su **composabilidad**, y su distinción explícita de plugins y prompts. Un agente sin Capabilities tiene cognición monolítica que confunde dominios; un agente con Capabilities bien organizadas opera con la disciplina de un profesional que conoce las distinciones de su campo.

Definición

Una **Capability** es una unidad de saber-hacer especializado que un agente comprende y aplica. Encapsula el conocimiento procedimental y declarativo de un dominio: qué se sabe, cómo se aplica, qué se decide, qué se pregunta cuando faltan datos. Las Capabilities residen en la **Capa 2 (Cognición)** de la Arquitectura Agentiva. La cognición selecciona y aplica Capabilities según la tarea — del mismo modo en que un consultor experto selecciona y aplica frameworks profesionales según el problema que tiene enfrente.

> *El agente no sabe lo que el modelo sabe. Sabe lo que sus Capabilities le permiten saber.*

La cita anterior distingue lo que la Capa 2 sabe de lo que el modelo subyacente conoce. El modelo subyacente — Claude, GPT, Gemini — fue entrenado con una cantidad masiva de información que

incluye, entre muchas otras cosas, conocimiento profesional. Pero el agente que vive en la Capa 2 **no opera directamente con el conocimiento del modelo**. Opera con las Capabilities que le fueron asignadas, que son curaciones específicas de saber-hacer aplicado a contextos particulares. Un agente que tiene la Capability de *General Ledger* pero no la Capability de *Tax* responde con autoridad sobre contabilidad general pero sabe que no es la fuente correcta para preguntas de impuestos. Un agente sin Capabilities específicas opera con el conocimiento difuso del modelo, que puede ser correcto en general pero rara vez es preciso en lo profesional.

¿Qué NO es una Capability?

Para fijar la definición con la precisión que la spec exige, contrastamos la Capability con conceptos vecinos que la industria usa con descuido. Cada uno de estos conceptos vecinos tiene su rol legítimo en sistemas de IA, pero ninguno es Capability, y confundirlos lleva a arquitecturas que terminan siendo collages mal integrados.

El término **Capability** queda reservado, en sentido estricto, al **saber-hacer cognitivo** de la **Capa 2 (Cognición)**: saber que interpreta, decide y razona sobre un dominio. Esta reserva importa porque otros entregables agentivos —conexiones a sistemas fuente, confecciones de presentación— viven en otras capas, tienen naturaleza distinta y se construyen con esquemas de desarrollo propios. Tratarlos a todos como Capabilities obligaría a apellidar el término en cada uso y diluiría su valor. La sección *Capability, Conector y Plantilla — tres entregables por capa* fija los términos propios de cada capa.

Un **plugin** vive en el contexto de una aplicación host. Extiende esa aplicación con una función nueva. Está atado al host: el plugin de Excel solo opera dentro de Excel, el plugin de Notion solo opera dentro de Notion. Una Capability no está atada a un host — vive en el árbol de Capabilities del agente y es invocable desde cualquier contexto donde el agente opere.

Un **prompt** es una formulación lingüística específica que se inyecta al modelo en una invocación particular. Los prompts son útiles, pero son artefacto transitorio: cambian de invocación a invocación, no codifican saber persistente. La Capability puede usar prompts internamente como uno de sus mecanismos de implementación, pero no se reduce a un prompt. Una Capability bien diseñada incluye vocabulario, conocimiento procedimental, conocimiento declarativo, heurísticas y referencias a tools — el prompt es solo uno de los componentes.

Un **system prompt** establece tono y comportamiento general del modelo. Es configuración del modo de operación, no saber-hacer modular. Las Capabilities **no son system prompts** — pueden coexistir con uno, pero capturan otra cosa: conocimiento del dominio aplicable según la tarea, no configuración global del agente.

Un **tool** vive en la Capa 4 (Acceso). Es acción ejecutable sobre el mundo externo: invocar una API, leer un archivo, enviar un email. La Capability **decide qué tool invocar y cómo interpretar su resultado**, pero no es el tool mismo. La Capability es saber; el tool es acción. Un agente financiero con la Capability de *Treasury* sabe cuándo y cómo invocar el tool que consulta los saldos bancarios — pero el tool en sí es componente separado que vive en otra capa.

Un **skill**, en la terminología popular del campo, es término que la industria usa con tanto descuido que prácticamente ha perdido sentido. Diversos proveedores llaman “skills” a plugins, a prompts, a tools, a configuraciones — sin distinción precisa. La Capability no es skill en el sentido genérico, pero la coincidencia parcial de uso popular puede confundir al lector. Para evitar la confusión, este libro reserva el término Capability con definición precisa y evita usar “skill” salvo en cita textual de fuentes que lo usan.

Tres rasgos distinguen a una Capability genuina. **Modular**: puede activarse o desactivarse independientemente de las demás. Si un agente tiene Capabilities de *Finance* y *Legal* y se le retira *Legal*, el agente sigue operando coherentemente con *Finance* solamente. **Composable**: múltiples Capabilities pueden coexistir en un mismo agente y combinarse cuando la tarea cruza dominios. Un agente que tiene *Finance* y *Operations* puede manejar un caso de logística que tiene componentes financieros sin tropezarse en la transición. **Saber, no acción**: la Capability sabe; los tools (Capa 4) actúan. La Capability decide qué tool invocar y cómo interpretar su resultado.

Capability, Conector y Plantilla — tres entregables por capa

Un proyecto agentivo de entrega rara vez entrega solo una Capability. La Capability cognitiva es la protagonista, pero suele venir acompañada de entregables que viven en otras capas y se construyen con esquemas distintos. La spec canoniza tres términos, uno por capa, para que cada entregable se nombre por lo que es sin apellidar el término Capability.

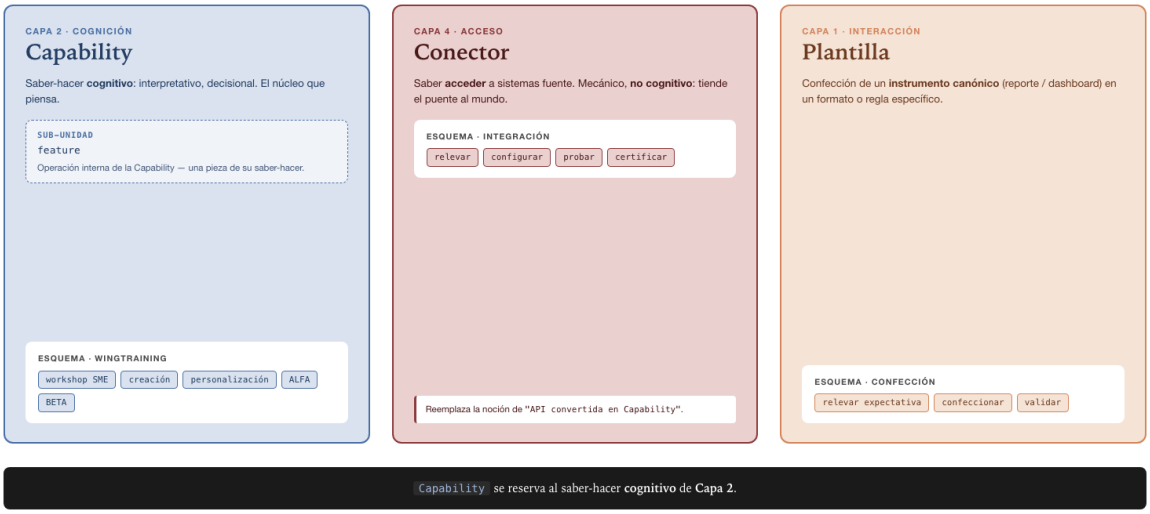


Figura 28: Tres entregables por capa — Capability (Capa 2) · Conector (Capa 4) · Plantilla (Capa 1)

Una **Capability** (sentido estricto) es saber-hacer **cognitivo**, interpretativo, decisional. Vive en la **Capa 2 · Cognición**. Es la protagonista del proyecto.

Un **Conector** es saber **acceder a sistemas fuente** — conexión con poder de ejecución. **No es saber cognitivo**. Vive en la **Capa 4 · Acceso**. El Conector es lo que el campo pre-agentivo conoce como la integración técnica que toca el mundo externo. Una API legacy, al traerse al Mundo Agentivo, se convierte en **Conector** (Capa 4), no en Capability.

Una **Plantilla** es la **confección específica del cliente** sobre un instrumento canónico — un reporte o dashboard que la Capability produce — en un formato o regla particular exigido por el cliente. Vive

en la **Capa 1 · Interacción**, junto a la Faceta, el Botlet de superficie y el Botlet de vista. Ejemplos anonimizados: una plantilla regulatoria de oferta básica de interconexión sobre el reporte canónico de costos; un formato de cierre financiero mensual gerencial sobre el dashboard canónico de rentabilidad. La Plantilla no es Capability, no es Faceta y no es Botlet: el comprador ya conoce el término "plantilla" del vocabulario pre-agéntico y no necesita aprender uno nuevo.

¿Cómo se desarrolla cada entregable?

Cada término tiene su capa, su naturaleza y su esquema de desarrollo propio:

Entregable	Capa	Naturaleza	Esquema de desarrollo
Capability (protagonista)	Capa 2 · Cognición	Saber-hacer cognitivo, interpretativo, decisional	Wingtraining 5 pasos (workshop SME · creación · personalización · ALFA · BETA)
Conector (compañero, si requiere acceso a sistemas fuente)	Capa 4 · Acceso	Saber acceder a sistemas; no es saber cognitivo	Esquema de integración (relevar · configurar · probar · certificar)
Plantilla (compañero, si la entrega debe conformar a una expectativa de presentación)	Capa 1 · Interacción	Confección de un instrumento canónico en un formato o regla del cliente	Esquema de confección (relevar expectativa · confeccionar sobre instrumento canónico · validar)

El **Wingtraining**, citado en la tabla, es el esquema canónico de desarrollo de una Capability en cinco pasos: **workshop con el SME** (*subject-matter expert* — el experto humano cuyo saber se transfiere al agente), **creación**, **personalización**, **ALFA** (validación con el SME) y **BETA** (validación en operación real). Su desarrollo pertenece a la práctica de entrega, no a este canon; el test de clasificación de más abajo lo usa como criterio porque es la prueba operativa de que un alcance es transferencia de saber — y no integración ni formato.

¿Cuál es la estructura conceptual de un proyecto agentivo de entrega?

Un proyecto agentivo de entrega se estructura como una **Capability protagonista** (Capa 2) acompañada típicamente de uno o más **Conectores** (Capa 4) y una o más **Plantillas** (Capa 1). Además, la propia Capability produce sin esfuerzo adicional sus **instrumentos de información** — reportes y dashboards canónicos —, que quedan implícitos en la entrega: no requieren esquema de desarrollo propio porque emergen de la Capability y su capa de interacción. La Plantilla aparece solo cuando uno de esos instrumentos debe conformar a una forma específica del cliente.

¿Capability o entregable de otra capa? — el test

Para decidir si un alcance es Capability en sentido estricto, los tres tests siguientes deben ser **sí**:

1. **¿Es saber-hacer cognitivo?** ¿Interpreta, decide o razona sobre un dominio — más allá de solo conectar o solo formatear?

2. **¿Tiene un SME identificable?** ¿Hay un humano experto cuyo saber se transfiere al agente?
3. **¿Pasa por los cinco pasos de Wingtraining sin forzar?** ¿Tiene sentido aplicarle workshop SME · creación · personalización · ALFA · BETA?

Si uno o más son **no**, el alcance no es Capability: pertenece a otra capa — Conector o Plantilla, según el mapeo de la sección *Capability, Conector y Plantilla* — o, si es saber cognitivo pero subordinado a una Capability mayor, es **feature** de esa Capability (ver la sección siguiente).

La estructura jerárquica — el árbol del saber

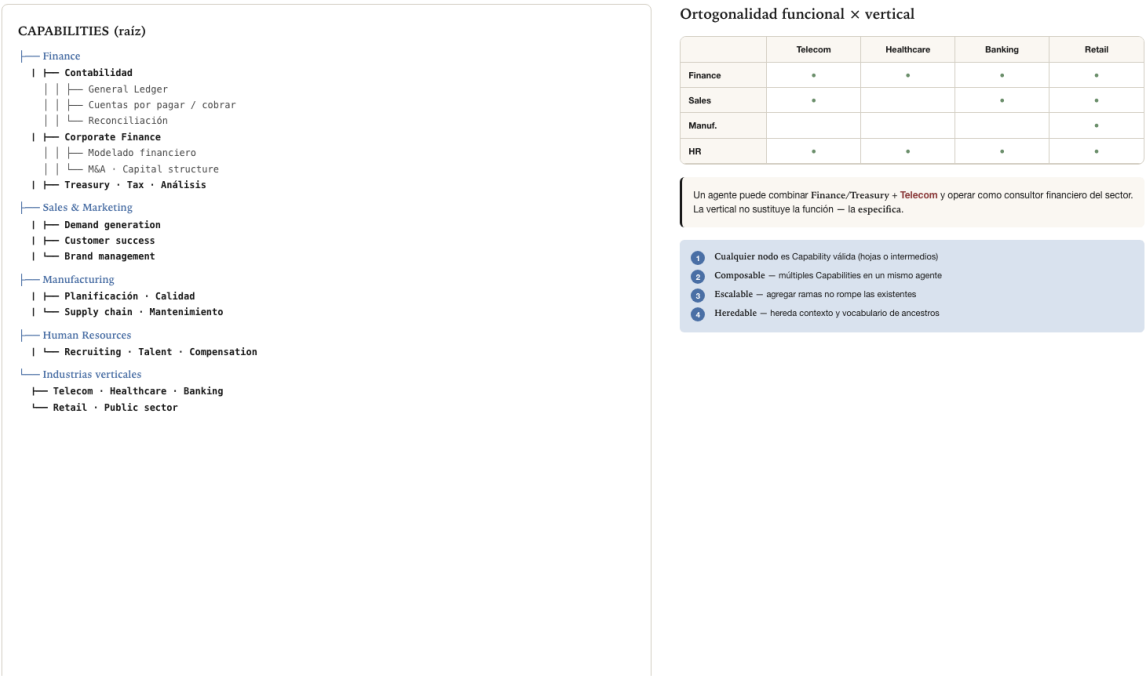


Figura 29: El árbol del saber + ortogonalidad funcional × vertical

Las Capabilities se organizan en un **árbol jerárquico**. La estructura no es decoración — es la forma natural en que el conocimiento profesional está organizado en cualquier disciplina seria. Los analistas financieros piensan jerárquicamente: las finanzas tienen sub-disciplinas (contabilidad, corporate finance, treasury, tax), cada sub-disciplina tiene áreas (general ledger, cuentas por pagar, reconciliación), cada área tiene prácticas específicas. La organización jerárquica refleja cómo el profesional domina el campo.

A continuación un ejemplo del árbol canónico que la spec propone como punto de partida. El árbol no es exhaustivo — se extiende según las necesidades de cada implementación —, pero ilustra la estructura (figura arriba).

Cuatro reglas estructurales gobiernan el árbol. La primera: **cualquier nodo es una Capability válida**. Tanto los nodos hoja — *Reconciliación*, *Demand generation*, *Calidad* — como los nodos intermedios — *Contabilidad*, *Sales & Marketing*, *Manufacturing* — son Capabilities válidas. La granularidad la decide el contexto. Un agente especializado en operación contable diaria adopta nodos hoja con alta

especificidad; un agente orquestador que coordina varios especialistas adopta nodos intermedios con visión panorámica; un agente multiespecialista combina varias raíces.

La segunda regla: **es composable**. Capabilities pueden compartirse entre agentes. Una Capability *Customer success* puede vivir en el agente de un equipo comercial y simultáneamente en el agente de un equipo de soporte, sin duplicación. Esto es crítico operativamente — la organización no necesita re-construir el saber para cada agente nuevo, sino que asigna Capabilities ya construidas según el rol del agente.

La tercera regla: **es escalable**. Se agregan ramas nuevas sin romper las existentes. La Capability *Telecom* puede subir en sofisticación — agregar sub-ramas *Network APIs*, *5G core*, *Customer experience* — sin que las otras ramas del árbol cambien. Esta propiedad es lo que permite que el árbol crezca con la organización: cada vez que la organización entra en un dominio nuevo, agrega ramas; cada vez que profundiza en un dominio existente, extiende ramas. El árbol nunca necesita reescritura completa.

La cuarta regla: **es heredable**. Una Capability hereda contexto y vocabulario de sus ancestros. Una Capability *General Ledger* no necesita re-explicar lo que es Contabilidad ni Finanzas — esa contextualización viene del árbol. Esto importa en la implementación porque permite que las Capabilities hojas sean más concisas: solo necesitan describir lo que es específico de su nodo, no lo que ya está implícito en el camino del árbol.

Anatomía de una Capability

La especificación canónica de una Capability incluye nueve componentes que desplegamos uno por uno. Los componentes uno y nueve son metadata; los componentes dos a siete son el cuerpo del saber-hacer; el componente ocho es el binding con la capa de acción.

El primer componente es **identidad**: nombre canónico más posición en el árbol. La identidad es lo que hace que la Capability sea referenciable — los agentes la invocan por nombre, las políticas se aplican a Capabilities específicas, los logs registran qué Capability se invocó.

El segundo componente es **vocabulario**: términos técnicos del dominio que el agente debe reconocer y usar correctamente. Un agente con Capability *General Ledger* debe distinguir débito de crédito sin titubeos, debe conocer la diferencia entre asientos manuales y automáticos, debe usar correctamente el concepto de período fiscal. El vocabulario es lo que permite al agente conversar con profesionales del dominio sin sonar amateur.

El tercer componente es el **conocimiento procedimental**: cómo se hacen las tareas típicas del dominio. Cómo se reconcilia una cuenta. Cómo se valida una factura. Cómo se prepara un cierre mensual. El conocimiento procedimental es secuencial y operativo — describe pasos, no solo conceptos.

El cuarto componente es el **conocimiento declarativo**: hechos y reglas verificables del dominio. Que las cuentas por pagar tienen vencimientos. Que los ingresos se reconocen cuando se cumplen ciertos criterios. Que la conciliación bancaria debe ocurrir mensualmente. El conocimiento declarativo es factual y permanente — describe qué es verdadero del dominio.

El quinto componente son las **heurísticas**: reglas de decisión profesionales del dominio. Cuando una factura se desvía más del cinco por ciento del monto típico, escalar. Cuando una conciliación tiene más de diez transacciones sin hacer match, suspender el proceso y escalar. Cuando un proveedor tiene tres reclamos en seis meses, marcar para revisión. Las heurísticas son juicio profesional codificado — capturan no qué es verdadero del dominio, sino qué decisión tomaría un profesional en cada situación.

El sexto componente son las **tools asociadas**: tools de Capa 4 que la Capability invoca típicamente. La Capability *General Ledger* invoca tools que consultan la base de datos contable, que ejecutan asientos, que generan reportes. La Capability declara qué tools usa para que el agente sepa qué necesita disponible cuando ejerce esa Capability.

El séptimo componente son las **Capabilities padres**: posición jerárquica de la cual la Capability hereda contexto. Una Capability *General Ledger* declara que es hija de *Contabilidad*, que a su vez es hija de *Finance*. La declaración de los padres es lo que permite la herencia descrita en la sección anterior.

El octavo componente es el **estado de madurez**: Borrador / Vigente / Deprecado. El estado de madurez es metadata operacional — permite que la organización gestione el ciclo de vida de sus Capabilities. Una Capability en Borrador todavía se está validando; una Vigente está aprobada para uso productivo; una Deprecada es legacy que se mantiene por compatibilidad pero no debe usarse en agentes nuevos.

El noveno componente es **versión**: trazabilidad de evolución. Las Capabilities cambian con el tiempo — las prácticas profesionales evolucionan, las regulaciones cambian, el saber acumulado se refina. Las versiones permiten que la organización rastree la evolución del saber y que distintos agentes puedan operar con versiones distintas según sus requisitos.

Los componentes dos a cinco — vocabulario, procedimental, declarativo, heurísticas — son lo que **diferencia profundamente** una Capability bien construida de un prompt elaborado. Un prompt da contexto al modelo de manera transitoria, una sola conversación. Una Capability codifica saber profesional con persistencia y modularidad. La diferencia es estructural: el agente puede consultar la Capability *General Ledger* sin que el saber contable inunde todas sus otras conversaciones — puede tener al mismo tiempo la Capability *Marketing* sin que los frameworks de marketing contaminen el rigor contable. Esta separación es lo que permite a los agentes operar con la disciplina del profesional que cambia de marcos según la tarea.

Features de una Capability

Una Capability expone operaciones internas. La spec canoniza el término **feature** para nombrarlas — equivalente práctico de lo que otros vocabularios llaman *feature*, *operation*, *skill* o *method*. Una feature es una operación interna que la Capability expone; comparte con la Capability contenedora su modelo de datos, su SME, su instalación y su runtime. Una Capability única de costeo, por ejemplo, expone como features la asignación de costos, la rentabilidad y el *pricing*: tres dimensiones de un mismo saber-hacer, no tres Capabilities.

¿Capability o feature interna? — el test

Un alcance se trata como Capability propia si y solo si los tres tests siguientes son **sí**:

1. **¿Independencia operativa?** ¿Puede instalarse y operar sin la otra capacidad?
2. **¿Identidad cognitiva?** ¿Tiene modelo de datos y SME distintos de la otra capacidad?
3. **¿Reusabilidad?** ¿Tiene valor para más de un consumidor o contexto fuera de este caso?

Si uno o más son **no**, el alcance es **feature** de la Capability contenedora, no Capability propia.

Convención de IDs

Una implementación MAY adoptar una convención de identificadores para trazar el árbol de entregables: la Capability (o entregable) lleva ID `E<n>` —`E1`, `E2`—; la feature lleva ID compuesto

`F<n>.<m>` —`F1.1`, `F1.2`— donde `<n>` es el ID de la Capability contenedora. La convención es opcional; lo canónico es que el constructo feature tenga nombre.

¿Qué patologías previene el test?

Distinguir feature de Capability previene dos patologías observadas en proyectos reales:

- **Inflación del codominio de entregables** — dimensiones de una misma Capability se documentan como Capabilities separadas, multiplicando el inventario y diluyendo la portabilidad. El ejemplo de costeo ya citado —asignación, rentabilidad y *pricing* tratadas como tres Capabilities cuando son features de una sola— es el caso típico.
- **Esconder lock-in** — alcances que sí son Capabilities propias, con lifecycle, modelo de datos y reusabilidad independientes, terminan empotrados como "sub-capacidades" de otra, escondiendo que podrían instalarse y portarse por separado.

Los dos tests se aplican en orden: el primero decide si el alcance es siquiera cognitivo; recién entonces el segundo decide su granularidad dentro de la Capa 2. Si el alcance no es cognitivo, se clasifica directamente como Conector o Plantilla por su capa, sin correr el test de feature.

¿Cómo opera la cognición sobre Capabilities?

Dado un agente con un conjunto de Capabilities activas, cuando el agente recibe una solicitud, el procesamiento sigue un flujo canónico de siete pasos.

Paso uno: la solicitud del usuario llega al agente. El agente la recibe en lenguaje natural — *"reconcilía la cuenta de ingresos del mes pasado"*, por ejemplo.

Paso dos: la cognición clasifica el dominio de la solicitud. Esta operación es de routing semántico — el agente identifica que la solicitud es sobre contabilidad, específicamente reconciliación. Esta identificación se basa en el vocabulario que las Capabilities activas le proveen al agente.

Paso tres: la cognición selecciona las Capabilities relevantes. En el caso del ejemplo, *Finance/Contabilidad/Reconciliació* es la Capability más específica aplicable. La cognición la selecciona junto con sus ancestros — Contabilidad, Finance — para tener todo el contexto heredado.

Paso cuatro: la cognición aplica el conocimiento procedimental y declarativo de las Capabilities seleccionadas. Sabe qué pasos componen una reconciliación, qué reglas debe seguir, qué errores típicos puede encontrar.

Paso cinco: la cognición invoca los tools que la Capability indica. Consulta la base de datos contable para los asientos del mes pasado, consulta el extracto bancario, ejecuta el proceso de match.

Paso seis: la cognición compone la respuesta usando el vocabulario del dominio y las heurísticas de la Capability. Reporta los matches encontrados, los items sin match, las recomendaciones de cómo proceder con las discrepancias. Usa terminología que un profesional contable reconocería como correcta.

Paso siete: si la solicitud es repetitiva — el agente ya hizo reconciliaciones similares en el pasado —, Pattern Recognition sugiere generar un Botlet que automatice el ciclo uno a seis para futuras solicitudes similares. La cognición evalúa la sugerencia y, si las condiciones son apropiadas (frecuencia alta, patrón estable), genera el Botlet.

Una observación importante: la Capability **no se ejecuta** — es ejecutada por la cognición. Esto importa estructuralmente: la misma Capability puede ser aplicada con distinta profundidad — rápida y superficial, o lenta y exhaustiva — según el modelo cognitivo y el modo de operación del agente. Un agente con cognición sofisticada puede aplicar la Capability *General Ledger* con todo el rigor de un controller experimentado; un agente con cognición más limitada puede aplicar la misma Capability con la profundidad de un junior. La Capability define el saber-hacer; la cognición define cuán profundamente lo aplica.

Las industrias verticales como Capabilities raíz

El árbol incluye una raíz dedicada a **industrias verticales** — Telecom, Healthcare, Retail, Banking, Public sector. Esto es deliberado: cada vertical tiene su propio vocabulario, sus propias regulaciones, sus propios procesos canónicos. La existencia de una raíz vertical separada de las raíces funcionales (Finance, Sales, Operations) refleja una propiedad estructural del saber profesional: las verticales y las funciones son **ortogonales**.

Un consultor financiero generalista tiene saber funcional — Finance — pero no saber vertical específico. Un consultor de finanzas para telecom tiene los dos: Finance (la función) y Telecom (la vertical). El consultor de telecom puede aplicar frameworks financieros generalistas, pero también conoce las particularidades del sector — los modelos de revenue assurance específicos de telecom, las regulaciones del regulador sectorial, los sistemas operacionales típicos de un operador. El conocimiento vertical es **adicional**, no sustituto, del conocimiento funcional.

El árbol de Capabilities refleja esta ortogonalidad. Un agente puede tener simultáneamente Capabilities de *Finance/Treasury* y de *Industrias verticales/Telecom* — y la combinación produce un agente que opera con saber funcional y vertical al mismo tiempo, exactamente como el consultor experto del sector.

Esta arquitectura tiene dos consecuencias importantes. La primera: **la especialización vertical no es prompt — es Capability**. Un "agente legal" o "agente médico" o "agente de telco" no es un prompt sofisticado del modelo general. Es un agente que carga la Capability vertical correspondiente, con su vocabulario, heurísticas y conocimiento normativo propio. Esto distingue Capabilities de System Prompts genéricos: el saber vertical es modular, persistente, y se compone con otras Capabilities no verticales sin contaminación.

La segunda consecuencia es comercial. Esta arquitectura explica el éxito comercial de los **especialistas verticales** del mercado actual: Cursor (coding), Harvey (legal), Jasper (marketing), Fin (customer support). Lo que estos productos venden no es "un GPT especializado" en su vertical — es una Capability vertical robusta que la cognición aplica con confianza. La diferencia con un GPT genérico no es marketing; es estructural. El usuario que prueba Cursor para programar no nota que la diferencia es la Capability *Coding* que la herramienta carga; nota que las respuestas son correctas más frecuentemente que con GPT genérico, y eso es exactamente lo que la Capability bien construida produce.

> *La Capability vertical es la diferencia entre un agente útil y un agente serio para el dominio.*

Marketplace de Capabilities

Una propiedad emergente de la arquitectura es que las Capabilities admiten un **mercado**. Una Capability bien construida puede distribuirse entre AgencyDomains, versionarse y mantenerse por un proveedor especializado distinto del operador del AgencyDomain, cobrarse por suscripción o licencia, y auditarse por terceros respecto a su corrección y completitud.

Esto da forma a una **economía de Capabilities** análoga a la economía de paquetes de software open source. Quien construye y mantiene Capabilities expertas — por ejemplo *General Ledger* IFRS-compliant, o *Telecom 5G core* — puede operar como proveedor especializado sin construir agentes ni AgencyDomains propios. Su producto es la Capability misma. Su modelo de negocio es licencia o suscripción de la Capability. Sus clientes son organizaciones que operan AgencyDomains y necesitan saber-hacer profesional verificado.

La economía emergente tiene precedentes claros. La industria del software cuenta con economías similares de componentes especializados desde hace décadas: librerías compiladas que se venden o licencian, módulos certificados para frameworks específicos, configuraciones de mejores prácticas que las consultoras venden como activos. La economía de Capabilities sería evolución natural de esos modelos al campo agentivo.

La especificación normativa del **protocolo de Marketplace de Capabilities** — formato de paquete, modelo de versionado, sistema de firmas, modelo de cobro — es **trabajo abierto** en la versión 1.0 de este libro. Las implementaciones contemporáneas pueden adoptar paquetes ad-hoc; la consolidación como estándar de industria está pendiente. Cuando el consenso llegue, una versión futura del libro lo incorporará como spec normativa.

Localidad y disponibilidad — clasificación operativa de Conectores

La descripción canónica anterior trata el saber-hacer como aplicable en cualquier contexto. La realidad operativa de sistemas con presencia física múltiple — restaurantes con locales, sucursales bancarias, tiendas de retail, plantas industriales — exige una clasificación adicional que la spec formaliza explícitamente: **localidad** y **disponibilidad offline**. La clasificación pertenece a los **Conectores** — la cara de Capa 4 del par que acompaña a la Capability —, porque lo que reside en un lugar y necesita (o no) red es el *acceso*, nunca el saber: la Capability que decide cuándo y cómo invocar no tiene localidad. Se documenta en este capítulo porque el par Capability–Conector se entrega y se razona junto. Sin esta clasificación, las decisiones de qué puede invocarse desde un Botlet edge en modo offline se hacen en la oscuridad.

La clasificación opera sobre **dos ejes ortogonales**:

Eje de localidad

Dónde residen físicamente los componentes del Conector:

- **Cloud-resident** — el Conector vive en un servicio remoto. Ejemplos canónicos: Conector `DTE-SII` (servicio del SII de Chile para emisión de boleta y factura electrónica), `Transbank-Onepay` (gateway bancario), `Stripe-Connect` (procesamiento de pagos). El agente los invoca por red; sin red no hay acceso.
- **Edge-resident** — el Conector vive en el sitio físico, asociado a hardware o sistemas locales. Ejemplos canónicos: Conector `ESC/POS-Printer` (impresora térmica de comandas y boletas con protocolo ESC/POS conectada por USB o serial), `Cash-Drawer` (gaveta de dinero del cajón), `Pinpad-Local` (pinpad de tarjetas conectado al POS), `Sensor-Temperatura` (sensor de cámara de frío conectado por GPIO). El agente los invoca contra el hardware del sitio; no necesitan red para operar.
- **Híbrido** — el Conector tiene un componente local y un componente cloud. Ejemplos canónicos: Conector `Cliente-DTE` (firma localmente el documento, lo encola si no hay red, lo envía al SII

cuando vuelve la red), `Cliente-Pinpad-Procesamiento-Diferido` (autoriza localmente con clave PIN y batch, envía al adquirente cuando vuelve la red). La parte local opera offline; la parte cloud sincroniza cuando hay red.

Eje de disponibilidad offline

Si el Conector puede ejecutarse sin red:

- **Online-only** — requiere red para ejecutar. Sin red, la invocación falla. Los Conectores cloud-resident son típicamente online-only en el sentido estricto, aunque algunos tienen variantes con cliente local que los convierten en híbridos.
- **Offline-capable** — ejecuta sin red. Si su contrato externo exige eventualmente comunicación cloud (un comprobante que debe llegar al SII, una transacción que debe consolidarse en central), **encola** y emite hacia afuera cuando la red vuelve. Los Conectores edge-resident son típicamente offline-capable; los híbridos también lo son por diseño.

Matriz canónica de clasificación

Cada Conector conforme declara explícitamente su posición en la matriz:

	Online-only	Offline-capable
Cloud-resident	DTE-SII (sin cliente local) · Transbank Onepay · API meteorológica	(combinación inusual; típicamente migra a híbrido)
Edge-resident	(combinación inusual)	ESC/POS-Printer · Cash-Drawer · Sensor-Temperatura · Pinpad-Local
Híbrido	(combinación inusual)	Cliente-DTE · Cliente-Pinpad-Procesamiento-Diferido · Sync-Inventario

Conexión con Capa 3 distribuida

La clasificación es necesaria estructuralmente cuando la Capa 3 está distribuida (Capítulo 5 §1). Un Botler edge debe **saber** qué Conectores puede invocar offline. Si no lo sabe, sus Botlets edge intentarán invocar Conectores cloud-resident sin red y fallarán catastróficamente — sin red, ni siquiera el fallback agéntico aplica, porque la cognición vive en cloud.

La regla operativa que la clasificación habilita es directa: **un Botlet edge senior, en un sitio físico sin red, opera invocando exclusivamente Conectores edge-resident y la parte local de Conectores híbridos**. Los cloud-resident y la parte cloud de los híbridos quedan inaccesibles temporalmente; los efectos diferidos (envío a SII, consolidación con central) se encolan; cuando la red vuelve, las colas drenan.

Propiedades exigidas

Propiedad	Nivel	Descripción
Declaración explícita de localidad del Conector	**MUST**	Cloud-resident, edge-resident o híbrido.
Declaración explícita de disponibilidad offline	**MUST**	Online-only u offline-capable.
Especificación del comportamiento offline para offline-capable	**MUST**	Qué hace cuando no hay red, qué encola, cómo drena.
Resolución determinista del componente que se ejecuta en híbridos	**MUST**	Bajo qué condiciones corre el componente local; bajo cuáles invoca el cloud.

Portabilidad de la Capability

La sección anterior clasifica *dónde* reside físicamente el Conector que acompaña a la Capability — cloud, edge o híbrido. Una propiedad distinta, que la spec formaliza explícitamente, es la **portabilidad de la Capability**: una Capability conforme puede instalarse y ejecutarse en **cualquier AgencyDomain conforme**, sin reescritura. Esta portabilidad es lo que vuelve a la Capability **propiedad real del cliente** — no del AgencyDomain que la aloja, ni del hosting que sostiene a ese AgencyDomain.

El argumento es el mismo de no-lock-in que el canon hace para el AgencyDomain, aplicado un nivel más abajo. Así como un AgencyDomain conforme migra a otra plataforma hosting conforme sin quedar cautivo de ella, una Capability conforme migra a otro AgencyDomain conforme sin quedar cautiva de él. El cliente que adquiere una Capability adquiere un activo portable, no un alquiler atado a una plataforma.

¿Las dos portabilidades?

Conviene no confundir dos portabilidades que operan en niveles distintos:

Portabilidad	¿Qué migra?	¿Hacia dónde?
Portabilidad del AgencyDomain	El AgencyDomain completo	A otra plataforma hosting conforme
Portabilidad de la Capability	Una Capability	A otro AgencyDomain conforme

¿Cuál es la relación Capability ↔ AgencyDomain?

La relación es asimétrica y explícita: un **AgencyDomain aloja y ejecuta** Capabilities; una **Capability corre en** un AgencyDomain anfitrión. La Capability es habitante de primer orden de la Capa 2 del AgencyDomain — el saber-hacer que da cognición a sus agentes —, no un recurso de soporte. Esta es la razón por la que la definición canónica de AgencyDomain nombra a las Capabilities entre lo que el AgencyDomain aloja y ejecuta, a la par de los agentes autónomos y los Botlets.

La certificación regulatoria reside en el componente certificado, no en el Botlet

Una propiedad estructural que aparece con fuerza en sistemas agentivos productivos en industrias reguladas — gastronomía, salud, finanzas, retail con DTE, farmacia, telecomunicaciones — y que la spec necesita formalizar explícitamente: **la certificación regulatoria de operaciones reside en el componente certificado que el Botlet invoca — el Conector certificado, con la Capability regulada que porta su saber normativo —, nunca en el Botlet que orquesta**. La separación es necesaria porque la naturaleza generada del Botlet hace imposible certificarlo a priori, y certificarlo a posteriori contradice su naturaleza regenerable.

El problema

El libro define que la cognición genera el código del Botlet (Capítulo 5 §2). Pero algunas operaciones que un Botlet ejecuta están **reguladas**: emisión de DTE bajo norma del SII, cobro con tarjeta bajo PCI-DSS, dispensación farmacéutica bajo registro sanitario, comunicación financiera bajo norma del regulador correspondiente. Para estas operaciones, **la regulación exige certificación del componente que ejecuta la operación**. Un sistema que emite boleta electrónica sin certificación SII no es legal; un sistema que cobra con tarjeta sin certificación PCI no puede operar.

Si la certificación residiera en el Botlet, cada Botlet que ejecuta una operación regulada tendría que ser certificado individualmente. Pero un Botlet es **código generado por la cognición** que se regenera cuando el ambiente cambia. Cada regeneración produciría un Botlet técnicamente distinto que requeriría re-certificación. La certificación regulatoria sobre Botlets convierte el ciclo `95/4/1` en imposibilidad operacional: cada cambio del 1% requeriría un proceso regulatorio.

La solución canónica

La spec resuelve la tensión separando responsabilidades con disciplina, y el reparto respeta la doctrina de capas del capítulo:

- **El Botlet orquesta.** Conoce el flujo del proceso, valida pre-condiciones operativas (¿hay productos?, ¿la mesa está abierta?, ¿el cliente tiene su RUT registrado?), captura el evento, formatea la solicitud según el contrato del componente certificado.
- **El Conector certificado ejecuta la operación regulada.** Recibe la solicitud del Botlet, ejecuta bajo todas las normas que aplican, devuelve el comprobante. El Conector `DTE-SII` recibe el detalle de la venta, firma con el certificado tributario, transmite al SII, recibe folio y timbre electrónico, devuelve el comprobante al Botlet. Es certificable precisamente porque es acceso estable, no saber generado.
- **La Capability regulada porta el saber normativo.** Qué exige la norma, cómo se interpreta, cuándo corresponde boleta y cuándo factura, qué hacer ante rechazo del regulador — el saber cognitivo con que la cognición decide y valida. Vive en Capa 2, se desarrolla con SME del dominio regulatorio, y acompaña al Conector como su par.

La separación tiene tres consecuencias estructurales:

Primera, la certificación es del componente certificable. El Conector `DTE-SII` puede certificarse formalmente — su código es estable, su contrato con el SII es explícito, su comportamiento es auditable. La certificación es trabajo único; vale para todos los Botlets que lo invoquen.

Segunda, los Botlets generados conviven naturalmente con cumplimiento regulatorio. Un Botlet `Cobrar-Mesa-9` que se regenera cuando la cocina cambia su menú no rompe la certificación

tributaria — sigue invocando el mismo Conector `DTE-SII` certificado. La regeneración del Botlet afecta lógica de orquestación, no la operación regulada.

Tercera, la frontera de auditoría queda nítida. Cuando el regulador audita, el AgencyDomain expone: el Botlet (lógica de negocio, mutable, regenerable) y el componente certificado (operación regulada, congelada, auditable). La inspección regulatoria se concentra en el Conector certificado — donde la certificación reside —, mientras la lógica de negocio se gobierna con los mecanismos de Trust del Capítulo 5 §4 sin contradecirse con la regulación.

Patrón canónico

El patrón se aplica a cualquier industria regulada:

Industria	Botlet (orquesta)	Componente certificado (ejecuta la operación regulada)
Gastronomía	`Cobrar-Mesa`	Conector `DTE-SII` (boleta o factura electrónica)
Banca	`Procesar-Pago`	Conector `Gateway-PCI-DSS` (tokenización + autorización)
Farmacia	`Dispensar-Receta`	Conector `Registro-Sanitario` (validación y registro de dispensación)
Telecom	`Activar-Servicio`	Conector `Registro-Subtel` (registro regulatorio de activación)
Salud	`Emitir-Prescripción`	Conector `MINSAL-Receta-Electrónica` (firma médica certificada)

El patrón es uniforme: el Botlet contiene la lógica de negocio mutable; el componente certificado contiene la operación regulada congelada; la Capability regulada aporta el saber normativo con que la cognición gobierna el conjunto. La frontera entre ellos es la frontera entre lo que la organización puede regenerar libremente y lo que debe mantener bajo certificación.

Propiedades exigidas

Propiedad	Nivel	Descripción
Componentes regulados declaran su régimen regulatorio	**MUST**	Qué norma cumple, ante qué regulador, con qué número de certificación.
Componentes certificados son inmutables entre auditorías	**MUST**	El código del Conector certificado no se regenera; cambia solo bajo proceso regulatorio.

Propiedad	Nivel	Descripción
Botlets pueden invocar componentes certificados sin restricción	**MUST**	El contrato del componente es estable; el Botlet lo invoca como cualquier otro.
Auditabilidad de la frontera	**MUST**	El log distingue claramente operaciones del Botlet (lógica de negocio) de operaciones del componente certificado (operación regulada).

Capabilities y Botlets — la relación

Capabilities y Botlets viven en capas distintas y resuelven problemas distintos, pero interactúan de manera estructurada. Capability vive en Capa 2 (Cognición). Es saber-hacer. Botlet vive en Capa 3 (Autonomía). Es hacer aprendido. Capability tiene forma de vocabulario más procedimientos más heurísticas más tools. Botlet tiene forma de código tradicional ejecutable. Capability es persistente y versionada. Botlet es auto-regenerable y efímero. Capability se aplica cuando el agente reconoce su dominio. Botlet se invoca cuando el agente reconoce el patrón. Capability es creada por humanos expertos o por proveedores especializados. Botlet es creado por la cognición del agente, automáticamente.

La interacción canónica es la siguiente: una Capability puede dar lugar a múltiples Botlets. El agente que aplica *General Ledger* repetidamente para una empresa específica empieza a generar Botlets que automatizan los pasos rutinarios del proceso — clasificar transacciones, conciliar cuentas, generar reportes mensuales — sin invocar la cognición. La Capability sigue siendo la misma; los Botlets son el residuo eficiente de su aplicación reiterada en un contexto particular.

Esta relación es lo que permite que el sistema agentivo escale económicamente. Las Capabilities son el saber estable que la organización adquiere, mantiene, evoluciona. Los Botlets son el residuo eficiente que la cognición genera al aplicar Capabilities reiteradamente en contextos específicos. La organización invierte en Capabilities; los Botlets emergen del uso. La inversión en saber genera ahorro en operación.

Conformidad

Una implementación de Capabilities conforme a esta especificación debe satisfacer los siguientes requisitos:

Requisito	Nivel
Estructura jerárquica en árbol	**MUST**
Cualquier nodo es Capability válida	**MUST**
Composabilidad entre Capabilities	**MUST**
Anatomía con vocabulario + procedimental + declarativo + heurísticas	**MUST**
Versionado explícito	**MUST**
Estado de madurez declarado (Borrador / Vigente / Deprecado)	**MUST**
Selección por la cognición, no ejecución directa	**MUST**

Requisito	Nivel
Verticales como raíz dedicada	**SHOULD**
Marketplace abierto	**MAY** (cuando la spec normativa exista)
Declaración explícita de localidad del Conector acompañante (cloud / edge / híbrido)	**MUST**
Declaración explícita de disponibilidad offline	**MUST**
Portabilidad entre AgencyDomains conformes	**MUST**
Componentes regulados (Capability normativa + Conector certificado): régimen declarado	**MUST**
Conectores certificados: inmutabilidad entre auditorías	**MUST**

Frontera de evolución

Tres áreas activas de evolución de la primitiva Capability merecen mención.

El **marketplace de Capabilities** es la primera. El protocolo normativo aún no está consolidado; cuando lo esté, la versión 2.0 de este libro lo incorporará.

Las **Capabilities cross-vertical** son la segunda. Cómo una Capability puede combinarse con verticales múltiples sin contradicciones — un agente que opera en finanzas para banca y para telecom simultáneamente, por ejemplo — exige refinamiento de los mecanismos de herencia que la versión 1.0 de la spec describe.

La **auditoría de Capabilities** es la tercera. Cómo certificar que una Capability hace lo que dice hacer — que una Capability *IFRS Compliance* efectivamente refleja IFRS y no su aproximación informal — es problema de gobernanza que el campo aún no resuelve. Las soluciones probables vendrán del lado de la auditoría profesional tradicional adaptada al contexto agentivo.

Trust Infrastructure

Hay una asimetría operativa que cualquier organización que ha intentado llevar IA agentiva a producción reconoce con incomodidad: lo que funciona en piloto rara vez funciona en producción enterprise. Un piloto controlado, con un puñado de usuarios sofisticados, supervisado de cerca por el equipo que lo construyó, puede ejecutarse exitosamente sin gobernanza explícita. Los pilotos demuestran capacidad técnica, no aptitud operativa. La distancia entre los dos es exactamente lo que esta sección desarrolla.

La cifra que más preocupa al campo en 2026 es la proyección de Gartner: más del cuarenta por ciento de los proyectos de IA agentiva serán cancelados antes de fines de 2027. Las razones que Gartner identifica son tres: costos imprevistos, valor de negocio poco claro, y controles de riesgo inadecuados. La tercera razón — controles de riesgo inadecuados — es la que conecta directamente con el contenido de esta sección. Las organizaciones cancelan proyectos no porque la tecnología no funcione, sino porque la organización no puede defender lo que la tecnología hace cuando algo sale mal. La diferencia entre un piloto exitoso y un proyecto cancelado es típicamente la madurez de la **infraestructura de confianza**.

Trust Infrastructure es el conjunto de propiedades transversales que permiten a una organización **confiar en que sus agentes operen con autonomía sin perder control**. No es una capa adicional de la Arquitectura Agentiva — es una propiedad que atraviesa las cuatro capas existentes (Interacción,

Cognición, Autonomía, Acceso) y se ejerce en distintos puntos según el pilar específico. Esta sección desarrolla los cinco pilares con el detalle que el lector arquitecto necesita para diseñar y el lector ejecutivo necesita para evaluar.

> *La Trust Infrastructure no es lo que se agrega después de que el agente funciona. Es lo que separa pilotos de producción.*

La urgencia de Trust Infrastructure ya no es solo arquitectónica. Es regulatoria. Singapore IMDA publicó en enero de 2026 el primer framework estatal de gobernanza específicamente para IA agentiva. La Unión Europea hace lo propio con el EU AI Act. NIST con su AI Risk Management Framework. ISO/IEC con la norma 42001. La pregunta ya no es si los reguladores exigirán infraestructura de confianza — es si la organización puede demostrarla auditablemente cuando se la pidan. Las organizaciones que no la tengan operativa enfrentarán, en el horizonte de los próximos veinticuatro meses, una decisión binaria: invertir aceleradamente para alcanzar conformidad, o suspender operaciones agentivas en mercados regulados.

Los cinco pilares

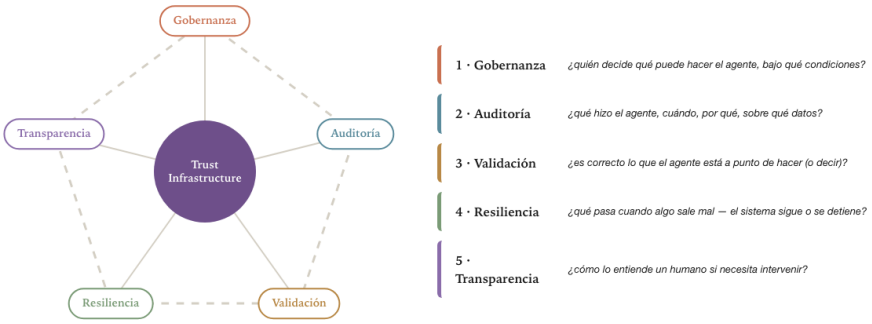


Figura 30: Los cinco pilares de Trust Infrastructure

Cinco pilares constituyen Trust Infrastructure. Cada uno responde una pregunta operativa específica que la organización necesita poder responder cuando alguien — un auditor, un regulador, un cliente — la cuestione.

Pilar	Pregunta que responde
Gobernanza	¿Quién decide qué puede hacer el agente, bajo qué condiciones?
Auditoría	¿Qué hizo el agente, cuándo, por qué, sobre qué datos?
Validación	¿Es correcto lo que el agente está a punto de hacer (o decir)?
Resiliencia	¿Qué pasa cuando algo sale mal — el sistema se detiene o sigue?
Transparencia	¿Cómo lo entiende un humano si necesita intervenir?

Cada pilar se ejerce en una o más capas y se materializa con mecanismos concretos. Las próximas cinco subsecciones desarrollan cada pilar con detalle. Después mostramos el mapa transversal — qué pilar opera principalmente en qué capa — y cerramos con la lectura regulatoria del campo.

Pilar 1 — Gobernanza

El pilar de Gobernanza define el conjunto de mecanismos mediante los cuales la organización establece **qué** puede hacer el agente, **bajo qué condiciones** y **con qué nivel de supervisión**. Es el pilar más visible para quien viene del mundo del IT tradicional, porque tiene el equivalente más directo en mecanismos conocidos — IAM, SSO, RBAC. Pero la Gobernanza agentiva es estructuralmente distinta de la gobernanza tradicional, y confundir las dos es fuente recurrente de proyectos fallidos.

La Gobernanza tradicional pregunta *"¿quién puede ver qué datos?"*. El sujeto del control es un humano con identidad estable; el objeto es un recurso discreto. La pregunta es estática: los permisos rara vez cambian, y cuando cambian es por evento humano explícito (alguien fue contratado, alguien fue promovido, alguien dejó la empresa). La Gobernanza agentiva pregunta *"¿qué puede hacer un agente, bajo qué condiciones?"*. El sujeto es un agente que actúa autónomamente; el objeto es una secuencia de acciones que el agente puede ejecutar con grados variables de impacto. La pregunta es dinámica: las condiciones cambian con el contexto, el momento, el estado del sistema. Las herramientas de Gobernanza tradicional — IAM, SSO, RBAC — son insuficientes para este modelo. Funcionan bien para sujetos humanos con identidad estable; no funcionan para agentes que actúan continuamente con grados variables de autonomía.

Los mecanismos canónicos de la Gobernanza agentiva son cuatro. Las **políticas configurables** son reglas declarativas — no código embebido — que definen qué tools puede invocar el agente, sobre qué datos, en qué horarios, con qué umbrales de impacto. La separación entre política (declarativa) y código (imperativo) es importante: las políticas deben poder cambiarse sin redeploy del sistema, deben poder versionarse independientemente del código, deben poder ser auditadas sin requerir review del código. Los **permisos CRUDLEX** — Create, Read, Update, Delete, List, Execute — son modelo granular de permisos sobre tools y datos, aplicable por usuario, agente o contexto. La operacionalización completa de CRUDLEX vive en el Capítulo 8. La **aprobación humana para operaciones críticas** establece que el agente puede ejecutar operaciones de bajo impacto autónomamente, pero las de alto impacto se detienen y solicitan aprobación. La definición de "alto impacto" es política, no técnica — la organización decide qué umbrales activan aprobación. El **registro de IA** es inventario formal de qué agentes están activos, qué Capabilities aplican, qué tools tienen autorizados, quién los aprobó. Es lo que un regulador verá cuando audite, y lo que la organización debe mantener actualizado y accesible.

Los datos del campo respecto a Gobernanza son crudos — el Capítulo 2 los documenta. La historia que cuentan es consistente: la mayoría de las organizaciones sabe que el problema existe pero no ha invertido lo suficiente para resolverlo, y los reguladores están en camino de forzar la inversión.

Pilar 2 — Auditoría

El pilar de Auditoría define la capacidad de reconstruir, después del hecho, **qué hizo el agente, cuándo, por qué y sobre qué datos** — con suficiente fidelidad para análisis forense, cumplimiento regulatorio o disputa contractual. Es el pilar que la organización necesita cuando algo sale mal: si un agente tomó una decisión que produjo un mal resultado, o ejecutó una acción que un cliente cuestiona, o realizó operación que el regulador quiere examinar, la organización necesita poder reconstruir auditablemente lo que pasó.

Los mecanismos canónicos de la Auditoría son cuatro. El **append-only log inmutable** es el componente central: toda acción del agente queda registrada en un log que no admite modificación retroactiva. Solo se agrega — nunca se edita ni se elimina. El log típicamente está encadenado criptográficamente: cada registro contiene el hash del anterior, formando una cadena verificable donde una alteración retroactiva sería inmediatamente detectable. El **trace de cada acción** registra identidad del agente, capability invocada, tool ejecutado, parámetros, resultado, timestamp, contexto. Cada acción genera un registro completo que reconstruye el estado del sistema en el momento de la decisión. El **lineage de decisiones** es la cadena causal de razonamiento que llevó a una acción particular: qué información consultó el agente, qué Capabilities aplicó, qué heurísticas usó, qué cognición evaluó. Sin lineage, una acción aparece como evento aislado; con lineage, aparece como resultado de un proceso de razonamiento reconstruible. El **identity tagging por acción** asegura que cada acción es atribuible inequívocamente a un agente identificable, no a "el sistema" o a "la IA". La distinción importa para responsabilidad: cuando algo sale mal, debe poder identificarse qué agente específico fue responsable, y por extensión, quién lo configuró, quién aprobó sus capacidades, qué política cubría su operación.

ISACA — la asociación profesional de auditoría — destaca que la IA agentiva presenta un desafío creciente para funciones de auditoría porque sus procesos de decisión carecen de trazabilidad clara cuando el sistema no fue diseñado con auditoría en mente. La observación es importante porque captura una propiedad estructural: un agente que combina cognición probabilística (LLM) con tools determinísticos toma decisiones cuyo camino es difícil de reconstruir si el log no está diseñado para hacerlo. Un log que solo registra "el agente ejecutó X tool con Y parámetros" no es suficiente — necesita registrar también "porque la cognición evaluó Z razonamiento basándose en W contexto". La Auditoría agentiva exige diseño explícito desde el inicio. No emerge naturalmente de una arquitectura agentiva — se construye con disciplina desde el inicio.

> *La auditoría agentiva exige diseño explícito. No emerge naturalmente de una arquitectura agentiva — se construye con disciplina desde el inicio.*

Pilar 3 — Validación

El pilar de Validación define la capacidad de verificar que la respuesta o acción del agente sea **correcta** antes de que afecte al mundo. Es el pilar más reciente en madurar como categoría — la industria de validación de IA es de los últimos tres años — y, simultáneamente, el más crítico para casos de uso donde el costo del error es alto.

La diferencia con la validación tradicional es importante. La validación tradicional verifica formatos: ¿el JSON es válido? ¿la fecha tiene el formato correcto? ¿el monto es número? La validación agentiva

verifica **semántica**: ¿el agente está diciendo la verdad? ¿está actuando dentro de los límites razonables del dominio? ¿está exponiendo datos que no debería? La validación tradicional opera sobre estructura; la validación agentiva opera sobre significado.

Los mecanismos canónicos de Validación son cinco. La **detección de alucinaciones** verifica que las afirmaciones factuales del agente sean consistentes con las fuentes consultadas. Es área activa de investigación; mecanismos contemporáneos incluyen self-consistency (preguntar lo mismo de varias maneras y comparar respuestas), retrieval-augmented verification (consultar fuentes autoritativas antes de afirmar), y model-as-judge (un segundo modelo evalúa la respuesta del primero). La **validación de respuestas estructuradas** verifica que las salidas con schema — JSON, XML, tablas — cumplan el contrato esperado antes de emitirlas. Es la validación más directa, equivalente a la validación tradicional pero aplicada sobre salidas que el modelo generó. La **prompt injection prevention** detecta intentos de manipulación a través de inputs maliciosos disfrazados como datos legítimos. Un usuario que intenta inyectar instrucciones en un campo de comentarios para que el agente las ejecute como si fueran instrucciones legítimas es ataque común que la validación debe detectar. Productos como Lakera y Lasso Security productizan precisamente este mecanismo. La **DLP** — Data Loss Prevention — detecta automáticamente datos personales (PII), información financiera sensible o material clasificado en lugares donde no deberían aparecer. Si un agente está a punto de incluir un número de seguridad social en una respuesta a un usuario externo, la DLP lo detecta y bloquea. La **tokenización** reemplaza datos sensibles por tokens antes de que lleguen al modelo. Permite que el agente razone sobre los datos sin exponerlos al proveedor de cognición externo. La organización mantiene el mapeo token-a-dato en almacén con seguridad reforzada — típicamente HSM (Hardware Security Module) o servicio dedicado.

Informatica formula con precisión la transición que la Validación representa: *"Because agents act without human approval loops, the data they use must be fully trusted, verified, and monitored."* La frase captura algo importante. En sistemas tradicionales, un humano supervisa cada operación importante antes de ejecutarla — es el último loop de validación, hecho de carne. En sistemas agentivos, ese loop humano no existe en cada operación — solo en operaciones críticas que escalan. La Validación tiene que **suplir el loop humano** para todas las demás operaciones, asegurando que lo que el agente está a punto de hacer es correcto antes de hacerlo. Sin esa suplencia, el sistema se queda corto: o ejecuta acciones incorrectas, o requiere supervisión humana en cada operación, anulando la productividad del agente.

La validación de la spec de un componente especialista admite un patrón estructural que merece nota: la **validación por delegación**. El runtime genérico de Capa 3 (el Botler) **hace valer** la validación de la spec de un Botlet sin entender su dominio — orquesta el punto de validación que el Botlet o su proto-Botlet provee, le entrega el contexto genérico que controla (catálogo de Capabilities, identidad, políticas del AgencyDomain) y audita el veredicto en el append-only log. El juicio de qué hace válida a la spec vive en el especialista; el runtime exige y registra la validación sin ejecutarla con conocimiento de dominio. El desarrollo de este patrón vive en el capítulo de Botlets; aquí solo se enlaza el principio con el pilar de Validación.

Con los **Agentlets** (Capítulo 5 §7), el pilar de Validación gana además una sede en la **Capa 3**: la inferencia acotada de cada Agentlet cursa por el punto de control `cognition_call` del handle del Botler, y ahí aplican los mecanismos de este pilar — detección de alucinaciones sobre sus veredictos, validación de salidas estructuradas contra el contrato del charter, DLP y tokenización sobre lo que entra y sale del modelo. La corrección estadística que los Agentlets introducen en la capa de la memoria muscular queda así confinada a Lets declarados y auditables — la disciplina de delgadez que la Bounded Concerns Architecture enseña, ejercida dentro del Mundo Agentivo.

Pilar 4 — Resiliencia

El pilar de Resiliencia define la garantía de que el sistema sigue operando — y la organización conserva control — cuando algo sale mal. Es el pilar más cercano a las prácticas de ingeniería de software tradicional — el campo de DevOps y SRE ha desarrollado patrones de resiliencia durante quince años — pero adaptado a las particularidades del sistema agentivo.

Los mecanismos canónicos son cinco. La **garantía de fallback** es la propiedad fundamental que ya describimos en Capítulo 5 §2 (Botlets) y que el §7 extiende al Agentlet: si un Botlet falla catastróficamente, la cognición ejecuta la tarea manualmente; si un Agentlet no resuelve dentro de su charter, escala a la Cognición plena; si la cognición falla, la operación escala al humano. Esta garantía es lo que distingue al sistema agentivo conforme a esta spec de cualquier "automatización con IA" frágil. El **manejo de errores estructurado** asegura que los errores son tipificados, accionables, propagados con contexto suficiente para que el siguiente nivel decida. Un error que dice "algo falló" no es manejo estructurado; un error que dice "API X devolvió código Y, parámetro Z, en operación W del agente V" es manejo estructurado. El **sandboxing** asegura aislamiento de ejecución de las unidades de Capa 3 y del código generado, con límites estrictos sobre qué pueden tocar. Detalle en la sección de Botlets. Los **circuit breakers** detienen y notifican cuando un agente o Botlet falla repetidamente, antes de seguir consumiendo recursos. Es patrón clásico de resiliencia adaptado al contexto agentivo: si un Botlet ha fallado N veces consecutivas, se detiene su ejecución automática y se escala al humano para revisión. El **rate limiting** establece límites configurables sobre frecuencia de invocaciones, tanto a la cognición (controlar costo) como a tools externos (proteger sistemas downstream). Sin rate limiting, un agente con un loop mal diseñado puede agotar los recursos del sistema en horas.

El principio de no-detención que la spec garantiza — declarado en Capítulo 5 §2 — es lo que permite a la organización delegar operación a agentes con la confianza de que un fallo aislado no detiene el negocio. La resiliencia es lo que hace esa confianza razonable.

Continuidad de negocio operacional vs garantía de fallback agéntico

La garantía de fallback que el Pilar 4 describe supone **cognición disponible**: cuando el Botlet falla por cambio de ambiente, la cognición rescata. Este supuesto se cumple en la mayoría de los escenarios — la cognición vive en cloud altamente disponible y los Botlets fallan ocasionalmente por cambios menores que la cognición resuelve sin esfuerzo. Pero en sistemas productivos físicos — locales sin red, hardware caído, energía cortada — **la cognición tampoco está disponible**, y la continuidad operacional necesita un protocolo adicional que la garantía de fallback agéntico no cubre.

La spec distingue por tanto **dos mecanismos complementarios** que resuelven escenarios distintos:

Garantía de fallback agéntico — la cognición ejecuta cuando el Botlet falla por cambios de ambiente. Es propiedad de la Arquitectura Agentiva, codificada en la spec del Botlet (§2). Cubre la inmensa mayoría de los fallos: el ambiente cambia, el Botlet detecta el cambio, la cognición rescata. Esta es la propiedad que produce la trayectoria de madurez del Botlet desde junior hasta senior.

Continuidad de negocio operacional — protocolos manuales documentados para cuando el Botlet senior cae por causas exógenas y la cognición tampoco está disponible. Es propiedad operacional, equivalente a la que cualquier negocio tradicional ya tiene cuando se le cae el sistema (corte de energía, hardware caído, red catastrófica, proveedor crítico abajo). No depende de la spec agentiva — depende del protocolo del cliente.

Las dos no se excluyen: se complementan. La primera resuelve el aprendizaje del Botlet y la mayoría

de los fallos operativos. La segunda cubre el residuo exógeno catastrófico que ningún sistema previene completamente.

Conexión con la madurez del Botlet

La conexión entre los dos mecanismos y la trayectoria de madurez del Botlet (§2) merece tratamiento explícito porque revela una propiedad estructural del sistema agentivo:

La garantía de fallback agéntico es lo que produce la madurez. Cada vez que el Botlet falla por un cambio de ambiente, la cognición rescata, regenera y devuelve operación — y esa regeneración es justamente lo que produce la incorporación progresiva de variantes hasta alcanzar madurez senior. Sin fallback agéntico, el Botlet quedaría atrapado en su versión inicial, sin manera de aprender. **Sin fallback agéntico, el Botlet no madura.**

La continuidad operacional, en cambio, no incide en la madurez. Opera sobre Botlets ya maduros que caen por causas exógenas, no por aprendizaje pendiente. Cuando un Botlet senior cae porque el local sufrió corte de energía, no es problema de aprendizaje — es problema operacional que el protocolo de continuidad debe resolver (caja manual, registro en papel, conciliación posterior).

Mecanismo	¿Cuándo opera?	¿Qué resuelve?	¿Quién lo provee?
Garantía de fallback agéntico	Botlet junior, en aprendizaje, o senior con variante nueva	Cambios de ambiente que el Botlet no anticipó	La spec agentiva (Capa 2 + Capa 3)
Continuidad de negocio operacional	Botlet senior caído por causa exógena, sin cognición disponible	Continuidad operativa cuando ningún componente computacional opera	Protocolo del cliente (procedimiento manual documentado)

¿Por qué importa la distinción?

Sin la distinción, los planes de continuidad operacional se confunden con la promesa agéntica. Un cliente lee "garantía de fallback" en la documentación del producto y asume que cubre cualquier fallo, incluyendo cortes de energía. Cuando ocurre el corte y descubre que el sistema no opera, atribuye el fallo a la arquitectura agentiva — y concluye que "el sistema falla".

Reconocer las dos propiedades como mecanismos separados pero complementarios reduce ansiedad sobre offline y deja claro qué resuelve la arquitectura y qué resuelve el protocolo operacional del cliente. La conversación con el cliente cambia: ya no se promete que "nunca pasa nada"; se promete que "los fallos por aprendizaje los maneja la arquitectura, los fallos exógenos los maneja el protocolo de continuidad — y ambos están documentados".

La distinción conceptual entre fallback agéntico y continuidad operacional es la sede de esta sección. Su operacionalización —el protocolo de campo por sitio, los modos de degradación y las marcas de log con que se registra la transición a continuidad— vive en el Capítulo 8, donde se desarrollan las exigencias operativas (**MUST** en su casi totalidad, incluida la trazabilidad de la transición a modo continuidad).

Pilar 5 — Transparencia

El pilar de Transparencia define la capacidad del humano de entender, en tiempo real, **qué está haciendo el agente y por qué** — con detalle suficiente para intervenir si es necesario. Es el pilar

que conecta los otros cuatro: la Gobernanza define qué puede hacer, la Auditoría registra qué hizo, la Validación verifica qué está por hacer, la Resiliencia asegura que sigue operando — la Transparencia asegura que un humano puede entender todo lo anterior.

Los mecanismos canónicos son cinco. La **observabilidad completa** entrega tracing end-to-end de cada operación, métricas de latencia, costo, calidad, y eventos estructurados. Es el equivalente agentivo de los sistemas de observabilidad tradicionales (Datadog, New Relic, Splunk) adaptados a las particularidades del sistema agentivo. Las **métricas operativas** miden tasa de éxito de Botlets, frecuencia de regeneración, latencia de cognición, consumo de tokens, errores por capa. Estas métricas son lo que un equipo de operaciones consulta diariamente para entender la salud del sistema. Los **traces consultables por humanos** aseguran que la traza de una decisión es legible por un humano técnico, no solo por otra máquina. Esto importa para auditoría reactiva: cuando algo salió mal y un humano necesita reconstruir qué pasó, debe poder leer los traces directamente, no depender de un sistema de análisis automatizado intermedio. Las **alertas proactivas** notifican cuando el agente detecta que está cerca de un límite, fallando con frecuencia inusual, o tomando decisiones de alto impacto. La proactividad importa: el sistema no espera a que el humano consulte para informar problemas; informa antes de que se vuelvan críticos. Los **dashboards de gobernanza** dan vista al responsable humano: qué agentes están operativos, qué hacen, con qué éxito, sobre qué recursos. Es interfaz de control para la persona que gobierna el sistema.

La observabilidad de IA es categoría madura del mercado. Productos como Langfuse, LangSmith, Helicone, Arize AI, Braintrust, Weights & Biases cubren distintas capas. La transparencia agentiva no exige construir estos productos desde cero — exige integrarlos coherentemente con los demás pilares de Trust Infrastructure. La organización adopta los productos que mejor se ajusten a su stack y los integra con el resto de la infraestructura.

Contrato declarativo de calidad

Los pilares de **Resiliencia** y **Auditoría** se ejercen mejor cuando lo que la organización debe auditar y sostener está **declarado**, no enterrado en código. El **contrato declarativo de calidad** es el mecanismo que lo habilita: cualquier Botlet conforme MAY declarar sus atributos de calidad como propiedades estructuradas, no como código embebido. Trust Infrastructure los lee de manera uniforme — sin acoplarse a la implementación de cada Botlet.

Los atributos canónicos del contrato son cinco. La **Frescura** declara la antigüedad máxima admisible de los datos que el Botlet consume. El **SLA** declara la latencia esperada extremo a extremo, expresada como percentiles (`p50`, `p99`). La **Política de degradación** declara el comportamiento ante fallo — `refuse` (rechaza y no entrega), `warn_and_show` (advierte y muestra de todas formas), `show_last_valid` (entrega el último resultado válido conocido), `agentic_fallback` (escala a la cognición). La **Audiencia** declara la política `RLS/CRUDLEX` aplicable a quién puede consumir la manifestación. La **Política de refresh** declara cómo se renueva — `on-demand`, `scheduled` o `push`.

Declarados así, estos atributos se vuelven mecanismos transversales que sirven a dos pilares a la vez. Para la **Resiliencia**, la Política de degradación y la Política de refresh son configuración auditable y no lógica frágil dispersa por la implementación: el runtime sabe qué hacer ante fallo y cuándo renovar sin leer el cuerpo del Botlet, y la Frescura y el SLA dan umbrales explícitos contra los cuales medir si un resultado sigue siendo confiable. Para la **Auditoría**, los cinco atributos permiten que Trust Infrastructure los audite uniformemente, los curse por **políticas globales** del AgencyDomain (una política puede endurecer la Frescura mínima o vetar `warn_and_show` para una clase de operación) y los reporte como **métricas estándar** comparables entre Botlets. La organización no inventa un

formato por Botlet; declara contra un vocabulario común que el append-only log y los dashboards de gobernanza ya entienden.

Mapa transversal — ¿qué pilar opera en qué capa?

PILAR / CAPA	CAPA 1 INTERACCIÓN	CAPA 2 COGNICIÓN	CAPA 3 AUTONOMÍA	CAPA 4 ACCESO
Gobernanza políticas, CRUDLEX, aprobación	–	–	–	PRINCIPAL
Auditoría append-only log, lineage	–	–	–	PRINCIPAL
Validación alucinación, prompt injection, DLP	–	parcial	–	PRINCIPAL
Resiliencia fallback, sandboxing, circuit breakers	–	–	PRINCIPAL	–
Transparencia observabilidad, traces, alertas	transversal	transversal	transversal	transversal

ausente · parcial · principal — pilar opera centralmente en la capa · transversal — opera en todas las capas

1 · La Capa 4 concentra la mayor carga
Gobernanza, Auditoría y Validación final operan principalmente en Capa 4 — donde la cognición se convierte en acción real.

2 · La Resiliencia vive en la Capa 3
La continuidad operativa se sostiene en la autonomía persistente — Botler, Botlets con fallback.

3 · La Transparencia es propiedad transversal
Atraviesa las cuatro capas. Exige diseño explícito de instrumentación en cada una, no agregada al final.

Figura 31: Mapa transversal · qué pilar opera en qué capa

Los cinco pilares se ejercen en distintas capas de la Arquitectura Agentiva, como sintetiza la figura anterior.

Tres lecturas del mapa son útiles. La primera: **la Capa 4 concentra la mayor carga**. Gobernanza, Auditoría y Validación final operan principalmente en Capa 4. Es coherente con su naturaleza: la Capa 4 es donde la cognición se convierte en acción real, y por tanto donde se ejerce el control. Las decisiones de qué se puede hacer, qué se está por hacer, y qué se hizo, todas pasan por Capa 4. La segunda: **la Resiliencia vive en la Capa 3**. La continuidad operativa se sostiene en la autonomía persistente del agente, su Botler, sus Botlets con fallback. Cuando el sistema agentivo "sigue funcionando" a pesar de que algún componente falló, esa continuidad la asegura la Capa 3. La tercera: **la Transparencia es propiedad transversal**. No vive en una capa específica — atraviesa todas. Esto exige diseño explícito de instrumentación en cada capa, no agregada al final como capa adicional. Cada capa emite eventos, cada capa tiene métricas, cada capa contribuye al log de auditoría.

Trust Infrastructure y los reguladores

La infraestructura de confianza no es decisión solo arquitectónica. Es decisión de **conformidad regulatoria** ante un marco creciente de regulación específica para IA agentiva. Los principales frameworks que el campo enfrenta a inicios de 2026 son:

El **EU AI Act** de la Unión Europea, vigente con aplicación gradual entre 2024 y 2026. El **NIST AI Risk Management Framework** de Estados Unidos, vigente como framework voluntario pero adoptado por industrias reguladas. La **ISO/IEC 42001**, publicada en 2023 con certificación voluntaria pero creciente adopción enterprise. El **MGF — Model AI Governance Framework for Generative AI** de Singapore IMDA, publicado en enero de 2026 y notable por ser el primer framework estatal específicamente para IA agentiva (el nombre conserva la IA generativa; el cuerpo gobierna agentes). Los **lineamientos del World Economic Forum** sobre onboarding y gobernanza de agentes. Las **guías de la NACD** — National Association of Corporate Directors — para boards corporativos.

Patrón común: todos estos frameworks exigen — en términos formalmente distintos pero funcionalmente equivalentes — los cinco pilares descritos en esta sección. No es coincidencia. La lista de pilares emerge del análisis funcional del problema, no de imitación a un regulador particular. Cualquier regulador serio que analice los riesgos del sistema agentivo llega a una lista similar: gobernanza, auditoría, validación, resiliencia, transparencia. La ecuación es estructural.

La consecuencia operativa para la organización es que invertir en Trust Infrastructure no es solo decisión arquitectónica — es **inversión regulatoria**. Una organización que adopta los cinco pilares correctamente se posiciona para satisfacer los cuatro frameworks principales simultáneamente — EU AI Act, NIST AI RMF, ISO/IEC 42001, IMDA MGF —, porque convergen en exigencias funcionales similares. Una organización que los descuida queda expuesta a los cuatro a la vez, sin defensa estructural.

Conformidad

Una implementación de Trust Infrastructure conforme a esta especificación debe satisfacer:

Requisito	Nivel
Los cinco pilares ejercidos en alguna capa	**MUST**
Append-only log inmutable de toda acción	**MUST**
Permisos CRUDLEX granulares por usuario / agente / contexto	**MUST**
Garantía de fallback ante fallo	**MUST**
Trazabilidad consultable por humanos	**MUST**
Detección de alucinaciones	**SHOULD**
Prompt injection prevention	**SHOULD**
DLP / tokenización de datos sensibles	**SHOULD**
Aprobación humana configurable para operaciones críticas	**MUST**
Conformidad con al menos un framework regulatorio reconocido	**SHOULD**
Distinción explícita entre fallback agéntico y continuidad operacional	**MUST**
Protocolo de continuidad operacional documentado para sistemas con presencia física	**MUST**

Frontera de evolución

Tres áreas activas de evolución de Trust Infrastructure merecen mención al cierre.

La **auditoría auditable** es la primera. Protocolos verificables criptográficamente que permitan a un auditor externo verificar el log sin acceso al sistema. Está en intersección con tecnologías de cómputo confidencial y zero-knowledge proofs. Cuando madure, permitirá auditoría externa sin exponer datos operativos al auditor.

El **trust scoring de agentes** es la segunda. Métricas compuestas de confiabilidad que evolucionan con el comportamiento del agente, similar al puntaje de crédito. Permitiría a la organización adoptar agentes con confianza modulada según su trust score: agentes con alto score reciben más autonomía; agentes con bajo score requieren más supervisión.

La **federación de Trust Infrastructure** es la tercera. Cuando dos AgencyDomains colaboran (federación, ver sección 1 de este capítulo), cómo sus respectivas Trust Infrastructures se reconocen y se componen. ¿La política del AgencyDomain A se aplica también a las invocaciones del AgencyDomain B? ¿Cómo se reconcilian políticas contradictorias? Es problema sin solución general en la versión 1.0 de la spec.

Asistente vs Agente Autónomo

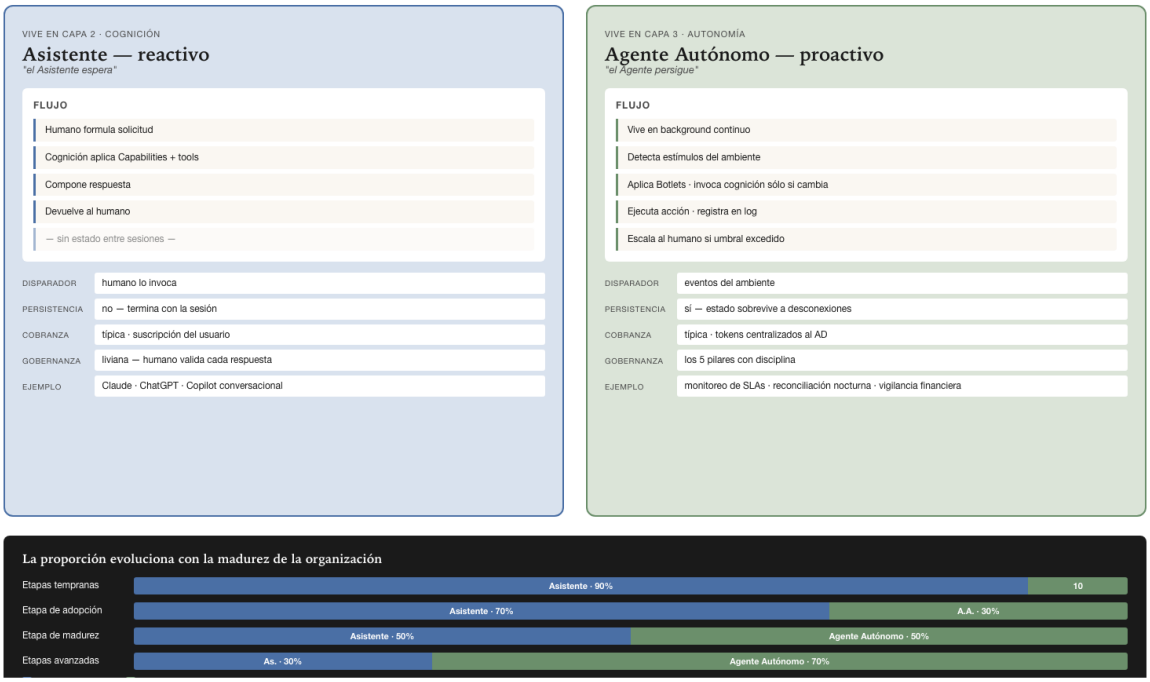


Figura 32: Asistente vs. Agente Autónomo · roles distintos

La industria habla de "agentes de IA" como si fueran una sola cosa. No lo son. Bajo ese término genérico conviven dos modos operativos distintos, con propósitos distintos, modelos económicos distintos, modelos de gobernanza distintos. Confundir los dos es probablemente la fuente más recurrente de proyectos agentivos que fracasan al pasar de piloto a producción.

Los dos modos son el **Asistente** y el **Agente Autónomo**. Esta sección desarrolla la distinción con el

detalle que merece, porque las consecuencias prácticas de mantenerla — o de ignorarla — son enormes.

La distinción

El **Asistente** vive en Capa 2 (Cognición). Es **reactivo**: responde cuando se le solicita, espera input, no mantiene Botlets propios, no tiene vida persistente entre sesiones. El usuario habla con el Asistente, el Asistente responde, la conversación termina. Cuando el usuario regresa más tarde, el Asistente reanuda como si fuera la primera vez — sin memoria del intercambio anterior salvo cuando se la inyecta explícitamente como contexto. Ejemplos paradigmáticos del Asistente son Claude, ChatGPT, Copilot conversacional. Son productos masivos y útiles, pero estructuralmente, son Asistentes — esperan que alguien les hable.

El **Agente Autónomo** vive en Capa 3 (Autonomía). Es **proactivo**: actúa por iniciativa propia, persigue objetivos, mantiene Botlets, los regenera cuando el ambiente cambia, vive en background continuo. El usuario no le habla — el Agente Autónomo opera. Cuando el usuario consulta al Agente Autónomo, no es porque la conversación apenas empieza, sino porque el Agente lleva horas o días operando y el usuario quiere saber el estado. Ejemplos paradigmáticos del Agente Autónomo son los bots que monitorean anomalías en una red, los procesos que ejecutan reconciliaciones nocturnas, los agentes que vigilan SLAs y escalan cuando se aproximan a violación. Son menos visibles que los Asistentes — no aparecen en aplicaciones consumer — pero estructuralmente son donde la mayor parte del valor económico del sistema agentivo se genera.

> *El Asistente espera. El Agente persigue.*

La frase canónica resume bien la diferencia operativa. El Asistente es trabajador de turno: aparece cuando se le llama, responde, se va. El Agente Autónomo es trabajador permanente: vive en el sistema, monitorea continuamente, ejecuta cuando corresponde, escala cuando es necesario.

La distinción **no es jerárquica**. Un Agente Autónomo no es un Asistente mejorado. Son **roles distintos** con propósitos distintos. Un sistema agentivo maduro contiene **ambos modos** y los compone. La organización que solo opera Asistentes se queda corta porque sus agentes no pueden operar en background; la organización que solo opera Agentes Autónomos se queda corta porque no puede atender solicitudes humanas conversacionales. Sistemas serios necesitan los dos.

El término paraguas **Agente** cubre un tercer miembro además de estos dos modos: el **Agentlet** — el agente empaquetado, de charter acotado, que vive como unidad de catálogo en la Capa 3 junto al Botlet. No es un tercer modo del mismo eje — los modos describen cómo opera el agente pleno; el Agentlet es una pieza que el agente pleno engendra y mantiene. Su formalización, y la frontera que lo separa del Agente Autónomo, viven en el §7 de este capítulo.

¿Por qué la distinción importa?

Tres razones operativas concretas justifican la atención que esta sección dedica a la distinción.

La primera razón es que **se diseñan distinto**. El Asistente se diseña para conversación. La latencia debe ser perceptible al humano — segundos típicamente —, la interfaz es textual o por voz, el modo es turno-respuesta. Cuando el humano cierra la conversación, el Asistente termina. El Agente Autónomo se diseña para vida persistente. La latencia es operativa — minutos u horas si conviene —, sin interfaz directa al humano salvo cuando escala, monitoreo continuo de eventos del ambiente. Cuando el humano se desconecta, el Agente sigue.

Una arquitectura que mezcla ambos modos sin distinción produce sistemas confusos: Asistentes con Botlets que el humano no entiende, o Agentes Autónomos que requieren atención constante del humano para operar. Un Asistente que en medio de la conversación lanza un Botlet en background sin notificar al usuario rompe la expectativa del usuario. Un Agente Autónomo que no puede ejecutar nada hasta que el humano abra la aplicación pierde el sentido de su autonomía. La distinción explícita en la arquitectura previene estas confusiones.

La segunda razón es que **se cobran distinto**. El Asistente típicamente vive bajo modelo de **suscripción del usuario** — el humano paga su Claude Pro, su ChatGPT Plus, su Copilot. La cognición se invoca durante las conversaciones. Para el sistema agentivo, esto significa que cada Asistente operando con suscripción del usuario tiene cuota disponible que el sistema no paga directamente. El Agente Autónomo típicamente vive bajo modelo de **tokens centralizados** del AgencyDomain — la organización paga el consumo agregado de los agentes que operan en su nombre. Para el sistema agentivo, esto significa costos predecibles pero materiales: cada decisión, cada validación, cada acción del Agente Autónomo consume tokens que la organización debe pagar.

Esta diferencia económica determina cuándo conviene cada modo. Si el caso de uso permite que el humano esté en el lazo, el Asistente bajo Suscripción del usuario es típicamente más barato — el costo lo absorbe el plan del usuario. Si el caso de uso requiere autonomía continua sin presencia humana, el Agente Autónomo bajo Tokens es la única opción viable, pero con costo predecible que la organización debe presupuestar. Confundir los modos lleva a errores económicos: Agentes Autónomos accidentalmente operando en modo Suscripción que agotan la cuota del usuario en horas, o Asistentes accidentalmente operando en modo Tokens que facturan al sistema lo que debería ir contra la suscripción del usuario.

La tercera razón es que **se gobiernan distinto**. El Asistente opera bajo control inmediato del humano. La validación es conversacional: el humano lee la respuesta antes de actuar, evalúa si es correcta, decide qué hacer con ella. La gobernanza es liviana — bastan permisos básicos. Si el Asistente comete un error, el humano lo nota inmediatamente y lo corrige. El Agente Autónomo opera **sin** control inmediato del humano. La validación debe ser sistémica: la organización confía en que el agente actúe correctamente cuando nadie está mirando. La gobernanza es robusta — exige los cinco pilares de Trust Infrastructure operacionalizados con disciplina. Si el Agente Autónomo comete un error, el humano lo descubre cuando ve el log o cuando la consecuencia se materializa, no en el momento.

> *Vender un Agente Autónomo con la gobernanza de un Asistente es vender un riesgo enmascarado de producto.*

Esta es la razón estructural por la cual los productos que prometen "agentes autónomos" pero gobernanza de Asistente fracasan en producción enterprise. La organización compra esperando autonomía; recibe productos que necesitan supervisión humana constante. La frustración resultante es lo que alimenta la ola de cancelaciones que el Capítulo 2 documenta.

Anatomía operativa

Desplegamos el flujo operativo de cada modo para hacer la distinción concreta.

El Asistente

El flujo del Asistente es lineal y conversacional. El humano formula una solicitud. La cognición — Capa 2 — la recibe. La cognición aplica las Capabilities relevantes para entender el dominio de la solicitud. Si necesita información adicional, invoca tools. Compone la respuesta. La devuelve al humano. El humano

sigue conversando o cierra la sesión. Cuando el humano se va, el Asistente no persiste — salvo que el sistema implemente memoria explícita (que es feature, no comportamiento default).

Lo característico del Asistente es que **no opera cuando el humano no está presente**. La cognición está disponible bajo demanda; cuando no hay demanda, no hay actividad. Esto es eficiente para casos de uso conversacionales pero es limitación severa para casos donde el trabajo necesita ejecutarse en momentos predecibles o ante eventos externos.

El Agente Autónomo

El flujo del Agente Autónomo es continuo y proactivo. El agente vive en background. Detecta estímulos del ambiente — cambios en datos, alertas, eventos. Cuando un estímulo activa una respuesta, aplica los Botlets relevantes. Si el Botlet ejecuta exitosamente, la operación termina. Si el Botlet falla o el caso es nuevo, el agente invoca cognición para resolver. Ejecuta la acción correspondiente — invoca un tool de Capa 4. Registra todo en el append-only log. Si la operación cruza umbrales de impacto definidos por política, escala al humano. Vuelve a esperar el siguiente estímulo.

Lo característico del Agente Autónomo es que **vive persistente**. Su vida es independiente de cualquier sesión humana. El agente opera mientras la organización opera, no solo cuando alguien le habla. Esto exige infraestructura de respaldo — persistencia de estado, monitoreo continuo, gobernanza activa — pero produce capacidad operativa que el Asistente no puede entregar.

¿Cómo cooperan en un sistema maduro?

Un sistema agentivo maduro **contiene ambos modos** y los compone. La composición típica funciona así: el Agente Autónomo opera continuamente en background ejecutando objetivos; el Asistente atiende solicitudes humanas que típicamente consultan el estado del Agente Autónomo o solicitan ajustes a su operación.

Cuando un CFO pregunta a su Asistente *"¿cuál es el estado de cashflow esta semana?"*, el Asistente no recalcula nada — consulta el estado que el Agente Autónomo financiero ha estado manteniendo continuamente. La conversación es **rápida** porque el trabajo pesado ya se hizo en background. El Asistente sirve como interfaz humana al estado que el Agente mantiene.

Cuando el mismo CFO ajusta los umbrales de cashflow — *"de ahora en adelante, escalame cuando el saldo proyectado caiga bajo X"* —, el Asistente comunica el ajuste al Agente Autónomo, que lo incorpora en su lógica continua. La interacción humana es momentánea; el efecto opera persistentemente.

Esta composición no es opcional para sistemas serios. Una organización que solo opera Asistentes tiene sistema agentivo limitado: los humanos deben pedir activamente cada cosa. Una organización que solo opera Agentes Autónomos tiene sistema agentivo intransigente: los humanos no pueden conversar con el sistema, solo recibir alertas o consultar logs. Sistemas maduros necesitan los dos modos cooperando.

Anti-patrones recurrentes

Tres anti-patrones recurrentes producen los fracasos que la industria documenta.

Anti-patrón A: vender Asistente como Agente Autónomo

Un producto que requiere que el humano lo invoque cada vez **es Asistente**, aunque su marketing diga "agente autónomo". El criterio operativo es directo: si cuando el humano se desconecta el sistema deja de

hacer trabajo, no es Agente Autónomo. Es Asistente. La consecuencia de este anti-patrón es que el cliente compra esperando autonomía y recibe asistencia con vocabulario inflado. La frustración subsiguiente es predecible: el cliente compara lo prometido con lo recibido, descubre la brecha, cancela.

Anti-patrón B: construir Agente Autónomo con arquitectura de Asistente

Un sistema que pretende ser autónomo pero opera invocando cognición en cada acción. Funciona en piloto. Falla en producción por costo y por velocidad. La economía detrás de este anti-patrón — por qué los Botlets son la respuesta arquitectónica que evita el colapso al escalar — se desarrolla en Capítulo 5 §2 (Botlets). Aquí basta retener el síntoma: si el sistema deja de funcionar económicamente cuando el volumen pasa de piloto a producción, está incurriendo en este anti-patrón.

Anti-patrón C: gobernar Agente Autónomo con políticas de Asistente

Asumir que basta con permisos básicos sobre datos cuando el agente opera autónomamente. Es error grave. El Agente Autónomo opera sin supervisión humana inmediata; necesita la gobernanza robusta descrita arriba —los cinco pilares de Trust Infrastructure—, no solo controles de acceso. La consecuencia: el agente actúa fuera de límites razonables sin que nadie lo note hasta el incidente. Es la causa típica de los comportamientos riesgosos que el Capítulo 2 documenta como mayoritarios en el campo.

La diferencia entre gobernanza de Asistente y gobernanza de Agente Autónomo es categórica. Para el Asistente, el humano que lee la respuesta antes de actuar cierra el lazo de validación; basta con que el sistema no permita acciones obviamente prohibidas. Para el Agente Autónomo, el humano no está en el lazo — la validación, la auditoría, los límites de impacto, la trazabilidad, todo debe ser sistémico (el pilar de Validación que suple ese lazo humano se desarrolla en el Capítulo 5 §4). Aplicar gobernanza de Asistente a un Agente Autónomo es construir sistema sin red de seguridad bajo el trapecio.

La evolución cooperativa

A medida que el sistema agentivo madura, la proporción de trabajo ejecutado por Agentes Autónomos crece respecto al ejecutado por Asistentes. Esta progresión es propiedad observable del campo, y refleja la transición de la organización a través de la Línea Nadella.

En las **etapas tempranas**, típicamente noventa por ciento del trabajo agentivo opera en modo Asistente: el humano sigue al timón, el Asistente lo ayuda con cada tarea, el Agente Autónomo es marginal. En las **etapas de adopción**, la proporción cambia a setenta por ciento Asistente y treinta por ciento Agente Autónomo: las primeras funciones — típicamente operacionales repetitivas — pasan a operación autónoma. En las **etapas de madurez**, la proporción se equilibra alrededor del cincuenta por ciento de cada modo: el Agente Autónomo opera funciones completas mientras el Asistente atiende consultas humanas. En las **etapas avanzadas**, la proporción se invierte — el Agente Autónomo ejecuta el setenta por ciento del trabajo y el humano interviene principalmente para definir reglas, supervisar, manejar excepciones.

> *La Línea Nadella separa al mundo donde dominan los Asistentes del mundo donde dominan los Agentes Autónomos.*

Esta progresión es lo que el Capítulo 2 describió como la transición de la **empresa en línea** a la **empresa en tiempo real**. Una organización que vive con noventa por ciento de Asistentes es empresa en línea — humanos asistidos por IA que esperan ser invocados. Una organización que vive con setenta por

ciento de Agentes Autónomos es empresa en tiempo real — sistemas que operan autónomamente con humanos gobernando el conjunto.

¿Cómo identificar el modo correcto?

Para una tarea dada, ¿conviene Asistente o Agente Autónomo? Cuatro criterios ayudan a decidir.

El primer criterio: **¿el humano debe ver cada decisión?**. Si la respuesta es sí — porque la decisión requiere juicio humano, porque la responsabilidad regulatoria exige supervisión, porque el costo del error es muy alto —, conviene Asistente. Si la respuesta es no — porque la decisión es repetitiva con criterios claros, porque el volumen es demasiado alto para supervisión humana, porque la velocidad lo exige —, conviene Agente Autónomo.

El segundo criterio: **¿la tarea es disparada por el humano o por el ambiente?**. Si la dispara el humano — el humano formula la pregunta, el humano solicita la operación —, conviene Asistente. Si la dispara el ambiente — un evento externo, un cambio en datos, una alerta de monitoreo —, conviene Agente Autónomo.

El tercer criterio: **¿la tarea es esporádica o continua?**. Si es esporádica o variable — sucede pocas veces al día, en momentos impredecibles —, conviene Asistente. Si es continua o repetitiva — sucede muchas veces, con regularidad —, conviene Agente Autónomo.

El cuarto criterio: **¿importa la latencia conversacional?**. Si el humano espera respuesta — la conversación tiene dinámica de turno-respuesta —, conviene Asistente. Si la operación se ejecuta en background sin presión de latencia inmediata, conviene Agente Autónomo.

La regla práctica que sintetiza los cuatro criterios: si los cuatro apuntan a Asistente, usar Asistente. Si los cuatro apuntan a Agente Autónomo, usar Agente Autónomo. Si la mezcla es ambigua, **diseñar ambos modos cooperando**: un Asistente que consulta el estado mantenido por un Agente Autónomo en background. Esta composición es la que sistemas maduros operan.

Conformidad

Una implementación que ofrece ambos modos conforme a esta especificación debe satisfacer:

Requisito	Nivel
Distinguir explícitamente Asistente de Agente Autónomo en API y documentación	**MUST**
Asistente vive en Capa 2; no requiere Capa 3	**MUST**
Agente Autónomo vive en Capa 3; persiste estado entre sesiones	**MUST**
Agente Autónomo ejerce los cinco pilares de Trust Infrastructure	**MUST**
Componibilidad: Asistente puede consultar estado del Agente Autónomo	**SHOULD**
Distinción de modelo de cobro entre los dos modos	**SHOULD**
Prevención de los tres anti-patrones	**MUST**

Facetas

Las primitivas que las cinco secciones anteriores describieron — AgencyDomain, Botlet, Capability, Trust Infrastructure, Asistente vs Agente Autónomo — viven principalmente en las Capas 2, 3 y 4 de la Arquitectura Agentiva. La Capa 1 (Interacción) había quedado sin una primitiva propia: solo descrita como **regímenes de generación** (Capítulo 4 §1) y como **composición de Botlets de presentación** (shell, vista, operación). Faltaba nombrar la pieza atómica con que esas superficies se construyen.

Esta sección formaliza la **Faceta** como **sexta primitiva canónica** — la unidad mínima de la Capa 1, instrumento que el agente invoca durante conversación o ensambla en Botlets de presentación.

> *El Botlet es memoria muscular del agente. La Faceta es instrumento que el agente toma mientras piensa.*

Definición

Una **Faceta** es un componente atómico reusable de la Capa 1 (Interacción) que ofrece al usuario una forma específica de interacción no conversacional: una pizarra de dibujo a mano alzada, un catálogo-selector, una matriz de colores, un calendario, un mapa clickeable, un slider, un ordenamiento drag-and-drop, un selector de archivos, una vista de canvas configurable. Una de las muchas caras que la interacción con el usuario puede tomar en un momento dado.

La Faceta es **instrumento**, no proceso. Vive y opera en la Capa 1. Es invocada por la cognición durante interacciones activas o ensamblada por Botlets de Capa 1 (shells y vistas) como pieza de su composición interna.

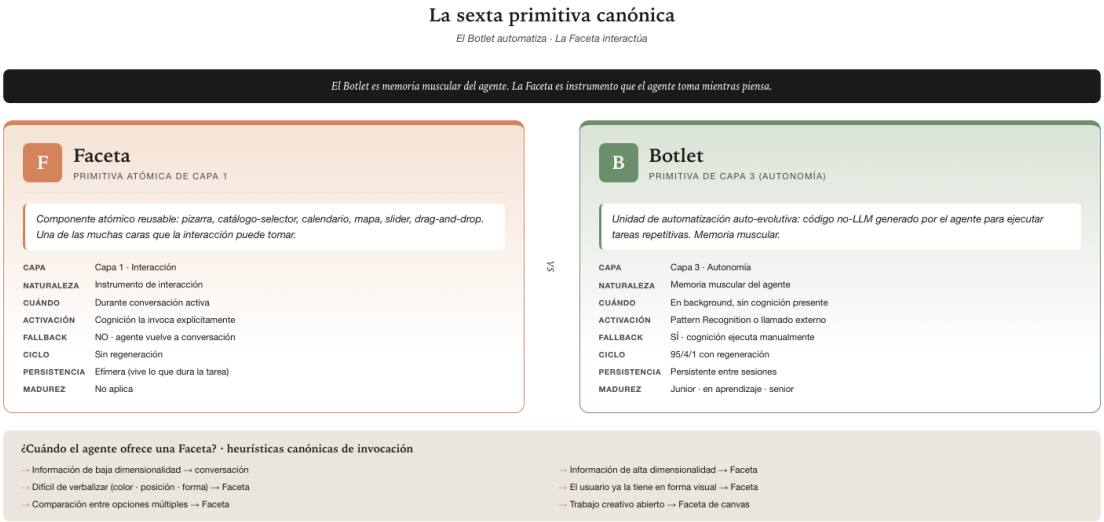


Figura 33: Faceta vs Botlet · dos primitivas, dos capas, dos naturalezas

Faceta vs Botlet — la distinción canónica

La Faceta y el Botlet son dos primitivas de software del agente. Se confunden fácil porque ambas son piezas con identidad propia que el agente usa para hacer cosas. La distinción canónica:

Eje	**Faceta**	**Botlet**
Capa	Capa 1 (Interacción)	Capa 3 (Autonomía)
Naturaleza	Instrumento de interacción	Memoria muscular del agente
¿Cuándo opera?	Durante conversación activa	En background, sin cognición presente
Activación	Cognición la invoca explícitamente	Pattern Recognition o llamado externo
Garantía de fallback	NO — si falla, agente vuelve a texto	SÍ — la cognición ejecuta manualmente
Ciclo de vida	Efímera (vive lo que dura la tarea)	Persistente entre sesiones
Regeneración	No tiene ciclo de regeneración	Ciclo `95/4/1` con regeneración
Fases de madurez	No aplica	Junior · en aprendizaje · senior
Reutilización	Catálogo plano de instrumentos	Catálogo por capacidad y dominio

La diferencia ontológica importa: **el Botlet automatiza; la Faceta interactúa**. Un Botlet ejecuta saber consolidado sin participación humana inmediata. Una Faceta abre un canal de comunicación visual-manipulativa con un humano que está activo en la conversación.

Dos usos canónicos

Uso 1 · Invocación directa por la cognición

El agente, durante una conversación, **decide** que la información que necesita del usuario se obtiene más rápido a través de una Faceta que continuando el diálogo verbal. Compone una superficie efímera con una o varias Facetas, la presenta al usuario, recibe la información, y la conversación continúa. La superficie efímera **no es un Botlet** y no persiste — vive lo que dura la tarea inmediata.

Este uso realiza directamente el régimen **GUI generada on-the-fly** del Capítulo 4 §1. La Faceta es la pieza que materializa ese régimen.

Uso 2 · Composición en Botlets de presentación

Los Botlets de **shell** y **vista** (Capítulo 4 §1, *Composición de la Capa 1*) ensamblan Facetas como piezas internas de su construcción. La vista de "detalle de pedido" usa la Faceta de "matriz de productos" + la Faceta de "calendario" + la Faceta de "selector de cliente". El Botlet de vista define la orquestación, el layout, el flujo de datos entre piezas; las Facetas son los instrumentos individuales que el Botlet pone en la pantalla.

Este uso permite que las superficies persistentes (régimen 3 del Capítulo 4 §1) se construyan reutilizando catálogo de Facetas, sin que cada Botlet de vista tenga que reinventar cada componente atómico.

Interacción declarada acotada

Una pieza de información ya materializada — un snapshot autocontenido que el agente forjó en su tiempo de Ingeniería — puede portar interacción sobre sus propios datos sin dejar de ser una pieza reproducible. Pero no toda interacción es del mismo tipo. La **interacción declarada acotada** es la categoría que separa lo que una pieza puede ofrecer de lo que pertenece a otra primitiva.

La distinción se traza entre dos interactividades:

- La **exploración libre** lanza queries nuevas y arbitrarias al origen (drill o pivot ad-hoc), opera sobre un espacio abierto, pierde reproducibilidad, excede `G1` (las generaciones del Botlet, Capítulo 5 §2) y vive fuera del proto-Botlet de información — en otro Botlet o en la cognición misma. Una pieza de información **MUST NOT** absorberla.
- La **interacción declarada acotada** opera sobre el snapshot ya materializado, dentro de un espacio declarado (dimensiones y valores acotados de antemano), mantiene la reproducibilidad, es `G1` —configuración, no código— y vive en la pieza misma, realizada vía Faceta.

¿Qué las distingue?	Exploración libre	Interacción declarada acotada
Query nueva al origen	sí, arbitraria (drill/pivot ad-hoc)	no — opera sobre el snapshot ya materializado
Espacio de interacción	abierto	declarado (dimensiones y valores acotados)
Reproducibilidad	se pierde	se mantiene (misma config + datos → mismo artefacto + mismo set de controles)
Generación	excede `G1`	`G1` (es configuración, no código)
¿Dónde vive?	otro Botlet / cognición (Capa 2 + Capa 1)	en la pieza misma, vía Faceta

Faceta embebida

El mecanismo que realiza la interacción declarada acotada ya es canónico: es la composición de Facetas (Uso 2), aplicada hacia adentro de una pieza materializada. Es la **pieza** la que compone la Faceta — no la Faceta la que compone otras —; la Faceta sigue siendo atómica. Una **Faceta embebida** es una Faceta acotada a una dimensión declarada de los propios datos de la pieza — un filtro, un segmentador, un selector de apariencia. Al activarla, los elementos *data-bound* de la pieza —KPIs y medidas como agregaciones declaradas (`sum`, `ratio`, y similares) sobre el dataset embebido, distribuciones, semáforos— se **recomputan client-side** sobre el subconjunto filtrado: la agregación se declara una vez y se reevalúa al cambiar el subconjunto. La cognición no explora; la Faceta filtra el snapshot.

El caso testigo es un dashboard de asistencia con filtro por área: el usuario elige una o varias áreas y la pieza recalcula sus KPIs y su semáforo sobre el subconjunto.

Esto reconoce un tercer uso de la Faceta — extensión del Uso 2 hacia la pieza materializada —: la distinción Faceta vs Botlet admite este tercer uso sin alterar la naturaleza de la Faceta, que sigue siendo efímera y sin garantía de fallback. Lo nuevo es que la pieza puede componerla para interacción acotada, además de la invocación por la cognición durante conversación.

Comportamiento agentivo asociado

La existencia de la Faceta como primitiva canónica habilita un comportamiento agentivo específico: el agente, durante una conversación, **estima en tiempo real** si la información que necesita se obtiene mejor verbalmente o visualmente. Cuando estima que la vía visual gana, ofrece una Faceta apropiada.

El cálculo es cognitivo del agente, no feature pre-programada del producto. La Faceta como primitiva habilita la decisión; la heurística la ejerce.

Heurísticas canónicas para invocación

Naturaleza de la información	Modalidad recomendada
Baja dimensionalidad + bien estructurada (un sí/no, una fecha, un número)	Conversación
Alta dimensionalidad (múltiples campos relacionados)	Faceta de formulario o composición
Difícil de verbalizar (color, posición, forma, gesto)	Faceta especializada (matriz de colores, mapa, dibujo)
El usuario ya la tiene en forma espacial o visual (un layout, un mapa, un dibujo en papel)	Faceta que reciba esa forma directamente
Comparación entre opciones múltiples	Faceta de catálogo-selector con vista comparativa
Configuración con muchas dimensiones independientes	Faceta de panel con sliders y toggles
Trabajo creativo abierto (no respuesta a pregunta cerrada)	Faceta de canvas o lienzo

Anti-heurísticas (cuándo NO ofrecer Faceta)

- Cuando la pregunta es genuinamente cerrada y verbal — ofrecer Faceta agrega fricción, no la reduce.
- Cuando el usuario está en un canal sin capacidad gráfica (voz pura, IVR, SMS) — la Faceta no es invocable.
- Cuando el costo de cargar la Faceta supera el beneficio de la interacción visual (interacciones de un solo paso, datos triviales).
- Cuando la conversación está en flujo y la Faceta interrumpe inadecuadamente.

El agente que aprende a calibrar estas decisiones — cuándo ofrecer, cuándo no — opera en una Capa 1 plena. El que solo conversa se queda en la mitad del rango interactivo posible.

Cuando una Faceta se compone dentro de una pieza materializada (interacción declarada acotada), la decisión deja de ser conversacional y pasa a ser parte de la configuración de la pieza: el agente declara el espacio de controles en tiempo de Ingeniería, preservando la **reproducibilidad** —propiedad MUST del artefacto de información— ya establecida arriba.

Anatomía de la Faceta

La especificación canónica de una Faceta incluye seis componentes:

1. **Identidad** — nombre canónico (ej: `pizarra-dibujo`, `matriz-colores`, `calendario-rango`) más versión.

2. **Modalidad de interacción** — qué tipo de input acepta (touch, mouse, teclado, gesto), qué tipo de output produce.
3. **Schema de entrada** — los datos que el invocador (cognición o Botlet de vista) le pasa al instanciarla.
4. **Schema de salida** — los datos que devuelve cuando el usuario completa su interacción.
5. **Estado interno** — qué mantiene mientras está activa (selecciones intermedias, edits parciales, undo stack).
6. **Compatibilidad de canal** — qué canales de Capa 1 la soportan (web, móvil, kiosk, no soportada en voz).

Las Facetas se publican en un **catálogo plano**: no hay jerarquía de Facetas porque cada una es atómica. Lo que hay es un set creciente de instrumentos disponibles, indexados por modalidad y por dominio de aplicación.

Catálogo emergente

La industria converge gradualmente hacia un set canónico de Facetas reusables — el equivalente del catálogo de componentes de UI de las eras pre-agentivas (Material, Bootstrap, Ant Design), pero con la diferencia ontológica de que estos componentes son **invocables por la cognición** y no se sirven solo para construir aplicaciones humanamente programadas.

Algunas Facetas canónicas emergentes:

- `pizarra-dibujo` — superficie de dibujo a mano alzada.
- `catalogo-selector` — vista de items con selección.
- `matriz-colores` — selector de paleta o color individual.
- `calendario-rango` — selector de fecha o rango de fechas.
- `mapa-clickeable` — mapa con puntos seleccionables.
- `slider-multi` — uno o varios sliders relacionados.
- `dragdrop-orden` — reordenamiento de items.
- `formulario-dinamico` — formulario con campos generados al vuelo.
- `lienzo-creativo` — canvas abierto para producción de artefactos.
- `selector-archivo` — invocación al sistema de archivos del cliente.

La spec no cierra el catálogo: nuevas Facetas se acuñan a medida que la industria identifica modalidades canónicas que justifican primitiva propia.

Conformidad

Una implementación de Facetas conforme a esta especificación debe satisfacer:

Requisito	Nivel
Identidad y versión declaradas	**MUST**
Schema de entrada y salida explícito	**MUST**
Compatibilidad de canal declarada	**MUST**
Atomicidad — no compone otras Facetas internamente	**MUST**
Distinción explícita Faceta vs Botlet en documentación	**MUST**

Requisito	Nivel
Invocabilidad directa por la cognición durante conversación	**MUST**
Composabilidad dentro de Botlets de shell y vista	**MUST**
Compuesta en una pieza materializada como interacción declarada acotada (espacio de controles declarado, sin queries nuevas)	**MAY**
Preservación de la reproducibilidad de la pieza cuando se compone como Faceta embebida (recómputo client-side, sin invocar Capabilities)	**MUST**
Catálogo público de Facetas disponibles para el AgencyDomain	**SHOULD**
Heurísticas de invocación documentadas para la cognición	**SHOULD**

Frontera de evolución

Tres áreas activas de evolución de la Faceta como primitiva merecen mención.

La **estandarización de catálogo** es la primera. La industria todavía no ha consolidado un set canónico universal de Facetas. Cada plataforma agentiva define el suyo, con intersecciones parciales. La emergencia de un catálogo común con identidades estables permitiría agentes operar sobre cualquier AgencyDomain conforme.

La **federación de Facetas** es la segunda. Cuando dos AgencyDomains colaboran (federación, Capítulo 5 §1), las Facetas que uno expone deben poder invocarse desde el otro. La spec no define todavía un protocolo formal para invocación federada de Facetas — es trabajo abierto.

La **negociación de canal** es la tercera. Una Faceta declara qué canales soporta. La cognición debe negociar — si el usuario está en voz, no puede ofrecer la Faceta; debe degradar a conversación. Sin esta negociación explícita, las superficies fallan en canales no esperados. La spec exige declaración pero no formaliza aún el protocolo de degradación.

Agentlets

En la entrada de una sala de urgencias hay una enfermera de triaje. Su tarea es la misma todo el día: recibir al paciente, evaluarlo, clasificarlo — rojo, amarillo, verde — y derivarlo. La tarea es repetitiva en su forma; ninguna repetición es igual a la anterior. Cada paciente llega con síntomas distintos, historias distintas, señales que no están en ningún manual exactamente como se presentan. La enfermera no puede resolver su trabajo con memoria muscular — cada clasificación exige juicio fresco —, pero tampoco necesita al jefe de medicina interna para cada paciente: su juicio opera dentro de un protocolo acotado, y cuando el caso excede el protocolo, escala. La enfermera de triaje no es un reflejo ni es la deliberación plena del hospital. Es una tercera cosa: **juicio de rutina, empaquetado en un rol**.

Los sistemas agentivos están llenos de trabajo con esa forma. Clasificar los correos que llegan a una casilla operativa. Hacer triaje de las alertas de monitoreo. Resumir cada documento nuevo que entra a un expediente. Extraer las entidades de una factura que nunca se había visto. Juzgar si un mensaje amerita interrupción inmediata o espera al reporte de la mañana. Son tareas **recurrentes en su forma**

pero frescas en cada instancia: el patrón es estable, la ejecución exige interpretación. No maduran hacia el determinismo — no por falta de regeneraciones, sino por naturaleza.

El canon, hasta aquí, les daba dos casas y ninguna era la correcta. La primera casa es el **Botlet**: pero el Botlet es código no-LLM por definición, y forzar una tarea interpretativa a código determinístico produce exactamente la fragilidad que la arquitectura existe para evitar. La segunda casa es la **Cognición plena** de la Capa 2: funciona, pero al costo de la vía más cara del sistema — deliberativa, conversacional, dimensionada para lo genuinamente nuevo — aplicada a un patrón que se repite cien veces al día. Entre la memoria muscular y la deliberación plena faltaba el peldaño del medio.

Esta sección formaliza el **Agentlet** como **octava primitiva canónica** — la unidad hermana del Botlet cuyo cuerpo de ejecución invoca inferencia acotada. El Agentlet es la casa del juicio de rutina: patrón empaquetado por fuera, juicio acotado por dentro.

> *El Botlet es memoria muscular. El Agentlet es juicio de rutina empaquetado. La Cognición se reserva para lo genuinamente nuevo.*

Definición

Un **Agentlet** es una unidad empaquetada de la Capa 3 (Autonomía), hermana del Botlet, cuyo cuerpo de ejecución **invoca inferencia acotada**: un modelo dimensionado a la tarea, aplicado dentro de un charter que el spec declara. Como el Botlet, es instancia configurada de una pieza pre-forjada — el **proto-Agentlet** —, es hospedado por el Botler y es engendrado y mantenido por el agente. A diferencia del Botlet, cada una de sus ejecuciones ejerce juicio: interpreta la instancia concreta que tiene delante en lugar de ejecutar código determinístico.

La frase que fija la doctrina de hermandad: **un Agentlet es un Botlet con juicio adentro — ni más ni menos**. Vive exactamente como su hermano. Hereda por estructura, no por excepción, todo el aparato de los Lets — las unidades empaquetadas de la Capa 3 —: el patrón proto-/instancia, los catálogos y sus efectos de red, la separación código fuente vs spec, la **manifestación** y la **temporalidad** (`discreta`/`continua`), el contrato declarativo de calidad, la cadena de derivación, el append-only log y los verbos del API de operación (`specialize` · `invoke`/`schedule` · `read`/`subscribe` · `status`/`activate`/`deactivate`/`retire`). Lo que cambia es un solo atributo — el cuerpo invoca inferencia — y de ese atributo se siguen, en cascada, las diferencias que el resto de la sección desarrolla.

Cuatro propiedades definen al Agentlet conforme. La primera es que su **charter está declarado en el spec**: qué tarea resuelve, sobre qué entradas, con qué salidas, dentro de qué límites. El Agentlet no elige qué hacer — su inferencia se gasta en *cómo* hacer lo suyo, nunca en decidir *qué* es lo suyo. La segunda es que su **inferencia es acotada**: el spec declara el modelo (dimensionado a la tarea, no el modelo frontera de la Cognición), las Capabilities que puede consultar y el presupuesto por ejecución. La tercera es que **toda su inferencia pasa por el Botler**: el handle controlado que el Botler le entrega incluye el punto de control de cognición, de modo que cada llamada al modelo queda medida, presupuestada, validada y auditada — el bypass es estructuralmente imposible, no solo prohibido. La cuarta es que tiene **garantía de fallback hacia la Cognición plena**: cuando su inferencia acotada no resuelve el caso — el charter no alcanza, la confianza del veredicto queda bajo umbral, la entrada no se parece a nada previsto —, el Agentlet escala y la Cognición de Capa 2 rescata. El proceso nunca se detiene.

La regla de contrabando

La frontera entre el Botlet y el Agentlet es un atributo binario, y la spec la protege en ambas direcciones:

- **Si hay inferencia en el cuerpo, es Agentlet.** Un Botlet con una llamada a modelo escondida en su código es no-conforme — contrabandea costo y no-determinismo hacia una unidad cuyas garantías (costo marginal cero, madurez que converge, offline senior) dependen de no tenerlos.
- **Si no hay inferencia en el cuerpo, es Botlet.** Un Agentlet cuyo charter resultó resoluble con código determinístico es un Botlet mal clasificado — y mal presupuestado: paga tokens por un trabajo que la memoria muscular haría gratis. La cognición, al detectarlo, lo cristaliza: genera el Botlet que lo reemplaza.

La segunda dirección de la regla describe además una **trayectoria natural**: hay tareas que ingresan al sistema como Agentlets — porque al inicio cada instancia parecía exigir juicio — y que, con la operación acumulada, revelan un núcleo cristalizable. La cognición extrae ese núcleo a un Botlet y deja al Agentlet solo el residuo genuinamente interpretativo, o lo retira por completo. El sistema maduro migra trabajo hacia abajo en la escalera del costo: de la Cognición a los Agentlets, de los Agentlets a los Botlets.

Agentlet vs Botlet — la distinción canónica

Eje	**Botlet**	**Agentlet**
Cuerpo de ejecución	Código no-LLM; cero inferencia	Invoca inferencia acotada, vía el handle del Botler
Metáfora	Memoria muscular	Juicio de rutina empaquetado
Tarea natural	Recurrente y **cristalizable** en código	Recurrente en forma, **interpretativa** en cada instancia
Determinismo	Converge con la madurez	No converge — la corrección es estadística por naturaleza
Costo marginal	~0 (cómputo tradicional)	Tokens por ejecución, presupuestados en el spec
Madurez	Junior → senior; senior = fallos solo exógenos	Semántica propia: spec estabilizado, tasa de escalamiento decreciente
Offline	Senior operable offline confiablemente	Solo con modelo edge-resident, declarado en el spec
Fallback	La cognición ejecuta manualmente	Escala a la Cognición plena

Lo que la tabla **no** contiene es tan importante como lo que contiene: proto-/instancia, catálogos, spec, manifestación, temporalidad, contrato de calidad, cadena de derivación, log, verbos de operación, hospedaje por el Botler, génesis por el agente — todo eso es idéntico, porque se predica del género y no de la especie. La spec llama al género por su nombre propio: los **Lets** — el Botler hospeda Lets; el Botlet y el Agentlet son sus dos especies. El nombre deriva de la morfología de la propia familia y es, como Botler, vocabulario normado del canon — no una novena primitiva. La relación canónica del runtime se enuncia en su forma general — **1 Proceso = 1 Botler + N Lets** — y cada contexto la instancia a la especie que corresponda.

Agentlet vs Agente — agenda vs charter

El nombre de la primitiva reclama membresía en la familia del **Agente**, y la spec se la concede: el término paraguas *Agente* cubre, desde esta versión del canon, **tres miembros** — el **Asistente** (Capa 2, reactivo), el **Agente Autónomo** (Capa 3, proactivo, habitante) y el **Agentlet** (Capa 3, empaquetado). La concesión es honesta — la criatura razona de verdad, con un modelo de verdad — pero exige trazar la frontera con el Agente Autónomo con el mismo rigor con que el canon trazó Faceta vs Botlet, porque los dos viven en la misma capa y los dos usan inferencia.

La frontera cabe en una línea: **el Agente tiene agenda; el Agentlet tiene charter.** El Agente Autónomo persigue objetivos — decide qué hacer, cuándo, con qué medios, y engendra las unidades que necesita. El Agentlet ejecuta la tarea que su spec declara — su juicio opera dentro del charter, jamás sobre el charter.

Eje	**Agente Autónomo**	**Agentlet**
Naturaleza	**Habitante** del AgencyDomain	**Pieza de catálogo** — instancia de un proto
Nace por	Provisioning (ciclo de vida de seis fases, identidad de primer orden)	`specialize` sobre un proto-Agentlet
Agenda	Persigue objetivos; decide qué hacer y cuándo	Charter fijo declarado en el spec
Cognición	Plena: bindings propios, árbol completo de Capabilities, multi-LLM	Acotada: modelo dimensionado, Capabilities que el spec declara
Engendra	Genera y regenera Botlets y Agentlets	No engendra nada; es mantenido
Fallback	*Es* el fallback — encima de él solo está el humano	Escala a la Cognición vía el Botler
Gobernanza	Ejerce los cinco pilares de Trust Infrastructure (MUST)	Gobernado por los puntos de control del handle del Botler

El **test de frontera** protege la categoría de desangrarse hacia arriba. Tres preguntas; cualquier respuesta del lado de la autonomía significa que la pieza es un Agente, no un Agentlet:

1. ¿Elige sus propias metas, o las recibe declaradas en el spec?
2. ¿Puede alterar su propio proceso o engendrar otros Lets?
3. ¿Su identidad es de habitante (provisioning) o de instancia (`specialize`)?

Un Agentlet con charter gordo, un loop interno y criterio propio sobre qué perseguir no es un Agentlet avanzado: es un Agente Autónomo disfrazado, operando sin la gobernanza que su naturaleza exige — la versión agentiva del anti-patrón C del §5.

El Botler como tutor — mismo runtime, un punto de control más

El Agentlet no trae runtime propio. Lo hospeda **el mismo Botler** que hospeda a los Botlets, y esta decisión es de principio, no de conveniencia. El Botler es genérico por definición: gestiona ciclo de vida, aislamiento y ejecución de cualquier unidad sin entender su dominio — y llamar a un modelo no es dominio; es un servicio de runtime. La extensión sigue el patrón que el Botler ya practica: junto

a `capability_call` y `log`, el handle controlado que el Botler entrega en cada invocación expone un tercer punto de control — **`cognition_call`** —, la única vía por la cual el cuerpo del Agentlet alcanza un modelo.

La consecuencia de gobierno es el argumento entero: **como toda la inferencia del Agentlet pasa por el handle, Trust Infrastructure la ve completa**. Cada llamada al modelo queda medida (tokens, latencia), presupuestada (contra el límite que el spec declara), validada (los mecanismos del Pilar 3 — detección de alucinaciones, DLP, tokenización — aplican en el punto de control) y auditada (en el mismo append-only log, con la misma identidad, bajo el mismo gobierno que el resto del AgencyDomain). Un runtime paralelo para Agentlets tendría que duplicar toda esa maquinaria — segundo sandbox, segundo esquema de handle, segunda cadena de escalamiento, segunda presencia en la Capa 3 distribuida — para terminar entregando las mismas garantías. La arquitectura plana del Botler ya las entrega.

La preocupación legítima que podría empujar hacia un runtime separado — el perfil de recursos: latencia de modelo, costo por token, eventual aceleración por hardware — es asunto de despliegue, y el canon ya tiene la puerta abierta: los subtipos de Botler se distinguen **por topología y rol de despliegue, nunca por dominio**. Un pool de ejecución segregado para Lets de inferencia pesada es exactamente una distinción de rol de despliegue: un Botler conceptual, N procesos si la operación lo pide.

Conviene dejar explícitas las **dos relaciones** que sostienen a los Lets, porque operan en pisos distintos y las dos cubren a las dos especies por igual:

Relación	¿Quién la ejerce?	Sobre Botlets	Sobre Agentlets
Hospedaje / ejecución — runtime sin agencia: ciclo de vida, aislamiento, handle controlado, log	El **Botler**	Sí	Sí (mismo handle, más `cognition_call`)
Génesis / mantenimiento — decidir que exista, especializarlo, regenerarlo, responder por él	El **agente** (la cognición)	Sí	Sí

El agente es el padre de ambas especies; el Botler es el mayordomo de ambas. Ningún Let tiene casa aparte.

Madurez del Agentlet — estabilización, no convergencia

La trayectoria junior → senior del Botlet converge porque cada regeneración **cristaliza** variantes en código: el senior es determinístico y por eso sus únicos fallos son exógenos. El Agentlet no recorre esa trayectoria — su corrección es estadística por naturaleza y ninguna cantidad de operación la vuelve determinística. Pretender que un Agentlet "madura a senior" en el sentido del Botlet es un error de categoría con consecuencias operativas: nunca se le puede prometer la confiabilidad offline del Botlet senior sobre la misma base.

La spec define para el Agentlet una **semántica de madurez propia**, observable en el mismo append-only log que ya rastrea a los Botlets:

- **Estabilización del spec** — el charter, el prompt operativo y la configuración dejan de cambiar entre revisiones; las ediciones del `specialize` se espacian.
- **Tasa de escalamiento decreciente** — la fracción de ejecuciones que el Agentlet resuelve dentro de su charter sin escalar a la Cognición plena crece y se estabiliza. Es el análogo funcional del `95/4/1`: la proporción entre juicio acotado que basta y juicio pleno que rescata.
- **Calidad sostenida bajo el contrato declarativo** — sus atributos de calidad declarados (frescura, SLA, degradación) se cumplen dentro de umbral por período sostenido.

El **offline** del Agentlet exige declaración explícita, no herencia del hermano. Un Botlet senior opera offline porque no necesita ningún modelo; un Agentlet solo opera offline si su modelo reside en el edge. El spec de todo Agentlet conforme **MUST** declarar la localidad de su cognición acotada — cloud-resident, edge-resident o híbrida, con el mismo vocabulario que la spec ya usa para los Conectores — y su comportamiento sin red. Sin esa declaración, la consecuencia «modo offline trivial» de la topología paralela se rompe en silencio: el sitio que contaba con su vía Autonomía descubre, con la red caída, que la mitad de sus unidades necesitaba un modelo al otro lado del cable.

proto-Agentlet — la pieza pre-forjada del juicio

Como su hermano, el Agentlet rara vez nace de la nada: nace de un **proto-Agentlet** — la pieza pre-forjada que el agente configura en su tiempo de Ingeniería para instanciar un Agentlet específico al caso. El proto-Agentlet contiene el cuerpo (el andamiaje de la tarea interpretativa: la estructura del charter, el esqueleto del prompt operativo, los contratos de entrada y salida, los umbrales de escalamiento); el Agentlet es la instancia configurada. Las dos clases del proto-Botlet aplican sin cambio: un proto-Agentlet **templado** resuelve una función interpretativa y se configura por parametrización acotada (un clasificador de correo operativo, un extractor de campos de factura); un proto-Agentlet **platafórmico** es un motor de juicio genérico cuya especialización vive en configuración composicional y cubre N funciones de su dominio.

La **cadena de derivación** se extiende sin fricción: los casos de uso documentados requieren Lets — cero, uno o varios, de cualquiera de las dos especies —, y cada Let es instancia de algún proto del catálogo. Todo Agentlet conforme **MUST** poder trazarse en esa cadena, y el append-only log **MUST** registrar el proto-Agentlet de origen de cada instancia. Los catálogos comunes — público abierto en AgencyDomains.org, códices propietarios, contratos privados, acuerdos soberanos — acumulan proto-Agentlets con los mismos efectos de red que acumulan proto-Botlets: el implementador n+1 recibe charters, prompts operativos y umbrales refinados por los implementadores 1 a n.

Las **generaciones** (`G1`/`G2`/`G3`) aplican con una lectura natural: en `G1` el agente configura el proto-Agentlet — rellena el charter, ajusta el prompt operativo dentro del andamiaje, fija umbrales — sin escribir su cuerpo. El filo `G1`/`G2` es el mismo del Botlet: configuración que una Capability bien definida evalúa es `G1`; extensión de la lógica interna del proto es `G2`.

La economía de los tres peldaños

Con el Agentlet, la escalera económica del sistema agentivo queda completa. El Capítulo 4 presentó dos vías — Cognición costosa, Autonomía barata —; la vista fina distingue **tres peldaños de costo por ejecución**:

Peldaño	Unidad	Inferencia	Costo marginal
1	**Botlet**	Cero	~0 — cómputo tradicional
2	**Agentlet**	Acotada: modelo dimensionado, presupuesto declarado	Bajo y **presupuestable por Let**
3	**Cognición**	Plena: deliberativa, modelo frontera, árbol completo	Alto — se reserva para lo nuevo

El peldaño intermedio es el que faltaba. Sin él, toda tarea interpretativa recurrente pagaba precio de peldaño 3 — o se forzaba, frágil, al peldaño 1. Con él, la organización asigna cada patrón a su costo natural, y el sistema maduro migra trabajo escalera abajo: la Cognición cede rutinas interpretativas a Agentlets; los Agentlets ceden núcleos cristalizables a Botlets.

La entrada del Agentlet **matiza, no rompe**, la promesa económica de la vía Autonomía. La vía sigue siendo la barata — pero deja de ser uniformemente gratuita: contiene Lets de costo marginal ~0 y Lets de costo acotado, y el mix se declara por Let, no se promedia. Bajo planes de **Suscripción fija**, el argumento del Capítulo 5 §2 se refina en el mismo sentido: los Botlets siguen siendo el mecanismo que hace posible la autonomía sostenida sin agotar la cuota, y los Agentlets consumen cuota — acotada, presupuestada, visible en el log — por lo que su proporción en el mix es una decisión económica explícita de la organización, exactamente como lo es la decisión de qué patrones consolidar en Botlets.

La escalera ordena tres cosas, no solo el costo. Ordena también la **latencia** —un Botlet responde en tiempo de cómputo tradicional; un Agentlet, en el de su modelo dimensionado; la Cognición, en el de la deliberación— y la **dependencia de infraestructura**: el Botlet senior no depende de ningún proveedor de inferencia, mientras que el junior todavía recurre a la Cognición en su fallback; el Agentlet depende del modelo que su spec declara, que puede ser edge-resident solo donde exista un modelo dimensionado con calidad suficiente (frontera abierta, ver «Frontera de evolución»); la Cognición depende de la infraestructura de frontera. Cada patrón que migra escalera abajo responde más rápido y depende menos de un tercero, sin eliminar la dependencia: lo nuevo y todo fallback siguen subiendo a la Cognición.

La decisión tipada — el cuerpo natural del Agentlet. En 2026 apareció el primer modelo público de una clase que no genera texto: recibe el estado de la instancia y un conjunto de preguntas de respuesta tipada —elegir entre opciones, puntuar en un rango, sí o no— y devuelve decisiones con probabilidad calibrada. Es la forma natural del charter de un Agentlet: el spec declara las preguntas y sus tipos, la salida no puede caer fuera del tipo, y la probabilidad calibrada fija el umbral de escalamiento —por encima, el Agentlet resuelve; por debajo, escala a la Cognición, que es la garantía de fallback de la especie—. Tres cautelas que la spec hace suyas: la calibración del Agentlet sigue siendo un problema abierto (ver «Frontera de evolución»), y la que declara el proveedor del modelo se mide antes de apoyar un umbral en ella (contrato declarativo de calidad, Pilar 3); y un modelo que solo se ofrece alojado por su proveedor no es edge-resident, de modo que el Agentlet que lo usa no opera offline y hereda la dependencia de ese proveedor; y una decisión tipada no trae explicación, de modo que lo que debe poder justificarse escala a la Cognición por diseño.

Hay una segunda consecuencia transversal: **el pilar de Validación gana presencia en la Capa 3.** Hasta esta versión del canon, la Validación se ejercía en la Capa 2 (parcial) y en la Capa 4 (principal),

porque la Capa 3 solo contenía código determinístico. Con los Agentlets entra corrección estadística a la zona de la memoria muscular — exactamente la volatilidad que la Bounded Concerns Architecture (Capítulo 3) enseña a confinar —, y los mecanismos del Pilar 3 (detección de alucinaciones, validación de salidas estructuradas, DLP, tokenización) aplican en el punto de control `cognition_call` del Botler, sobre cada ejecución de cada Agentlet. La delgadez del dominio se preserva: el juicio estadístico queda confinado a Lets declarados, gobernados y auditables, nunca disperso por el runtime.

¿Cuándo usar Agentlet — y cuándo no?

La decisión entre las tres casas — Botlet, Agentlet, Cognición — se resuelve con dos preguntas en cascada:

Primera: ¿la tarea es recurrente? Si no — si es única, exploratoria, o su patrón todavía no se estabiliza —, pertenece a la Cognición. La regla práctica de las diez invocaciones del §2 aplica a las dos especies de unidad por igual.

Segunda: ¿cada instancia exige juicio? Si no — si la lógica es cristalizable en código determinístico —, **Botlet**, siempre: es el peldaño más barato y el único que madura hacia el offline confiable. Si sí — si el patrón es estable pero cada ejecución interpreta una instancia nueva —, **Agentlet**.

Tres señales confirman que una tarea es territorio de Agentlet: la entrada es **lenguaje natural o contenido no estructurado** (correos, documentos, transcripciones, descripciones libres); la salida exige **clasificar, resumir, extraer o juzgar** más que calcular o transportar; y los intentos de resolverla con reglas producen **listas de excepciones que crecen sin converger**. Y dos anti-señales devuelven la tarea a sus vecinos: si el "juicio" resultó ser una tabla de decisión estable, es un Botlet que todavía no se cristaliza; si el charter no logra declararse — la tarea exige decidir qué hacer, no solo cómo —, es trabajo de Agente, con la gobernanza de Agente.

Conformidad

Una implementación de Agentlet conforme a esta especificación debe satisfacer:

Requisito	Nivel
Charter declarado en el spec: tarea, entradas, salidas, límites	**MUST**
Inferencia acotada declarada: modelo, Capabilities accesibles, presupuesto por ejecución	**MUST**
Toda inferencia cursada por el punto de control `cognition_call` del handle del Botler	**MUST**
Garantía de fallback: escalamiento a la Cognición plena ante caso fuera de charter o confianza bajo umbral	**MUST**
Cero inferencia fuera del handle (la regla de contrabando, dirección Botlet→Agentlet)	**MUST**
Hospedaje por el Botler genérico — sin runtime paralelo por especie	**MUST**
Trazabilidad: cada ejecución y cada llamada de inferencia en el append-only log	**MUST**

Requisito	Nivel
Trazable en la cadena casos-de-uso → Lets → protos; proto-Agentlet de origen registrado	**MUST**
Declaración de localidad de la cognición acotada (cloud / edge / híbrida) y comportamiento offline	**MUST**
Validación del Pilar 3 aplicada en el punto de control de cognición	**MUST**
Distinción explícita Agentlet vs Agente Autónomo en API y documentación (el test de frontera)	**MUST**
Métricas de madurez propias: estabilización del spec y tasa de escalamiento	**SHOULD**
Cristalización: detección de núcleos deterministas y extracción hacia Botlets	**SHOULD**
Pool de ejecución segregado por perfil de recursos (rol de despliegue)	**MAY**

Frontera de evolución

Tres áreas activas de evolución del Agentlet merecen mención al cierre.

Los **modelos edge para juicio acotado** son la primera. La viabilidad del Agentlet offline depende de modelos pequeños ejecutando en el sitio físico con calidad suficiente para charters acotados. El estado del arte avanza rápido; la spec ya provee el vocabulario de declaración (localidad de la cognición acotada) para que las implementaciones adopten modelos edge sin cambio de contrato.

La **calibración de confianza** es la segunda. El escalamiento del Agentlet a la Cognición plena depende de que el propio Agentlet estime cuándo su veredicto no alcanza el umbral — un problema de calibración estadística abierto. Mientras madura, las implementaciones conservadoras compensan con umbrales bajos: escalar de más cuesta tokens; escalar de menos cuesta confianza.

La **cristalización asistida** es la tercera. La detección automática de núcleos deterministas dentro de charters de Agentlet — la trayectoria Agentlet → Botlet de la regla de contrabando — es hoy juicio de la cognición en su tiempo de Preparación. Su formalización como mecanismo de Pattern Recognition inverso (detectar lo que *dejó* de necesitar juicio) es trabajo abierto.

Con el Agentlet cierra el elenco de construcciones formales de este capítulo — **ocho primitivas canónicas**: AgencyDomain, Botlet, proto-Botlet, Agentlet, Capability, Trust Infrastructure, la distinción Asistente vs Agente Autónomo y la Faceta. Quien haya seguido los Capítulos 4 y 5 tiene en mano el vocabulario constructivo completo con el que la categoría agentiva puede ser razonada y construida — desde el espacio computacional que lo contiene todo hasta la escalera de tres peldaños por la que el trabajo encuentra su costo natural.

El Capítulo 6 desplaza la mirada del sistema individual al mercado. Permite a quien construye o invierte responder con disciplina la pregunta de dónde compite cada actor — propio o ajeno — y por qué un mismo eslabón de la cadena puede ser zona muy disputada o territorio aún abierto.

Capítulo 6 · Mercado

Una arquitectura no vive en el vacío: compite en un mercado. Este capítulo sitúa a cualquier actor mediante la **cadena de valor de IA** —once eslabones × cuatro profundidades—, profundiza en eslabones selectos con *deep-dives* y extiende el modelo al **mundo de carbono**. La §1 desarrolla el modelo general; las secciones siguientes lo aplican.

La cadena de valor de IA

> **Nota sobre fechabilidad de los productos mencionados.** Los listados de productos específicos en este capítulo describen el estado del mercado de IA agentiva a **mayo de 2026**. La estructura conceptual de los eslabones de la cadena de valor es estable; los actores listados son ilustrativos del momento. Lecturas posteriores deben tomar los nombres como instantánea, no como cobertura permanente.

La industria de Inteligencia Artificial se presenta a mayo de 2026 como un ecosistema denso donde decenas de productos, plataformas y frameworks compiten y coexisten. Sin un modelo de clasificación claro, resulta difícil responder preguntas fundamentales que cualquier ejecutivo, arquitecto o estratega serio se hace cuando enfrenta el campo: ¿dónde juega cada actor? ¿qué eslabones domina? ¿dónde hay concentración y dónde hay espacio? ¿en qué territorio compite mi propuesta y contra quién?

La fragmentación del campo no es accidente. Es resultado de un crecimiento explosivo donde cada nuevo entrante construye su propia categoría, escoge su propio vocabulario, define su propio posicionamiento. El consumidor — sea un comprador empresarial, un analista de mercado, un inversionista — termina abrumado por un lenguaje que cada actor moldea a su conveniencia. *Plataforma de agentes, gateway de IA, framework de LLM, infraestructura agentiva, agente autónomo, asistente vertical, marketplace de modelos* — todos términos que circulan sin definición precisa, todos términos que distintos vendors usan con distintos significados.

Este capítulo propone un mapa que disciplina esa conversación. El mapa no resuelve todas las ambigüedades del campo — la industria está demasiado joven para que un mapa único capture toda su complejidad —, pero entrega un marco compartible: un lenguaje preciso que permite situar a cualquier actor en su posición, comparar actores entre sí, y razonar sobre la estrategia de un producto particular respecto al campo más amplio. El mapa que sigue es la **cadena de valor de IA**, en su versión bidimensional. Es contribución original de este libro, derivada del trabajo previo del autor en la conceptualización del campo, y se ofrece como herramienta abierta para la industria, no como propiedad intelectual propietaria.

Las dos dimensiones

El modelo organiza la cadena de valor tecnológica de IA en **dos dimensiones**. Las dos dimensiones operan ortogonalmente — un actor se sitúa en cada una independientemente —, y la combinación de ambas produce el espacio de posicionamiento donde el actor vive. Las dos dimensiones son **cobertura** y **profundidad**.

La **cobertura** es la dimensión horizontal: qué eslabones de la cadena de valor toca el actor. Un actor puede tocar uno solo, varios, o muchos. La cobertura es métrica de **alcance** — cuánto del territorio del campo opera el actor. Un actor con cobertura amplia toca muchos eslabones; un actor con cobertura focal toca uno o pocos.

La **profundidad** es la dimensión vertical: con qué nivel de control opera el actor dentro de cada eslabón. Un actor puede consumir un eslabón superficialmente — usar APIs de terceros — o construir el eslabón profundamente — fabricar la tecnología de base. La profundidad es métrica de **control** — cuánto del eslabón el actor domina. Un actor con profundidad superficial depende de proveedores subyacentes; un actor con profundidad profunda construye el sustrato sobre el cual otros operan.

Cada actor puede posicionarse en uno o más eslabones, a distintas profundidades en cada uno. Un mismo actor puede operar a profundidad Core en su eslabón nativo y a profundidad Plataforma en eslabones adyacentes — es patrón común en el mercado contemporáneo.

El resultado es un **mapa que permite** clasificar cualquier producto de IA según los eslabones que abarca, comparar actores por cobertura y profundidad en la cadena, identificar zonas de concentración y zonas de oportunidad, y posicionar estratégicamente productos propios frente al mercado.

Los once eslabones

La cadena de valor de IA se descompone en once eslabones secuenciales, cada uno con función clara y separable. La separación no es arbitraria: cada eslabón corresponde a una capacidad funcional distinta que un actor puede operar independientemente, con economía y dinámica competitiva propias.

Desplegamos cada eslabón con su descripción funcional. La secuencia no es lineal en el sentido de que un proceso de datos pase por todos los eslabones en orden, pero sí refleja una progresión conceptual desde la materia prima del campo (los datos) hasta donde el agente toca el mundo real (el entorno).

Eslabón 1 · Datos (Data Layer). Adquisición, anotación, gestión de datasets de entrenamiento. Es la materia prima que alimenta los modelos fundacionales. Los actores en este eslabón producen datasets curados, herramientas de anotación, pipelines de procesamiento de datos a gran escala. Sin este eslabón, los modelos no existen.

Eslabón 2 · Modelo (Foundation Model). Modelos base de IA: LLMs y modelos multimodales que proveen capacidades fundamentales de lenguaje, razonamiento y generación. Es donde los grandes laboratorios — OpenAI, Anthropic, Google, Meta, DeepSeek — concentran capacidad. Los actores aquí construyen los modelos que el resto del campo consume.

Eslabón 3 · Acceso (Access Layer). APIs y capas de acceso a modelos. Control de cuotas, autenticación y monetización del consumo. Es donde la inferencia se vende como servicio: APIs de OpenAI, Anthropic, Bedrock de AWS, Vertex AI de Google. También es donde operan productos como gateways de modelo (Portkey) que ofrecen abstracción sobre múltiples modelos.

Eslabón 4 · Agentes (AI Agents). Interfaces conversacionales y asistentes. Desde agentes reactivos — chat — hasta agentes autónomos capaces de ejecutar tareas complejas. Es donde aparecen los productos

Figura 34: Los once eslabones de la cadena de valor de IA

más visibles: ChatGPT, Claude, GPT-4 con plugins, sistemas de agentes orquestados.

Eslabón 5 · Especializaciones (Domain Experts). Agentes autónomos especializados por dominio vertical: coding, legal, marketing, soporte, productividad, memoria del trabajo profesional. Es donde aparecen los especialistas verticales: Cursor para coding, Harvey para legal, Jasper para marketing, Fin para customer support, umeeta para memoria del engagement de consultoría. La diferencia con el eslabón 4 es de profundidad de saber-hacer en un dominio específico.

Eslabón 6 · Runtime (Agent Runtime). Ambiente operativo donde los agentes viven y operan de manera autónoma. Ciclo de vida, persistencia de estado, identidad, scheduling y orquestación multi-agente. Es donde la Capa 3 de la Arquitectura Agentiva se materializa como producto. Eslabón emergente — la mayoría de los actores tradicionales todavía no lo cubren explícitamente.

Eslabón 7 · Firewall (Security Layer). Seguridad, control y governance. Protección contra prompt injection, alucinaciones, filtrado de contenido y auditoría de uso. Productos como Lakera, Lasso Security operan aquí. Es eslabón crítico para producción enterprise — sin firewall, el sistema agentivo no puede operar en industrias reguladas.

Eslabón 8 · Observabilidad (Observability). Monitoreo, trazabilidad, costos y calidad de los sistemas de IA en producción. El ciclo de feedback operacional. Productos como Langfuse, LangSmith, Helicone, Arize operan aquí. Es eslabón maduro — la observabilidad de IA tiene varios productos de profundidad Core compitiendo activamente.

Eslabón 9 · Herramientas (Tools). Capacidades específicas que los agentes pueden invocar. Incluye meta-herramientas: protocolos (MCP), vector databases (Pinecone, Weaviate), frameworks de RAG. Es

donde el agente extiende su capacidad para tocar sistemas específicos.

Eslabón 10 · Integraciones (Integration Layer). Puente entre el mundo IA y el Entorno. Orquestación, transformación y mapeo de lógica de integración entre sistemas. Productos como Zapier, Make, n8n operan en este eslabón en su forma tradicional; el equivalente agentivo todavía es categoría emergente.

Eslabón 11 · Entorno (Environment). Lo externo a la cadena: sistemas empresariales (ERPs, CRMs, bases de datos), mundo físico (IoT, procesos industriales) y sistemas biológicos. Es el eslabón menos desarrollado, y desarrollamos sus implicaciones con detalle en la sección de mundo de carbono.

Los eslabones no son arbitrarios. Cada uno corresponde a una **decisión de diseño operativo** en cualquier sistema de IA productivo. Saltarse un eslabón no es elegancia: es deuda arquitectónica que se paga en producción.

Las cuatro profundidades

Los eslabones definen **dónde** participa un actor en la cadena. Pero dentro de un mismo eslabón, los actores operan a distintos niveles de profundidad. Un actor que consume una API de modelos y otro que entrena el modelo fundacional **ambos participan en el eslabón Modelo**, pero su diferenciación, dependencia y foso competitivo son radicalmente distintos.

El modelo define cuatro niveles de profundidad, de menor a mayor control sobre la capacidad del eslabón. Las cuatro profundidades aplican a cualquier eslabón — un actor puede ser Wrapper en Datos, Plataforma en Modelo, Core en Acceso. La uniformidad permite comparación cruzada entre eslabones distintos.

Wrapper (nivel 1). El actor consume capacidades vía APIs o SDKs de terceros. Agrega experiencia de usuario o lógica de negocio sin construir la capacidad subyacente. Características: baja diferenciación respecto a otros wrappers que usan los mismos proveedores subyacentes, alta dependencia del proveedor, costo de cambio bajo. Una app que llama a la API de OpenAI para responder preguntas es Wrapper en el eslabón Modelo.

Plataforma (nivel 2). El actor opera y gestiona capacidad propia sobre componentes Core de terceros. Agrega orquestación, SLAs y control operacional. Diferenciación moderada: el cliente paga por las capacidades operacionales que la Plataforma agrega, no por la capacidad subyacente que sigue siendo de terceros. Azure OpenAI es Plataforma en Modelo: opera modelos de OpenAI con SLAs y gobernanza enterprise, pero los modelos son del proveedor original.

Core (nivel 3). El actor construye la capacidad fundacional del eslabón con tecnología propia: modelos, motores o algoritmos diferenciados. Foso competitivo alto basado en propiedad intelectual. OpenAI es Core en Modelo: construye sus propios modelos. Anthropic, Google con Gemini, Meta con Llama, todos son Core en Modelo. La distinción entre Core y los niveles superiores es donde está la mayor parte del valor capturado en el campo de IA.

Infraestructura (nivel 4). El actor provee el sustrato computacional, de almacenamiento o conectividad sobre el cual operan los niveles superiores. Foso muy alto basado en escala y capital. NVIDIA es Infraestructura en Modelo: las GPUs que NVIDIA fabrica son el sustrato sobre el cual los modelos operan. AWS, GCP, Azure son Infraestructura en muchos eslabones — proveen el cómputo y almacenamiento subyacentes a casi toda la industria.

La progresión Wrapper → Plataforma → Core → Infraestructura es de **control creciente** sobre el eslabón. Wrapper consume; Plataforma opera; Core construye; Infraestructura sustenta. Cada nivel

de profundidad típicamente implica mayor inversión, mayor especialización técnica, mayor foso competitivo. También implica mayor riesgo: un Core que apostó por una tecnología que el mercado descartó queda con activo difícil de reposicionar; un Wrapper que apuesta mal cambia de proveedor en horas.

Cobertura × Profundidad — el espacio de posicionamiento

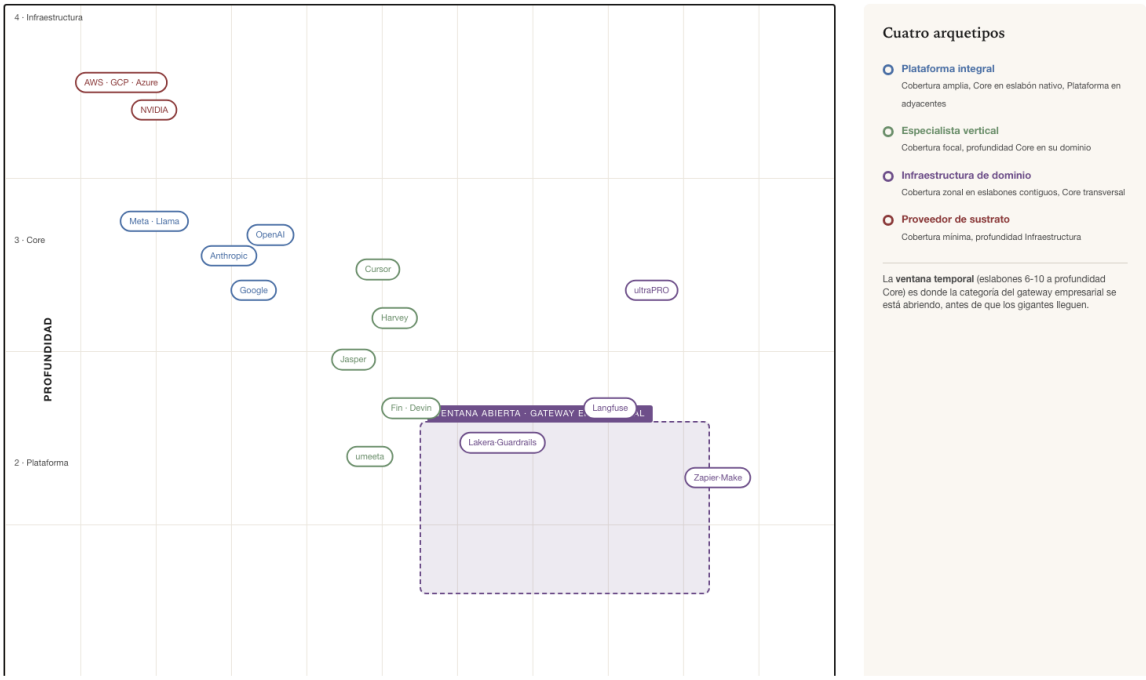


Figura 35: Espacio bidimensional · cobertura × profundidad

La combinación de cobertura (eslabones) y profundidad (niveles) produce un **espacio bidimensional** donde se posiciona cualquier actor. El eje horizontal muestra **cuántos eslabones** abarca un actor; el eje vertical muestra **a qué profundidad** participa en cada uno.

Un mismo actor puede operar a distintas profundidades en distintos eslabones. OpenAI es Core en Modelo pero Plataforma en Acceso (sus APIs) y Plataforma en Agentes (ChatGPT). Esta heterogeneidad por eslabón es la regla, no la excepción. Pocos actores tienen profundidad uniforme a través de todos los eslabones que tocan — y cuando la tienen, típicamente son actores muy enfocados como NVIDIA en Infraestructura computacional.

La diversidad de posiciones en el espacio bidimensional permite identificar arquetipos de posicionamiento que se repiten en el mercado, con propiedades estratégicas distintas. La sección siguiente desarrolla los cuatro arquetipos canónicos.

Arquetipos estratégicos emergentes

De este espacio bidimensional emergen **cuatro arquetipos** recurrentes. Cada arquetipo describe un patrón de posicionamiento con propiedades estratégicas características. Los cuatro arquetipos son:

Figura 36: Los cuatro arquetipos estratégicos

Plataforma integral, Especialista vertical, Infraestructura de dominio, Proveedor de sustrato.

Plataforma integral

El arquetipo de **Plataforma integral** combina cobertura amplia (tres o más eslabones) con profundidad **Core** en su eslabón nativo y profundidad **Plataforma** en eslabones adyacentes. Es el arquetipo de los grandes laboratorios de IA que dominan el campo en 2026.

OpenAI ejemplifica el arquetipo: Core en Modelo (construye GPT), Plataforma en Acceso (vende API), Plataforma en Agentes (opera ChatGPT y Operator), Plataforma en Especializaciones emergentes (los GPTs verticalizados). Anthropic sigue patrón similar pero con énfasis distinto: Core en Modelo (construye Claude), Core en Acceso vía MCP (su contribución abierta a Herramientas), Plataforma en Agentes. Google con Gemini hace análogo. Meta con Llama es caso particular: Core en Modelo distribuyendo open source, sin plataforma propia de Acceso o Agentes — su foso es la distribución del modelo, no la operación.

El foso competitivo de la Plataforma integral es la propiedad intelectual del Core combinada con la integración vertical hacia eslabones adyacentes. Un Wrapper que llama a OpenAI no puede replicar fácilmente lo que OpenAI hace en su totalidad — para hacerlo, tendría que entrenar su propio modelo (eslabón Modelo a profundidad Core), construir su propia infraestructura de Acceso, operar su propia plataforma de Agentes. Esa cadena completa requiere capital y talento que pocos actores tienen.

Especialista vertical

El arquetipo de **Especialista vertical** combina cobertura focal (uno o dos eslabones) con profundidad **Core**. Es el arquetipo de los actores que se han concentrado en dominios específicos y construyen profundidad allí.

Cursor ejemplifica el arquetipo en coding: Core en Especializaciones para programación. Harvey AI hace lo mismo en legal. Jasper en marketing. Fin (de Intercom) en customer support. Devin pretende Core en Especializaciones de coding autónomo. umeeta opera el mismo arquetipo en consultoría profesional, con Core en la capa de memoria del engagement. Cada uno tiene cobertura estrecha — uno o pocos eslabones — pero profundidad Core en su vertical específica.

El foso competitivo del Especialista vertical es la profundidad de saber-hacer vertical, que se materializa típicamente como Capabilities densas — el saber profesional codificado del que hablamos en el Capítulo 5. Un GPT genérico puede responder preguntas sobre legal, pero Harvey AI las responde con mucho mejor calidad porque tiene Capabilities de Legal construidas con disciplina. La diferencia no es marketing — es estructural. Un competidor que quisiera replicar Harvey tendría que construir el árbol de Capabilities de Legal con el mismo rigor, lo que toma años.

Infraestructura de dominio

El arquetipo de **Infraestructura de dominio** es el más reciente en la industria y el menos poblado. Combina cobertura zonal (dos o más eslabones contiguos) con profundidad **Core** transversal a un dominio funcional, con posibles extensiones a eslabones no contiguos a menor profundidad.

Un actor que es Core en Runtime, Firewall, Observabilidad, Herramientas e Integraciones — los eslabones 6, 7, 8, 9, 10 — con extensión Plataforma en Acceso constituye el caso paradigmático del arquetipo. La combinación de cobertura zonal en cinco eslabones contiguos con profundidad Core constituye un **gateway empresarial**: la capa fundacional para conectar y controlar IA en producción.

El foso competitivo de la Infraestructura de dominio es la integración profunda entre eslabones que otros actores tratan separadamente. Construir Core en Runtime es mérito; construir Core simultáneamente en Runtime, Firewall, Observabilidad, Herramientas e Integraciones, **integrados coherentemente**, es propiedad arquitectónica que pocos actores tienen. La razón estructural es que estos cinco eslabones operan juntos en producción — sin uno, los otros pierden valor — y construir solo uno deja al actor dependiente de complementarios que típicamente no existen como producto integrado.

Proveedor de sustrato

El arquetipo de **Proveedor de sustrato** combina cobertura mínima (un eslabón) con profundidad **Infraestructura**. Es el arquetipo de los actores que sustentan la industria desde la capa más profunda.

NVIDIA ejemplifica el arquetipo en Modelo: las GPUs que NVIDIA fabrica son el sustrato computacional sobre el cual operan los modelos. AWS, GCP y Azure son Proveedores de sustrato en múltiples eslabones — Datos, Modelo, Cómputo en general. Cisco lo es en redes para IA distribuida.

El foso competitivo del Proveedor de sustrato es escala, capital intensivo, y efectos de red en hardware o data center. Construir una empresa que compita con NVIDIA en GPUs requiere inversiones de billones de dólares y generaciones de I+D acumuladas. Construir una empresa que compita con AWS en cómputo a escala requiere infraestructura física global. Estos fosos son los más altos del campo, pero también son los que requieren mayor capital inicial y tienen ciclos de retorno más largos.

Mapeo de actores principales

A modo de ejemplo, la tabla siguiente clasifica familias de productos representativas del mercado actual según los eslabones que abarcan y la profundidad en cada uno. Las cifras son **niveles de profundidad** (1-4); los paréntesis indican framework o meta-herramienta (para construir, no para usar).

Actor	Da	Mo	Ac	Ag	Xp	Ru	Fi	Ob	He	In
Datos y anotación										
Scale AI / Labelbox	3									
Hugging Face	3	2								
Plataformas integrales										
OpenAI (ChatGPT, GPT API)		3	2	2	2	3			3	
Anthropic (Claude, MCP)		3	3	2					3	
Google (Gemini)		3	2	2					3	
Meta (Llama)		3								
Perplexity			2	3	3					
DeepSeek / Qwen / Ernie		3	3	3						
Especializaciones verticales										
GitHub Copilot				2	3					
Cursor / Replit					3					
Devin					3	3			3	
Harvey / Jasper / Fin					3					
umeeta (memoria engagement)					3					
Sustratos agentivos										
Agentia (régimen privado)				3		3				
Soveria (régimen público)				3		3				
Frameworks y herramientas										
LangChain / Graph				(3)		(3)			(3)	
AutoGPT / CrewAI				(3)		(3)				
Pinecone / Weaviate									(3)	
Operaciones y governance										
Guardrails / NeMo / Lakera							3			
Langfuse / LangSmith / W&B								3		
Zapier / Make / n8n										3
ultraPRO (gateway empres.)			2			3	3	3	3	3
Infraestructura computacional										
NVIDIA		4								
AWS / GCP / Azure	4	4								

Leyenda de eslabones: **Da** Datos · **Mo** Modelo · **Ac** Acceso · **Ag** Agentes · **Xp** Especializaciones · **Ru** Runtime · **Fi** Firewall · **Ob** Observabilidad · **He** Herramientas · **In** Integraciones. Niveles de profundidad: **1** Wrapper · **2** Plataforma · **3** Core · **4** Infraestructura. Los paréntesis — p. ej. (3) — indican framework o meta-herramienta (para construir, no para usar).

> **Nota.** La tabla cubre los eslabones 1 a 10. El eslabón 11 (Entorno) se omite por ser externo a la cadena — es el territorio sobre el cual actúan los eslabones anteriores, no un eslabón que un actor de IA ocupe a una profundidad. Sus implicaciones se desarrollan en la sección de mundo de carbono.

La tabla, leída en conjunto, permite ver patrones que la inspección individual de cada producto no revela. Las plataformas integrales tienden a concentrarse en los eslabones 2-4. Los especialistas verticales se acumulan en el eslabón 5. Los productos de operaciones y governance se distribuyen entre los eslabones 7-10. La infraestructura computacional ocupa principalmente el eslabón 2 a profundidad 4.

Lecturas estratégicas del mapa

El mapa no es solo descriptivo — es herramienta para análisis estratégico. Tres lecturas del mapa de 2026 permiten entender el estado del campo y dónde están las oportunidades.

La concentración por arquetipo

Los **especialistas verticales** dominan el eslabón 5 (Especializaciones). Cursor, Harvey, Jasper, Fin, Devin: cada uno construyó Capabilities verticales densas y captura mercado en su dominio. La concentración es saludable — múltiples actores con poco solapamiento, cada uno dueño de su vertical. Es donde la innovación está más vibrante en 2026.

Las **plataformas integrales** concentran los eslabones 2-4 (Modelo, Acceso, Agentes). Cinco actores dominantes globales — OpenAI, Anthropic, Google, Meta, DeepSeek/Qwen/Ernie — y posiciones derivadas. La concentración es alta y crece con el tiempo, porque construir Core en Modelo requiere capital y talento que pocos actores pueden sostener. Es el eslabón con mayor barrera de entrada.

La **infraestructura** está concentrada en NVIDIA en cómputo y los hyperscalers en datos y cómputo a escala. Foso de capital extremo. La concentración aquí es estructural y probablemente persistente — es razonable esperar que no aparezca un entrante significativo en estos eslabones a profundidad Infraestructura en el horizonte previsible.

Los espacios menos disputados

Hay zonas del mapa donde la profundidad Core está abierta y donde un actor con disciplina puede construir posición competitiva sin enfrentar incumbents masivos.

El **eslabón 1 (Datos) a profundidad Core**: pocos actores Core (Scale AI, Labelbox); el resto son commodity. Hay espacio para actores que construyan capacidad propia en datos especializados.

Los **eslabones 6-10 simultáneamente** — Runtime, Firewall, Observabilidad, Herramientas, Integraciones — a profundidad Core con cobertura zonal: territorio de la Infraestructura de dominio. Actores que combinan estos cinco eslabones a profundidad Core son raros. Es el espacio donde la categoría del **gateway empresarial completo** se está abriendo.

El **eslabón 11 (Entorno)** con conexión a IoT y mundo físico: prácticamente vacío en términos de actores específicamente diseñados para el Mundo Agentivo. Es la frontera de la próxima generación, y la sección sobre mundo de carbono desarrolla las implicaciones.

La trayectoria de los gigantes

Los gigantes — OpenAI, Anthropic, Google, Microsoft — avanzan **eslabón a eslabón**: del Modelo al Acceso, del Acceso a los Agentes, de los Agentes a las Especializaciones. La progresión es histórica observable. OpenAI nació como laboratorio de Modelo, expandió a Acceso (API), expandió a Agentes (ChatGPT), está expandiendo a Especializaciones (GPTs).

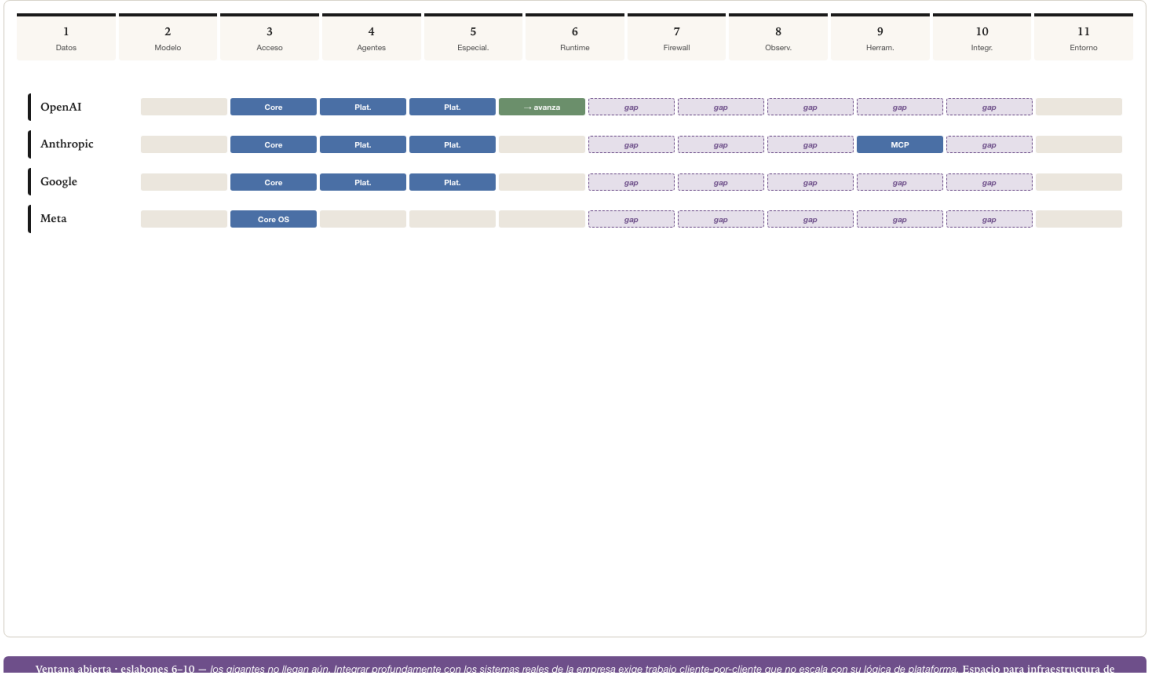


Figura 37: La trayectoria de los gigantes · y la ventana abierta

Pero llegar al eslabón Integraciones — donde el agente toca los sistemas reales de la empresa — exige un trabajo integrador de empresa por empresa que no escala con la lógica de estos actores. OpenAI puede ofrecer ChatGPT Enterprise con conectores a Slack y Salesforce, pero integrar profundamente con el ERP de cada cliente, con su CRM particular, con su data warehouse legacy — eso no es trabajo de plataforma, es trabajo de integración. **Esto crea una ventana temporal** para que actores especializados en eslabones 6-10 (infraestructura de dominio) construyan posición antes de que los gigantes lleguen. La ventana no es indefinida — los gigantes eventualmente llegan a integraciones, posiblemente vía adquisición — pero existe ahora y ofrece oportunidad estratégica para quien la entiende.

El gateway empresarial como categoría

La combinación **Core en Runtime + Firewall + Observabilidad + Herramientas + Integraciones**, con extensión **Plataforma en Acceso**, define una categoría arquitectónica con función única: **conectar y controlar simultáneamente** la operación de agentes empresariales. Es la materialización formal de la Capa 4 de la Arquitectura Agentiva sobre los eslabones de mercado.

A esta categoría se le llama **gateway empresarial de IA** (figura arriba).

Conectar sin **controlar** es Zapier — capacidad de integración sin gobernanza. **Controlar** sin **conectar** es Lakera — capacidad de seguridad sin integración. La combinación de ambos en un único punto arquitectónico es categoría reciente y aún poco poblada. Los actores que la ocupen primero capturan el espacio antes de que los gigantes lleguen.

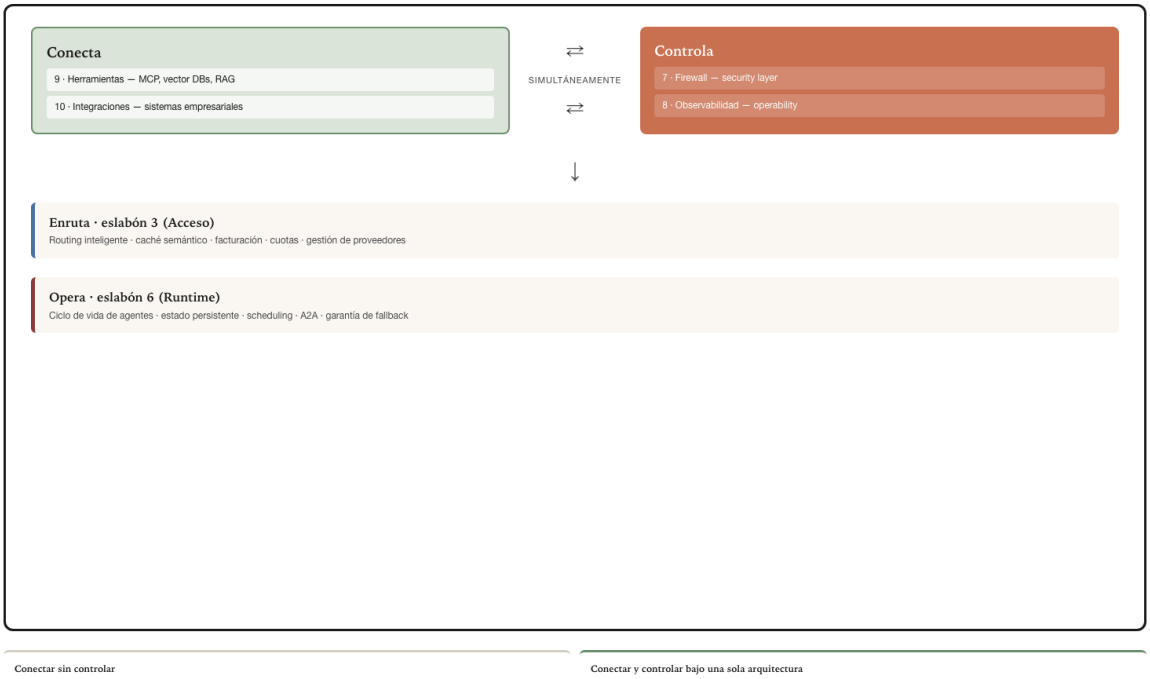


Figura 38: El gateway empresarial de IA · conectar y controlar

Análisis competitivo del gateway empresarial

Cuando se evalúan actores que pretenden ocupar el gateway empresarial, una rúbrica útil compara nueve capacidades a través de los actores principales del campo.

Capacidad	Portkey	Lasso	Lakera	Langfuse	Credo AI	Noma	Gateway empresarial completo
Routing LLM	✓						✓
Caché semántico	✓						✓
Prompt security		✓	✓			✓	✓
Tokenización		✓				✓	✓
DLP		✓	✓			✓	✓
Políticas / CRUDLEX	△	△			✓	△	✓
Aprobación humana					△		✓
Validación respuestas			△	△	△	△	✓
Observabilidad	△			✓	△	△	✓

Capacidad	Portkey	Lasso	Lakera	Langfuse	Credo AI	Noma	Gateway empresarial completo
Conectividad empresarial (Herram. + Integrac.)							✓

La rúbrica documenta, columna por columna, la tesis ya enunciada en la sección anterior: cada actor cubre algunas capacidades — Portkey cubre routing y caché, Lasso cubre prompt security y DLP, Langfuse cubre observabilidad —, pero ninguno integra conectividad empresarial profunda y control bajo una misma arquitectura. La última fila — **gateway empresarial completo** — es la única con cobertura íntegra de la rúbrica.

Esta es la oportunidad estratégica más clara que el mapa revela. El gateway empresarial completo es categoría que apenas está emergiendo, con espacio para actores que la construyan con disciplina arquitectónica. ultraPRO es uno de los actores que se posiciona en esta categoría, integrando los cinco eslabones de control y conectividad bajo el patrón tripartito Cloud + Cliente + Local que el Capítulo 8 desarrolla en detalle.

Implicaciones para constructores

El mapa de la cadena de valor tiene tres lecturas operativas para quien construye en el campo de IA.

La primera: **no competir en eslabón equivocado**. Si una empresa pequeña intenta ser Core en eslabón Modelo, compite contra OpenAI, Anthropic, Google, Meta — laboratorios con respaldos de billones. La derrota es estructural. Si la misma empresa busca Infraestructura de dominio en eslabones 6-10, compite en categoría poco disputada con foso construible. La asimetría es real y favorable. Elegir bien el eslabón es probablemente la decisión estratégica más importante para una empresa que entra al campo.

La segunda: **la profundidad Core requiere disciplina**. Llegar a profundidad Core en cualquier eslabón exige construir capacidad técnica profunda — no integración de terceros. La diferencia entre Wrapper, Plataforma y Core no es opinable: se mide por dependencias estructurales del producto. Un Wrapper deja de operar si el proveedor lo apaga; un Core opera independiente. Si tu producto deja de funcionar cuando OpenAI cambia su pricing, eres Wrapper. Si tu producto sigue funcionando, eres Core.

La tercera: **la cobertura amplia exige integración**. Una empresa que pretende cubrir múltiples eslabones a profundidad Core (infraestructura de dominio) debe resolver el problema de integración interna entre esos eslabones. Operar Runtime + Firewall + Observabilidad + Herramientas + Integraciones como cinco productos separados produce una empresa con cinco productos. Operarlos como una arquitectura coherente produce un gateway. La diferencia es lo que el cliente percibe como valor.

La monetización cruza la línea

Hay una lectura del mapa que no es de posicionamiento sino de **modelo de negocio**, y conviene hacerla explícita porque es la consecuencia económica más profunda de la tesis del libro: el modelo que

financió al software empresarial durante dos décadas — la licencia o suscripción SaaS, cobrada porque los usuarios *usan* la aplicación — no sobrevive intacto al cruce de la Línea Nadella. Si los empleados nunca abren Power BI, ¿cómo se cobran licencias de Power BI? Si ningún usuario ve jamás la interfaz de Salesforce, ¿cómo demuestra Salesforce el valor de su CRM? La tensión alcanza al propio autor de la frase: Microsoft vende exactamente las interfaces cuyo colapso su CEO predice — y su reconversión acelerada a plataforma de agentes es la respuesta, no la negación.

Del lado agentivo de la línea, la monetización migra hacia cuatro modelos emergentes:

1. **Por capacidad, no por licencia** — no "cien asientos de Salesforce" sino acceso de agentes a la capacidad de gestión de relaciones con clientes; el asiento desaparece junto con la interfaz que lo justificaba.
2. **Por uso real, no por acceso potencial** — no suscripción fija sino cargo por análisis ejecutado, transacción procesada, conversación resuelta; el consumo agentivo es medible por diseño (el append-only log ya lo registra).
3. **Por valor creado, no por herramienta provista** — participación en la eficiencia ganada o en el ingreso atribuible a los insights de los agentes; el modelo más difícil de instrumentar y el mejor alineado.
4. **Por capacidades críticas que los agentes necesitan** — gobernanza, auditoría, permisos, identidad: servicios cuyo valor no depende de que nadie "abra" nada. No es casual que este libro dedique dos capítulos a esa categoría: la Trust Infrastructure no es solo la condición técnica de la producción — es de los pocos lugares donde el modelo de negocio del software queda *más* claro después del cruce, no menos.

La lectura estratégica: en el lado agéntico el valor se cobra en la interfaz; en el agentivo, en la capacidad, el consumo y la confianza. Los actores del mapa que ya monetizan sin depender de que un humano abra su producto — infraestructura, firewall, observabilidad — cruzan la línea con el modelo intacto; los que monetizan asientos frente a pantallas tienen pendiente, además del reposicionamiento técnico, un reposicionamiento de caja.

Discoverability agentiva — el desplazamiento de la capa de descubrimiento

El modelo de cadena de valor describe dónde se *produce* el valor. Pero hay una propiedad estructural del Mundo Agentivo que el modelo de cadena por sí solo no captura: **dónde se descubre** el valor producido. La capa de descubrimiento del software empresarial cambió, y la cadena no opera bien si el lector no entiende ese cambio.

En el mundo de aplicaciones, el descubrimiento ocurría en buscadores: Google para servicios web, las app stores para móvil, los marketplaces verticales para SaaS. El humano que necesitaba una capacidad la encontraba escribiendo una búsqueda, y los actores invertían en posicionamiento — SEO, ASO, contenido, ads — para ser encontrados. La capa era el buscador, y los buscadores eran un puñado.

En el Mundo Agentivo, el humano que necesita una capacidad no abre Google — le pregunta al asistente que ya tiene abierto. *"¿Dónde puedo publicar este agente?"*, *"¿Qué herramienta me sirve para esta tarea?"*, *"¿Hay un AgencyDomain que cubra este dominio?"*. La respuesta no viene del índice de la web — viene del modelo entrenado. La capa de descubrimiento se desplazó del *índice de búsqueda* al *modelo de cognición*.

Esto tiene tres consecuencias estructurales para cualquier actor que construya en la cadena de valor agentiva.

La primera consecuencia es que **la presencia entrenada importa más que el ranking**. Un actor que

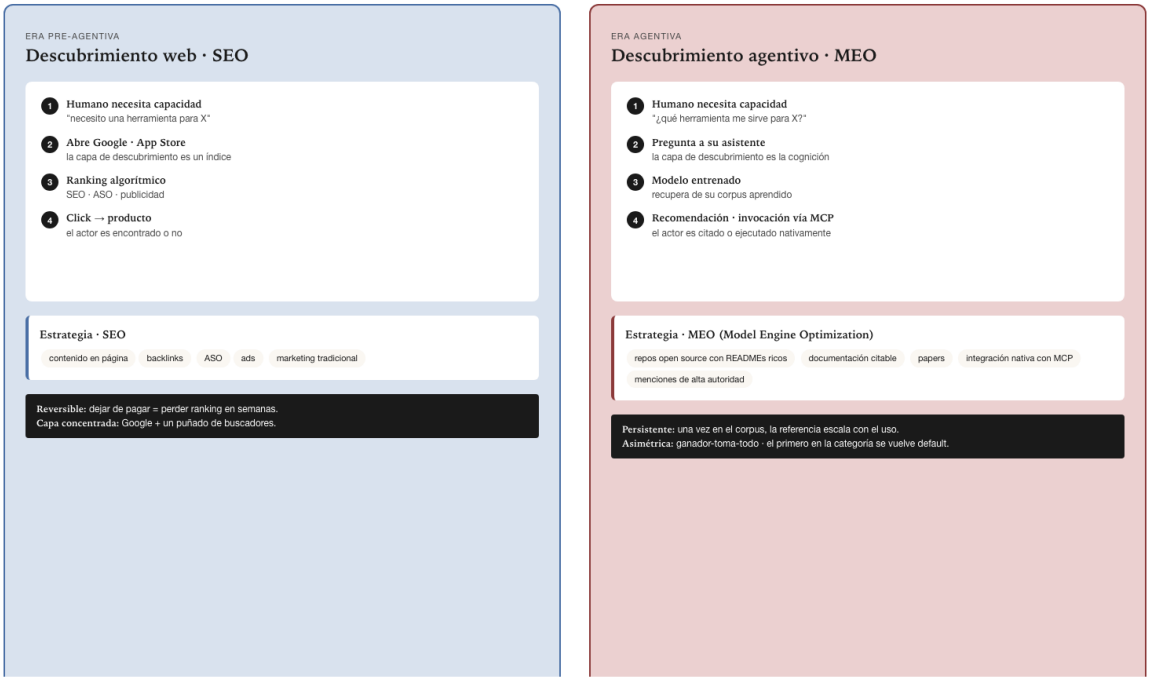


Figura 39: Del SEO al MEO · descubrimiento web vs. descubrimiento agentivo

no aparece en el corpus de entrenamiento de los modelos frontera es invisible, independientemente de su SEO o su marketing tradicional. El equivalente conceptual del SEO en este nuevo mundo es lo que la industria empieza a llamar **MEO — Model Engine Optimization**: el conjunto de prácticas que aseguran que los modelos frontera (Claude, GPT, Gemini, Llama, los que vengan) tengan al actor en su conocimiento entrenado y operativo. Se construye con presencia pública estructurada — repositorios open source con READMEs ricos en casos de uso, documentación abundante con ejemplos citables, papers, integración nativa con la spec MCP, mentions en blogs y foros técnicos relevantes.

La segunda consecuencia es que **la dinámica es persistente y asimétrica**. Una vez que un modelo frontera "aprende" a un actor de la cadena, la referencia escala con el uso del modelo. No depende de pagar por click ni de mantener inversión continua en posicionamiento. Es ventaja acumulativa que sobrevive a ciclos de marketing, y tiene tendencia a ganador-toma-todo: si un actor es el primero que los modelos frontera referencian sistemáticamente para una categoría, los siguientes pelean contra esa default. Quien construya hoy con visión MEO captura ventaja temporal que se hace progresivamente difícil de revertir.

La tercera consecuencia es que **la integración con MCP es vector de difusión**. Cuando un actor publica capacidades como tools del Model Context Protocol que un modelo puede invocar nativamente, ese modelo no solo *menciona* al actor — lo *ejecuta*. La invocación reiterada construye familiaridad estructural del modelo con el actor, distinta de la familiaridad citacional que produce documentación pública. La spec MCP es entonces, además de protocolo de integración técnica, vector de presencia en los modelos frontera.

Para el arquitecto y para el inversionista, la conclusión operativa es que **la cadena de valor de IA**

opera sobre una capa de descubrimiento agentiva, y esa capa tiene reglas distintas a las del descubrimiento web. Quien construya productos agentivos debe pensar la presencia entrenada como categoría de inversión propia — no como sub-problema del marketing tradicional.

El modelo de cadena de valor es el mapa. Las dos secciones siguientes del Capítulo 6 profundizan en eslabones específicos donde quien construya o invierta encontrará lecturas distintas pero igualmente operativas: una sobre un eslabón ya maduro que separa a los actores serios de los que improvisan, y otra sobre el eslabón menos desarrollado y más prometedor del campo, donde la próxima década del valor económico se va a definir.

Deep-dive · Observabilidad (eslabón 8)

Hay un patrón observable en cualquier industria que madura: la primera generación de productos vende capacidad — *"esto puede hacer X"* — y la segunda generación vende observabilidad — *"esto puede hacer X y tú puedes saber qué está haciendo cuando lo hace"*. El campo de IA agentiva está atravesando exactamente esa transición. Los primeros tres años de la categoría se vendieron en torno a capacidades crecientes; los próximos años se venderán en torno a operabilidad, y la operabilidad se sostiene sobre Observabilidad.

Esta sección desarrolla el eslabón 8 — Observabilidad — con el detalle que el eslabón merece. Es uno de los eslabones más rápidos en consolidarse dentro de la cadena de valor de IA, y probablemente el primero en alcanzar madurez de mercado donde múltiples actores Core compiten activamente. Para el arquitecto, la Observabilidad es el eslabón sin el cual los sistemas agentivos en producción son cajas negras inoperables. Para el inversionista, es el eslabón donde la próxima generación de productos enterprise va a definir su categoría.

¿Por qué Observabilidad merece deep-dive?

Observabilidad es uno de los **eslabones más rápidos en consolidarse** dentro de la cadena de valor de IA. La razón es operativa: una organización que opera agentes en producción — sea uno o cien — necesita saber, en tiempo real, qué hacen, cuánto cuestan, qué tan bien funcionan y cuándo fallan. Sin esa visibilidad, opera a ciegas, y operar a ciegas un sistema que toma decisiones autónomas es indefendible regulatoria y comercialmente.

A diferencia de eslabones más tempranos como Modelo o Acceso, donde el mercado está concentrado en pocos actores dominantes, Observabilidad es **mercado fragmentado y joven**: múltiples actores Core con cobertura parcial, espacio para diferenciación por dominio o por integración con eslabones adyacentes. La fragmentación no es debilidad — es síntoma de un mercado donde distintos actores eligen distintos énfasis de las múltiples capacidades que la Observabilidad cubre, y los compradores combinan productos según sus prioridades específicas.

> *Sin Observabilidad, los agentes son cajas negras. Con Observabilidad, son sistemas operables.*

La cita anterior resume bien el rol del eslabón. Un sistema agentivo sin observabilidad puede funcionar técnicamente, pero la organización no puede operarlo: no puede diagnosticar fallos, no puede optimizar costos, no puede defender sus decisiones, no puede mejorar su performance. La Observabilidad transforma capacidad técnica en operabilidad enterprise.

Definición canónica

Observabilidad (eslabón 8) es la capa que **observa, mide y retroalimenta** sobre el comportamiento de un sistema de IA en producción. Su función arquitectónica es proveer el **ciclo de feedback operacional** que permite a la organización mantener confiabilidad, costo y calidad bajo control.

Hay que delimitar la Observabilidad respecto a eslabones cercanos que la industria suele confundir con ella. Tres distinciones precisas. **Observabilidad no es seguridad**: la seguridad — eslabón 7, Firewall — protege; la Observabilidad describe. Son funciones distintas que cooperan en la práctica pero responden a problemas distintos. Una organización puede tener Firewall sin Observabilidad o Observabilidad sin Firewall, aunque la mayoría madura tiene los dos. **Observabilidad no es herramientas**: las herramientas — eslabón 9 — son lo que el agente usa para tocar el mundo; la Observabilidad observa qué herramientas usa, cuándo, y con qué resultado. **Observabilidad no es validación**: la validación — parte de Trust Infrastructure, pilar 3 — verifica que la respuesta sea correcta antes de emitirla; la Observabilidad registra qué se emitió y permite reconstruir después.

Pregunta que responde	Eslabón
¿Es seguro?	Firewall (7)
¿Cómo funciona?	**Observabilidad (8)**
¿Qué puede hacer?	Herramientas (9)

Las seis capacidades canónicas

Una implementación completa de Observabilidad para sistemas agentivos cubre seis capacidades. Las seis son distintas, atacan problemas operacionales distintos, y los productos del mercado típicamente cubren algunas más profundamente que otras. Las desplegamos con el detalle que cada una merece.

La primera capacidad es **tracing**. Es trazabilidad end-to-end de cada operación del agente. Permite reconstruir, a posteriori, qué pasó: qué solicitud entró, qué Capabilities se aplicaron, qué tools se invocaron, en qué orden, con qué latencia, qué resultado se generó. El tracing exige instrumentación explícita — un agente bien instrumentado emite eventos estructurados en cada paso significativo (decisión cognitiva, invocación de tool, generación de respuesta, escalación al humano). Esos eventos se correlacionan por trace ID que sigue al request a través del sistema, permitiendo armar la historia completa de una operación. Productos como Langfuse, LangSmith, Helicone, Arize AI hacen tracing como capacidad central.

La segunda capacidad es **monitoreo de costos**. Desglose en tiempo real del consumo de tokens y otros recursos pagos: por modelo, por usuario, por proyecto, por tool. La economía operativa de un sistema agentivo depende críticamente de esta visibilidad — un agente puede ser técnicamente correcto y económicamente inviable si se invoca cognición costosa cuando un Botlet bastaría. El monitoreo de costos en sistemas maduros es **predicción, no solo registro**: las plataformas avanzadas proyectan el gasto del mes en base al patrón de uso de los días corridos, alertan cuando el ritmo va camino de un overrun, permiten configurar cuotas que detienen el sistema cuando se alcanzan. Helicone y Langfuse destacan especialmente en costos. Portkey integra monitoreo de costo con routing inteligente, dirigiendo cada solicitud al modelo más eficiente según parámetros configurables.

La tercera capacidad es **evaluación de calidad**. Verificación sistemática de que las respuestas del agente cumplen estándares de calidad. Tiene dos sub-modos canónicos. La **evaluación automatizada** — eval as service — corre regularmente un dataset de prueba contra el modelo en producción,

midiendo precisión, completitud, formato. Detecta degradación: si el modelo o las Capabilities cambian y la calidad baja, la evaluación lo detecta antes de que el cliente lo note. La **evaluación humana** complementa con revisión muestral de respuestas reales por humanos calificados — detecta problemas que las métricas automáticas no capturan: tono inapropiado, sutileza perdida, juicio profesional dudoso. Productos como Braintrust, Patronus AI, Weights & Biases destacan en eval. La industria converge en frameworks como LangChain Evaluators y OpenAI Evals como marcos comunes que múltiples productos de eval pueden compartir.

La cuarta capacidad son las **métricas de rendimiento** — las métricas operativas clásicas adaptadas al sistema agentivo. La **latencia** mide el tiempo desde la solicitud hasta la respuesta, distinguiendo percentiles (p50, p95, p99). Un p99 alto puede degradar la experiencia incluso con p50 bueno — y en sistemas agentivos en producción, el p99 importa porque es donde los outliers de cognición costosa o tools lentos se materializan. El **throughput** mide solicitudes manejadas por unidad de tiempo, crítico para sistemas multi-tenant que operan a escala. La **disponibilidad** mide uptime del sistema agentivo, distinguiendo disponibilidad del agente, del modelo subyacente, y de tools downstream. La **tasa de éxito** mide proporción de solicitudes completadas correctamente — y en sistemas con Botlets, debe distinguir éxito por Botlet versus por agente versus por sistema, porque cada nivel falla por razones distintas.

La quinta capacidad es **debugging y reproducibilidad**. La capacidad de **replay de invocaciones** — re-ejecutar una operación pasada exactamente como ocurrió, para diagnosticar fallos. Exige guardar el contexto completo: prompt, modelo, parámetros, datos consultados, tools invocados, resultado. El debugging agentivo es estructuralmente más complejo que el debugging tradicional por tres razones. Primera, los modelos LLM son **probabilísticos** — la misma entrada puede producir salidas distintas, lo que dificulta reproducir exactamente un fallo. Segunda, los agentes pueden tener **estado persistente** — el contexto cambia entre invocaciones, y reproducir un fallo requiere reproducir también el estado. Tercera, los Botlets se **regeneran** — la versión que falló puede ya no existir cuando se intenta reproducir, porque el agente la regeneró cuando el ambiente cambió. LangSmith y Langfuse productizan replay como capacidad central, con mecanismos para preservar estado y versiones.

La sexta capacidad son las **alertas y anomalías**. Detección proactiva de comportamientos fuera de patrón: latencia o costo que se dispara, tasa de éxito que cae bajo umbral, cambios en distribución de tipos de solicitudes, Botlets que regeneran con frecuencia inusual, tools que fallan con mayor frecuencia. Las alertas no solo notifican: pueden disparar acciones automáticas — circuit breakers que detienen el sistema cuando las condiciones se deterioran, rollback a versión anterior del agente cuando una nueva degrada calidad, escalación al humano cuando los umbrales se aproximan a violación.

Productos representativos del eslabón

El mercado de Observabilidad agentiva tiene ya varios actores Core compitiendo activamente. Los principales actores Core incluyen los productos siguientes, con sus diferenciadores:

Producto	Cobertura principal	Diferenciador
Langfuse	Las 6 capacidades, fuerte en tracing y costos	Open source, despliegue self-hosted
LangSmith	Tracing, evaluación, debugging	Integración nativa con LangChain
Helicone	Tracing, costos, observabilidad de proxy	Drop-in proxy para OpenAI/Anthropic

Producto	Cobertura principal	Diferenciador
Arize AI	Eval, monitoreo de drift	Foco en ML clásico extendido a LLMs
Braintrust	Eval automatizado y humano	Workflow de eval como CI/CD
Patronus AI	Eval especializado en hallucination y safety	Categorías propias de evaluación
Weights & Biases	Tracking de experimentos, evaluación	Madurez del producto en ML clásico

La fragmentación es intencional: distintos actores eligen distintas capacidades como diferenciador. Una organización madura **combina dos o tres productos** según su mezcla de necesidades, en lugar de buscar una solución monolítica. Esta combinación es lo que el mercado llama "stack de observabilidad" — análogo al stack de monitoreo tradicional con Datadog para metrics, Splunk para logs, PagerDuty para alertas. Cada eslabón de capacidad lo opera el producto que mejor lo ataca; la integración entre productos es responsabilidad del operador.

Diferenciación con eslabones adyacentes

Precisamos las diferencias con eslabones adyacentes con tablas comparativas que dejan clara la separación funcional.

Frente a Firewall (eslabón 7), las dos categorías operan en momentos distintos del ciclo de la acción del agente. El Firewall opera **antes** de la ejecución; la Observabilidad opera **durante y después**. El Firewall actúa **bloqueando** lo que considera prohibido; la Observabilidad actúa **registrando y describiendo**. El foco del Firewall es prevención; el foco de la Observabilidad es diagnóstico. Un sistema bien diseñado integra ambos: el Firewall bloquea lo prohibido en tiempo real; la Observabilidad registra qué se bloqueó para detectar patrones y mejorar las políticas. Pero son funciones distintas con productos distintos — confundirlos lleva a soluciones que cubren mal ambas.

Frente a Herramientas (eslabón 9), la diferencia es entre **capacidad activa** y **capacidad descriptiva**. Las herramientas extienden lo que el agente puede hacer — es capacidad activa. La Observabilidad observa cómo el agente usa las herramientas — es capacidad descriptiva. Sin Observabilidad, las herramientas son opacas: el desarrollador puede saber qué tools tiene el agente registrados, pero no cómo las usa en la práctica, qué tools ejecuta más, cuáles fallan más, qué patrones de uso emergen.

Frente a Trust Infrastructure (transversal), Observabilidad **es uno de los componentes** de Trust Infrastructure, específicamente del pilar 5 (Transparencia). Pero Trust Infrastructure es **superior en alcance**: incluye también gobernanza, auditoría, validación y resiliencia. La Observabilidad es necesaria pero no suficiente para Trust Infrastructure completa. Una organización que tiene Observabilidad excelente pero no tiene los otros pilares operacionalizados sigue sin estar lista para producción enterprise.

La trayectoria del eslabón

Tres tendencias visibles en el mercado de Observabilidad agentiva a inicios de 2026 anticipan hacia dónde va el campo.

La primera tendencia es la **convergencia de eval y tracing**. Productos que partieron como eval puro (Braintrust, Patronus) están agregando tracing. Productos que partieron como tracing (LangSmith,

Helicone) están agregando eval. El mercado converge hacia **observabilidad agentiva integral** que cubre las seis capacidades — pero con productos especializados con foco distinto. Es probable que en dos a tres años emerjan productos que cubren las seis capacidades a profundidad respetable, compitiendo con productos especializados que cubren una o dos capacidades a profundidad excelente.

La segunda tendencia es la **competencia entre open source y SaaS**. Langfuse fue pionero del modelo open-source self-hosted, contra el modelo SaaS dominante. La adopción enterprise de Langfuse — especialmente en sectores regulados que no pueden enviar datos sensibles a SaaS externo — sugiere que el modelo open-source tiene espacio sostenible. La pregunta de los próximos años es si el modelo open-source captura el segmento enterprise regulado mientras el modelo SaaS captura el resto, o si la dinámica converge hacia un solo modelo dominante.

La tercera tendencia es la **integración con stacks de desarrollo**. LangSmith integra con LangChain. Braintrust integra con frameworks de eval populares. La tendencia es que el desarrollador no cambie de IDE para ver observabilidad — se vuelve parte natural del workflow de desarrollo de agentes. Esta integración es probablemente la diferenciación competitiva sostenible — los productos que se integren mejor con los stacks de desarrollo populares capturan adopción que los que se quedan como herramientas standalone no logran.

Implicaciones para constructores

Para una organización que opera agentes en producción, tres lecciones operativas emergen del análisis del eslabón.

La primera: **Observabilidad no es opcional**. Las seis capacidades son **mínimo viable**. Operar agentes en producción sin tracing, sin monitoreo de costos, sin eval, sin métricas, sin replay, sin alertas, es operar a ciegas. Las plataformas maduras saben esto; los pilotos exploratorios típicamente no lo aprenden hasta que un incidente lo expone. El primer incidente serio es típicamente cuando la organización descubre que necesita Observabilidad de verdad — y lamenta no haberla diseñado desde el inicio.

La segunda: **la integración importa más que el producto individual**. Combinar dos o tres productos especializados — por ejemplo Langfuse para tracing y costos, Braintrust para eval, alertas en una herramienta corporativa propia — **suele ser superior** a usar un único producto que cubre todo a profundidad media. La integración exige trabajo, pero la profundidad por capacidad lo compensa. Las organizaciones que intentan minimizar el número de proveedores típicamente terminan con observabilidad superficial; las que aceptan la complejidad de stack y la integran bien terminan con observabilidad profunda.

La tercera: **la instrumentación se diseña al inicio**. Agregar observabilidad a un sistema agentivo construido sin instrumentación es trabajo costoso y produce visibilidad incompleta. Los sistemas que se diseñan con observabilidad en mente desde el inicio — emitiendo eventos estructurados en cada paso significativo — terminan con observabilidad mucho más útil que los que la agregan después. La inversión inicial en instrumentación se paga muchas veces durante la operación.

> *La observabilidad no se compra. Se diseña.*

Mundo de carbono · eslabón 11

Los primeros diez eslabones de la cadena de valor de IA cubren la operación de agentes en el mundo digital: datos digitales, modelos digitales, agentes que invocan tools digitales sobre sistemas digitales.

Esta cobertura es necesaria, pero **no es suficiente** para responder a la mayor parte del valor económico que existe en el mundo. La mayor parte del PIB global no se genera en software — se genera en industrias que producen materia: manufactura, energía, agricultura, transporte, salud, construcción. Estas industrias del mundo físico son el territorio que la próxima generación del campo agentivo debe alcanzar, y son precisamente lo que el eslabón 11 — Entorno — captura.

Esta sección desarrolla el eslabón menos desarrollado de la cadena de valor de IA y, simultáneamente, el más importante para la próxima década. Es donde la Arquitectura Agentiva enfrenta su próxima generación de problemas, y donde las organizaciones que construyan correctamente capturarán una posición competitiva difícil de igualar.

La frontera del eslabón 11

El eslabón 11 — **Entorno** — extiende la cadena al territorio donde la operación digital se encuentra con el mundo físico. La extensión no es trivial. El mundo digital opera con bits que se replican sin costo, transacciones que se revierten con relativa facilidad, validación posterior por revisión de logs. El mundo físico opera con materia que tiene masa, procesos que tienen consecuencias irreversibles, validación que ocurre en sensores físicos que pueden fallar. Operar agentes en este territorio requiere consideraciones que el mundo digital no enfrentaba.

A este territorio expandido lo llamamos **mundo de carbono** — la materia, los procesos físicos, las máquinas, los seres vivos. La frontera entre el mundo de silicio (digital) y el mundo de carbono (físico) es donde la Arquitectura Agentiva enfrenta su próxima generación de problemas. La metáfora — silicio versus carbono — captura algo importante: la diferencia no es solo de medio, es de naturaleza. El silicio se manipula con bits; el carbono se manipula con materia. Las arquitecturas que funcionaron para el primero necesitan adaptación profunda para el segundo.

> *Los agentes no terminan de servir cuando llegan al borde del mundo digital. Empiezan a servir de verdad cuando lo cruzan.*

¿Por qué importa esta extensión?

Tres razones operativas concretas justifican atender al eslabón 11 con prioridad estratégica, no como nota al pie del campo digital.

La primera razón es que **la mayor parte del valor económico vive en el mundo de carbono**. Las industrias que mueven el PIB global — manufactura, energía, agricultura, transporte, salud, construcción — son industrias del mundo de carbono. Sus procesos productivos no son flujos de información: son flujos de materia y energía con sensores y controladores que los gobiernan. La industria de IA contemporánea, predominantemente concentrada en aplicaciones de mundo digital — chatbots, asistentes, generación de contenido, herramientas de software —, está cubriendo la **periferia del valor económico**. El cruce al mundo de carbono es donde están los problemas con impacto mayor y donde los volúmenes de negocio que el campo puede capturar son órdenes de magnitud mayores.

La segunda razón es que **las industrias del mundo de carbono ya generan datos masivos**. Una planta de manufactura moderna genera terabytes diarios de datos de sensores. Una flota de camiones conectados, lo mismo. Un campo agrícola con riego inteligente y drones de monitoreo, lo mismo. Una red de telecomunicaciones, lo mismo. **El dato existe**. Lo que falta no es la materia prima — es la capa arquitectónica que convierte ese dato en operación autónoma gobernada. La cadena de valor de IA, desarrollada hasta el eslabón 10, conecta. El eslabón 11 es donde el agente actúa sobre el mundo que los datos describen.

La tercera razón es que **el cruce abre categorías técnicas nuevas**. Operar agentes que tocan el mundo físico exige resolver problemas que los agentes puramente digitales no enfrentan. La **latencia con consecuencias físicas** es una: un agente que demora treinta segundos en decidir abrir una válvula puede ser inaceptable; el proceso físico no espera, y un retraso puede significar pérdida de producto, daño a equipo, o riesgo de seguridad. La **reversibilidad limitada** es otra: en el mundo digital, la mayoría de las acciones son reversibles (con suficiente trabajo); en el mundo físico, una vez que se cortó una pieza, se aplicó un fertilizante o se inyectó un medicamento, no hay rollback. La **validación con sensores físicos** es la tercera: la validación de Capa 4 no se hace solo contra schemas de datos — se hace contra mediciones físicas (temperatura, presión, peso, posición), y el agente debe leer el mundo, no solo APIs, con todas las complicaciones de sensores que pueden fallar, tener deriva o devolver lecturas atípicas. La **resiliencia ante fallo de hardware** es la cuarta: un sensor malo, un actuador atascado, una red intermitente — son condiciones cotidianas en el mundo físico, y la Trust Infrastructure debe contemplarlos como caso normal, no como excepción.

Sub-categorías del eslabón 11

Figura 40: Mundo de carbono · cuatro sub-categorías por madurez × regulación

El Entorno no es uniforme. Para fines de análisis, distinguimos cuatro sub-categorías con propiedades distintas. Cada una tiene su madurez de mercado actual, sus actores líderes, sus desafíos específicos.

La primera sub-categoría son los **sistemas empresariales tradicionales**: ERPs (SAP, Oracle, Microsoft Dynamics), CRMs (Salesforce, HubSpot), DBMS (Oracle, SQL Server, PostgreSQL), sistemas de RRHH, sistemas financieros legacy. Son **digitales pero institucionales** — el agente los toca vía APIs, pero las APIs reflejan modelos de datos que llevan décadas evolucionando con su propia lógica. Es la sub-categoría más madura del eslabón 11. La industria de **integración empresarial** — Zapier,

Make, n8n, Workato, MuleSoft — es Core en este sub-eslabón pero opera mayormente en modelo agentic, no autónomo. Los productos contemporáneos integran sistemas pero no operan agentivamente sobre ellos; eso es la próxima generación.

La segunda sub-categoría es el **mundo físico industrial** — manufactura y energía. Sistemas SCADA, MES (Manufacturing Execution Systems), PLCs (Programmable Logic Controllers), sensores industriales, robots de línea, válvulas, bombas, hornos, turbinas, redes eléctricas. Son **digitales pero conectados a hardware** — cada API termina, eventualmente, en un equipo físico que actúa sobre materia. Es la sub-categoría más prometedora en términos de valor económico capturable, y simultáneamente la más conservadora. Los procesos industriales operan bajo regulaciones estrictas — seguridad de planta, calidad de producto, certificaciones de equipos — que no admiten experimentación libre. El agente debe demostrar Trust Infrastructure ejemplar antes de tener autorización para tocar sistemas críticos.

La tercera sub-categoría es el **mundo físico móvil** — transporte, logística, agricultura. Vehículos conectados, flotas, drones, equipos agrícolas con sensores. Son **digitales, conectados a hardware, y móviles** — agregan al desafío anterior la conectividad intermitente y la coordinación geográfica distribuida. Es la sub-categoría con adopción más rápida. Las flotas de logística (Amazon, FedEx, DHL) ya operan con agentes autónomos en routing y despacho. La agricultura de precisión avanza rápido en países desarrollados.

La cuarta sub-categoría es el **mundo biológico**. Datos genómicos, monitoreo médico continuo, registros médicos electrónicos, sistemas de farmacovigilancia, seguimiento epidemiológico. Son datos del mundo de carbono biológico, con regulaciones extremadamente estrictas — HIPAA, GDPR salud, regulaciones farmacéuticas. Es la sub-categoría con mayor potencial de impacto humano y, simultáneamente, mayor exigencia de Trust Infrastructure. Un agente que interpreta resultados médicos opera en territorio donde un error tiene consecuencia humana directa.

Patrones de integración para el mundo de carbono

Operar agentes en el eslabón 11 exige patrones arquitectónicos específicos que no son comunes en el mundo digital puro. La adaptación de la Arquitectura Agentiva a este territorio merece ser explícita.

El primer patrón es **edge computing como Capa 3 distribuida**. Los agentes que tocan procesos industriales no pueden depender de cognición remota — la latencia y la conectividad intermitente lo prohíben. Si un agente que controla una válvula necesita esperar el round-trip a un servidor cloud antes de cada decisión, el sistema no es viable. La Capa 3 — Autonomía — se distribuye al **edge**: gateways industriales, controladores de borde, dispositivos on-premises que mantienen agentes operando localmente con sincronización eventual al centro. Los Botlets son particularmente útiles aquí: ejecutan localmente sin requerir cognición remota; la cognición se invoca solo cuando un cambio de ambiente lo exige y la conectividad está disponible.

El segundo patrón es **twin digital del mundo físico**. Una práctica creciente: los agentes operan no sobre el mundo físico directamente, sino sobre un **gemelo digital** (digital twin) que refleja el estado del sistema físico en tiempo real. El gemelo digital actúa como abstracción de la Capa 4 — el agente lo consulta y lo modifica; el gemelo propaga al mundo físico cuando es seguro hacerlo. El gemelo digital permite **validación previa**: el agente puede simular en el gemelo el efecto de una decisión antes de aplicarla al mundo real. Esto es crítico cuando la reversibilidad es limitada — si la simulación muestra que la decisión produce resultado problemático, el agente puede ajustar antes de afectar al mundo físico.

El tercer patrón son **múltiples niveles de aprobación**. A diferencia del mundo digital donde

la mayoría de las decisiones del agente se ejecutan autónomamente, en el mundo de carbono la aprobación humana es habitual para acciones de alto impacto. La Trust Infrastructure debe modelar capas de aprobación: el agente decide, un operador local aprueba, un supervisor remoto verifica. Cada capa de aprobación agrega latencia pero también agrega seguridad — y en el mundo físico, donde las consecuencias son irreversibles, la latencia adicional se justifica para las decisiones de alto impacto.

El cuarto patrón es **sensores como tools de Capa 4**. Una particularidad del mundo de carbono: los sensores son tools de Capa 4. El agente "consulta" un sensor de temperatura del mismo modo que consulta una API. La diferencia es que el sensor no devuelve datos sintéticos — devuelve mediciones del mundo físico, con todo el ruido, deriva, y posibles fallos que eso implica. La validación de Capa 4 debe contemplar **calidad de la medición**: detectar sensores con valores atípicos, sensores que dejaron de actualizar, sensores cuya calibración derivó. Esto es distinto de la validación de respuestas de APIs digitales, donde el dato está bien o mal por reglas claras; el sensor puede estar técnicamente operando pero devolver lecturas que no reflejan correctamente la realidad.

Estado actual del mercado

A inicios de 2026, el mercado del eslabón 11 para agentes está fragmentado y joven. La madurez varía significativamente entre las cuatro sub-categorías, y los actores líderes son típicamente actores de cada vertical específica que están agregando capacidad agentiva, no actores de IA agentiva que están entrando a la vertical.

Sub-categoría	Madurez del mercado	Actores representativos
Sistemas empresariales tradicionales	Alta (industria de integración madura)	Salesforce · SAP · Oracle · Workato · Zapier
Mundo industrial — manufactura/energía	Baja-media (pilotos en curso, escalamiento parcial)	Siemens (Mindsphere) · GE (Predix) · PTC (ThingWorx) · Aveva
Mundo móvil — transporte/logística	Media (operadores grandes con capacidades propias)	Amazon Logistics · Tesla · Deere · proveedores de fleet management
Mundo biológico	Baja (alto potencial, alta regulación)	Tempus · Flatiron · Veeva — actores especializados por sub-vertical

El patrón visible: en sub-categorías altamente reguladas, el mercado es de actores especializados por vertical. En sub-categorías menos reguladas, hay espacio para infraestructura horizontal aún no construida.

La oportunidad de infraestructura agentiva para el mundo de carbono

La cadena de valor de IA actual cubre los eslabones 1-10 con creciente madurez en cada uno. El eslabón 11 sigue siendo, en su mayor parte, territorio de actores legados que no nacieron diseñados para integrarse con agentes autónomos. Siemens, GE, PTC, Aveva — los actores tradicionales del mundo industrial — tienen el conocimiento del dominio pero no la disciplina arquitectónica agentiva. Sus plataformas operan principalmente en modelo de monitoreo y control humano-supervisado, no en modelo agentivo donde los agentes ejecutan autónomamente.

Esto crea una **oportunidad arquitectónica**: una infraestructura agentiva que se especifique como gateway hacia el mundo de carbono, ofreciendo tools normalizadas para conectarse a sistemas SCADA/MES/PLCs, patrones de edge computing pre-construidos, twins digitales como abstracción nativa, Trust Infrastructure ajustada a las regulaciones de cada sub-vertical, y modelos de aprobación humana multi-nivel.

Esta infraestructura no es producto contemporáneo de ningún actor del mercado de IA digital. Su construcción exige conocimiento profundo del mundo de carbono — vocabulario industrial, regulaciones, prácticas operativas — combinado con la disciplina arquitectónica de la cadena de valor de IA. Los actores que lo logren primero se quedan con un territorio que los gigantes tardarán años en pisar.

> *El gateway empresarial de IA conecta la cognición con sistemas digitales. El gateway empresarial extendido al mundo de carbono conecta la cognición con la materia.*

La frontera de evolución

El eslabón 11 es la frontera más visible de evolución de la Arquitectura Agentiva. Tres trabajos abiertos que la comunidad técnica deberá resolver para que el cruce sea masivo.

El primer trabajo abierto son los **estándares de tools para mundo industrial**. MCP (Model Context Protocol) provee un estándar para tools de mundo digital. **No existe equivalente** maduro para tools de mundo industrial. Los protocolos existentes — OPC UA, MQTT, Modbus — son de la era pre-agentiva: el agente puede consumirlos pero no son diseñados para él. El trabajo abierto es construir una **MCP para industria** que defina cómo un agente descubre, autentica y opera sensores y actuadores industriales con la misma uniformidad con que opera APIs digitales hoy.

El segundo trabajo abierto es la **Trust Infrastructure especializada por vertical**. Las regulaciones del mundo de carbono — seguridad funcional (IEC 61508, ISO 26262), seguridad de proceso (IEC 61511), salud (HIPAA, FDA), aviación (DO-178C) — exigen requisitos específicos que la Trust Infrastructure genérica no cubre por completo. El trabajo abierto es construir extensiones verticales de Trust Infrastructure que codifiquen los requisitos regulatorios de cada vertical, certificables formalmente. Una Trust Infrastructure con certificación IEC 61508 puede ser usada en plantas industriales reguladas; sin esa certificación, el sistema agentivo simplemente no puede operar en esos ambientes.

El tercer trabajo abierto es el **aprendizaje de modelo en el mundo de carbono**. Los modelos de fundación se entrenaron mayormente en datos digitales — texto, imágenes, código. Su comprensión del mundo físico es indirecta — leen documentación, no operan equipos. La frontera técnica de la Capa 2 (Cognición) es entrenar modelos que comprenden el mundo de carbono **directamente**: a partir de datos de sensores, simulaciones físicas, video industrial, datos biomédicos. El trabajo abierto es construir modelos multimodales que integran datos del mundo de carbono como modalidad nativa, no como traducción a texto.

Implicaciones estratégicas

Para quienes construyen sobre la Arquitectura Agentiva, tres lecciones operativas importan.

La primera: **el mundo de carbono no es horizonte lejano para todas las industrias**. Para industrias que ya operan en el mundo de carbono — manufactura, energía, telco, salud, agricultura, logística —, el eslabón 11 es el eslabón inmediato de su realidad. No es opción posponerlo: cada decisión de stack agentivo que tomen tiene que contemplarlo desde el inicio. Una organización industrial que adopta IA

agentiva sin considerar cómo va a tocar sus PLCs y SCADAs construye sistema que servirá para tareas de oficina pero no para operación productiva.

La segunda: **el gap de infraestructura es ventaja temporal**. Para los actores que construyan gateway empresarial extendido al mundo de carbono ahora, hay ventana de varios años antes de que los actores del mundo digital lleguen. Los actores actuales del mundo industrial tienen el conocimiento del dominio pero no la disciplina arquitectónica agentiva. Los actores del mundo digital tienen lo opuesto. Quien combina ambos primero, define la categoría. Esa ventana se cierra eventualmente — los gigantes adquieren capacidad o construyen — pero existe ahora.

La tercera: **Trust Infrastructure es el filtro**. En el mundo de carbono, el filtro para entrar al mercado **no es el agente más capaz** — es el agente con Trust Infrastructure certificable. Un agente brillante que no puede demostrar conformidad con las regulaciones de seguridad funcional simplemente no puede operar en una planta. Esto invierte la prioridad típica del mundo digital, donde la capacidad domina sobre la conformidad. En el mundo de carbono, la conformidad es prerrequisito; la capacidad es diferenciador secundario una vez cumplido el prerrequisito.

Lo que sigue

Cerrado el modelo de mercado, el libro entra en su tramo más concreto. Quien busque un caso donde la Arquitectura Agentiva entregue valor demostrable hoy encontrará en el siguiente capítulo el desarrollo de una aplicación canónica fundacional. Quien necesite el detalle operativo para construir lo que hasta aquí se describió como principio encontrará en el Capítulo 8 la traducción a artefactos accionables.

Esta página se dejó intencionalmente en blanco.

Capítulo 7 · Conocimiento en tiempo real

Hay un problema que casi cualquier ejecutivo serio de una empresa mediana o grande reconoce de inmediato cuando se le describe: la **lentitud insufrible** del acceso al conocimiento sobre su propio negocio. El ejecutivo tiene una intuición sobre algo que está pasando — los márgenes parecen estar cayendo, el churn parece estar subiendo, una región parece estar comportándose distinto de las otras —, formula la pregunta a su área de BI, y la respuesta llega entre dos y ocho semanas después. Para entonces, la decisión que motivó la pregunta ya pasó. La organización quedó operando con la intuición sin verificar.

Este capítulo trata sobre el primer caso de uso que demuestra, con métricas claras, el valor de cruzar la Línea Nadella. Es el caso donde la diferencia entre el Mundo Agéntico y el Mundo Agentivo deja de ser abstracción y se vuelve experimentable: una pregunta que tomaba semanas en producir respuesta, en el Mundo Agentivo toma segundos. La transformación no es cuantitativa solamente — es **cualitativa**: cuando preguntar es gratis, las organizaciones descubren que pueden preguntar cosas que antes no se hacían, y descubren que las preguntas no formuladas contenían los insights más valiosos.

Una arquitectura formal queda incompleta sin un caso canónico que la demuestre. Para la Arquitectura Agentiva, ese caso es el **acceso a información analítica** — el problema más cotidiano y mejor entendido de la operación empresarial moderna. Tres razones hacen al caso fundacional para el libro.

La primera razón es que **es universal**. Toda organización que opera con datos enfrenta el problema de convertir datos en decisiones. La BI conversacional aplica a finanzas, operaciones, ventas, marketing, recursos humanos — sin distinción. No importa el sector, no importa el tamaño, no importa la geografía: el problema existe en todas partes y tiene forma similar.

La segunda razón es que **tiene métrica clara**. El valor se mide en tiempo: cuánto tarda una pregunta de negocio en convertirse en respuesta accionable. La diferencia entre semanas y segundos no requiere argumento — es directamente perceptible. Cuando un ejecutivo experimenta por primera vez una pregunta nueva respondida en segundos en lugar de semanas, la diferencia se vuelve preferencia adquirida sin necesidad de venta.

La tercera razón es que **construye sobre arquitectura existente**. La metodología Kimball — el estándar canónico de data warehousing desde 1996 — sigue siendo válida. Lo que cambia es **el propósito** del modelo: pasa de ser contenedor de reportes a ser **activo estratégico** sobre el cual los agentes razonan. Esta propiedad — que no exige descartar la inversión existente sino capitalizarla — es lo que hace al caso adoptable. Una organización con data warehouse maduro no necesita reescribir su infraestructura; necesita agregar la capa agentiva por encima.

> *El valor agentivo no se demuestra reemplazando lo que ya funciona. Se demuestra haciendo que lo que ya funciona produzca resultados a velocidades que antes eran imposibles.*

El problema histórico

El acceso a conocimiento analítico ha sido históricamente lento porque cada pregunta nueva requiere un proyecto. La secuencia es conocida y dolorosa, y captura exactamente el problema que la BI conversacional resuelve.

El ejecutivo formula la pregunta a su área de BI. La pregunta puede ser tan simple como *"¿por qué cayeron los márgenes en el segmento corporativo el mes pasado?"* o tan compleja como *"¿qué clientes tienen patrones de uso similar a los que cancelaron en los últimos seis meses?"*. En ambos casos, la pregunta tiene que ser interpretada por humanos del área de BI, traducida a especificación técnica, asignada a un desarrollador, construida como reporte o dashboard, validada con el ejecutivo, ajustada según feedback, y finalmente entregada.

La secuencia tiene cuatro fases típicas: **coordinación** (días para programar reuniones y alinear expectativas), **levantamiento** (días para entender la pregunta y especificar los datos requeridos), **desarrollo** (semanas para construir el reporte), **validación** (días para revisar y ajustar). El proceso completo toma típicamente entre cuatro y doce semanas. Y aún así, lo que se entrega es un reporte específico que responde la pregunta original — no es capacidad reutilizable para preguntas relacionadas que el ejecutivo se hará después.

El cuello de botella real es el que el diagnóstico del Capítulo 2 nombró: humanos en el medio, cada uno agregando latencia y margen de malinterpretación. El ejecutivo formula la pregunta de manera ambigua porque está pensándola en voz alta; el analista interpreta la ambigüedad de un modo; el desarrollador implementa otra interpretación. Cuando el reporte llega, el ejecutivo descubre que no es exactamente lo que necesitaba, y el ciclo recomienza.

El costo de poner el modelo tradicional en operación tampoco es trivial. Construir la cadena Data Lake → Synapse → Power BI, o cualquier equivalente moderno, requiere típicamente entre cien mil y quinientos mil dólares de setup inicial, entre diez mil y cincuenta mil dólares mensuales de operación, y un tiempo a primer dashboard útil de entre tres y seis meses. Y toda esa inversión entrega capacidad de responder solo aquellas preguntas que alguien previó al diseñar el sistema. Para la pregunta no anticipada — la intuición nueva del lunes — vale el diagnóstico del Capítulo 2: fuera del menú, a la cola o al olvido.

La industria de BI consolidada reconoce abiertamente que el modelo llegó a su techo — el Capítulo 2 citó las voces (Tellius, Superwise, Cube, Tableau, BCG) y no las repetiremos. La sustancia del reconocimiento importa más que sus citas: quince años de rediseños cosméticos — dashboards más bonitos, self-service para usuarios no-técnicos, NLP sobre datos pre-modelados — prometieron democratizar el acceso al conocimiento y entregaron mejoras incrementales, porque el problema nunca fue cosmético sino estructural. El cuello de botella era el humano en el medio, y ningún rediseño podía eliminarlo sin reemplazarlo con algo equivalentemente capaz.

La solución agentiva

Con la Arquitectura Agentiva implementada sobre los datos de la organización, el ejecutivo conversa **directamente con un agente** que tiene acceso a esos datos. La secuencia que en el modelo tradicional tomaba cuatro a doce semanas se colapsa en una operación de segundos. La pregunta llega al agente, el agente la procesa, ejecuta dinámicamente la consulta sobre los datos, devuelve la respuesta. Si el ejecutivo quiere refinar — *"y ahora muéstrame solo el segmento corporativo"* —, el agente refina en la siguiente respuesta. Si quiere profundizar — *"¿qué cambió específicamente en septiembre?"* —, el agente profundiza. La conversación reemplaza al proyecto.

Figura 41: El colapso del costo de una pregunta · semanas a segundos

La diferencia es estructural. Los pasos intermedios — coordinación, levantamiento, desarrollo, validación — desaparecen porque el agente los ejecuta dinámicamente en cada pregunta. El humano que tenía que ser intermediario sale del medio. Y como el agente puede responder cualquier pregunta, no solo las pre-construidas, el ejecutivo no necesita "pedir un nuevo reporte" cada vez que tiene una intuición nueva. La capacidad analítica está disponible para cualquier pregunta sobre los datos disponibles.

El cambio cuantitativo es radical. La métrica de tiempo a respuesta pasa de cuatro a doce semanas en el modelo tradicional, a cinco a sesenta segundos en el modelo agentivo. La diferencia es de tres órdenes de magnitud — no es mejora, es transformación.

Cuando el costo de una pregunta colapsa de semanas a segundos, **cambia la naturaleza** de la relación entre la organización y su información. Los tres efectos que el Capítulo 2 ya describió se materializan en el caso del conocimiento en tiempo real. La capacidad analítica se vuelve elástica — se adapta a la necesidad actual, no a lo que alguien pre-definió hace meses. La iteración reemplaza a la especificación — el ejecutivo explora, refina, profundiza en una conversación continua con la información, en lugar de definir requerimientos por adelantado y esperar el resultado. Y las preguntas que nunca se hacían, ahora se hacen — cuando preguntar es gratis, la curiosidad analítica deja de estar limitada por el presupuesto de BI.

> *El cambio más importante no es la velocidad. Es la libertad cognitiva del ejecutivo, que deja de tener que elegir qué preguntar.*

El tiempo real como temporalidad continua

Conviene precisar qué es exactamente el "tiempo real" de este caso — el estado al otro lado del Salto Cuántico que el Capítulo 2 acuñó. No es un canal de entrega ni un atributo de la pregunta: es la **temporalidad** del Botlet que sostiene la manifestación informativa. Aplicado a la BI, un **reporte** (snapshot punto-en-tiempo) es el Botlet con temporalidad discreta, y un **dashboard vivo** es el mismo Producto de Información (`PI`) con temporalidad continua, sostenido sobre un runtime persistente de Capa 3. El "tiempo real" del caso BI no se elige marcando un modo de entrega: se elige dándole al Botlet temporalidad continua. La especificación de temporalidad — los dos regímenes y el único runtime que los unifica — vive en el Capítulo 5 §2.

El `PI` que esta aplicación canónica entrega puede, además, componerse de **múltiples vistas nombradas con navegación por contexto** (drill-through). Su descripción normada — multi-vista, la propiedad data-anchored / no-bypass — vive en el Capítulo 5 §2; aquí basta señalar que esa composición es ortogonal a la temporalidad: aplica igual a un snapshot que a un dashboard vivo.

Hay que ser preciso sobre qué cambia y qué no. El modelo Kimball **no desaparece**. El agente lo ejecuta dinámicamente — no lo reemplaza. La metodología, los conceptos (hechos, dimensiones, conformed dimensions, slowly changing dimensions), las prácticas profesionales del data warehousing siguen siendo el sustrato. Los data warehouses **no desaparecen**. Siguen siendo donde los datos están almacenados, modelados y gobernados. Lo que cambia es **quién los consume**: deja de ser solo el dashboard humano; pasa a ser también el agente. Los analistas y CIOs de datos **no desaparecen**. Su trabajo se desplaza: pasan de construir dashboards específicos a diseñar la capa semántica sobre la cual los agentes razonan correctamente. Es trabajo más estratégico, menos operativo.

La arquitectura subyacente — Kimball Barnizada

La metodología Kimball, formulada por Ralph Kimball en *The Data Warehouse Toolkit* (1996), sigue siendo el estándar vendor-neutral para modelado dimensional. Su validez no caduca — su propósito evoluciona. A esta evolución la llamamos **Kimball Barnizada**: preserva la estructura conceptual de Kimball — Source, ETL, Presentation, BI Apps, más Metadata transversal — y le agrega un acabado contemporáneo que refleja el cambio de propósito del modelo de datos.

Los componentes canónicos de Kimball clásico son cinco. **Source** son los sistemas operacionales: ERPs, CRMs, sistemas transaccionales que generan los datos en su forma cruda. **ETL** es el proceso de extract-transform-load: limpieza, conformidad, transformación de los datos desde su forma operacional a su forma analítica. **Presentation** son los data marts: hechos y dimensiones, conformed dimensions que permiten consistencia inter-marts. **BI Apps** son los dashboards, reportes, herramientas de exploración interactiva — la superficie con la cual el humano consulta los datos. **Metadata** atraviesa todo: gobernanza, calidad, lineage.

La arquitectura clásica funciona para la era pre-agentiva. Sirve para que humanos consulten datos vía dashboards. No sirve, por sí sola, para que agentes consulten datos para razonar autónomamente. La razón no es que el modelo Kimball esté mal — es que está incompleto para el caso agentivo. Le falta la capa que traduce el modelo en algo que el agente entiende.

Sobre la estructura clásica de Kimball se monta una **capa contemporánea** que sirve a los agentes. La primera componente es la **capa semántica explícita**: un knowledge graph o capa semántica equivalente que codifica el significado de las dimensiones, las relaciones, las reglas de negocio. La capa semántica es lo que el agente consulta antes de generar consultas SQL — sin ella, el agente alucina.

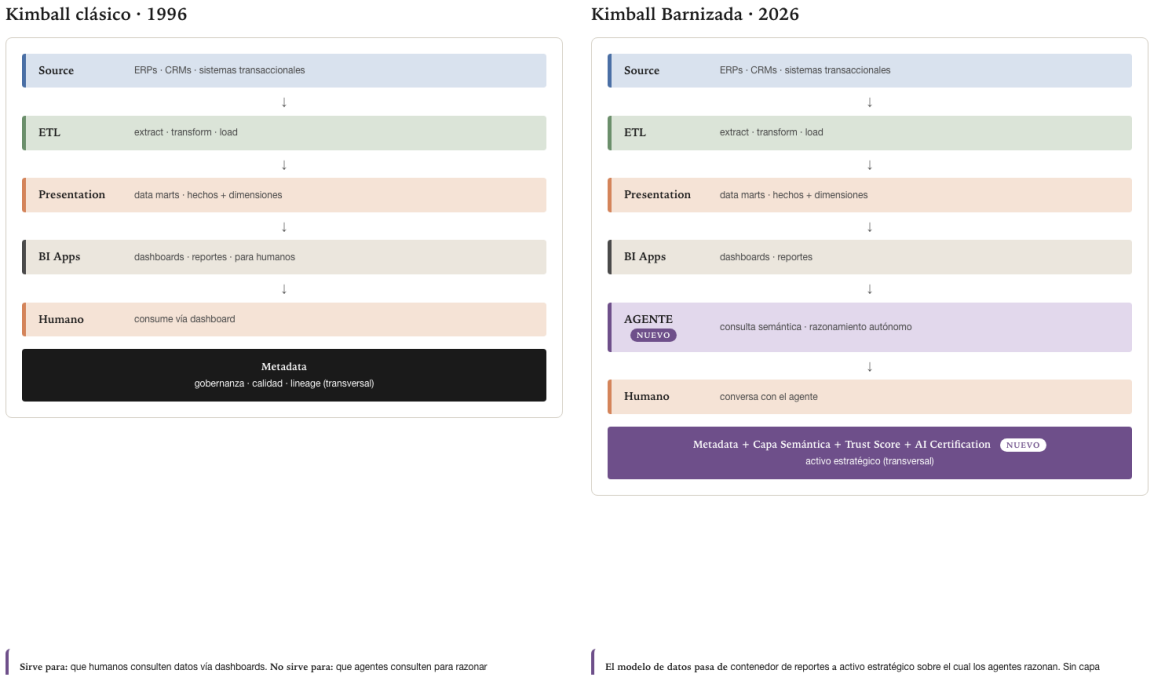


Figura 42: Kimball clásico vs. Kimball Barnizada

La segunda componente es el **Trust Score por dato**: métrica de confiabilidad de cada dato basada en linaje, calidad, governance y observabilidad. Componente del modelo de datos como activo estratégico. La tercera componente es la **AI Certification**: proceso automatizado de verificación de madurez y readiness de los modelos analíticos para ser consumidos por agentes. Aborda requisitos de NIST AI RMF, ISO/IEC 42001, EU AI Act. La cuarta componente es la **observabilidad de queries agentivos**: monitoreo de las consultas que los agentes ejecutan: cuáles, con qué frecuencia, con qué éxito. Permite identificar gaps en la capa semántica.

La estructura completa de Kimball Barnizada es la que sintetiza la figura anterior.

El cambio más importante de la arquitectura: el modelo de datos pasa de ser **contenedor de reportes** a ser **activo estratégico**. Esto significa que la calidad del dato deja de medirse por su limpieza y pasa a medirse por su accionabilidad por agentes — un dato bien limpiado pero sin contexto semántico es inservible para un agente, mientras que un dato algo sucio pero rico en contexto puede ser utilísimo. Significa que la gobernanza pasa de ad hoc a certificada. Significa que el modelado se dirige por las preguntas que los agentes harán, no por los reportes esperados.

La medición de AtScale que el Capítulo 2 citó — más del **ochenta por ciento** de fallo sin capa semántica; precisión cercana al **cien por cien** con ella — es el sustento cuantitativo de esta primera componente. Y los términos que la industria acuñó para la capa — el **Agentic Semantic Layer** de ThoughtSpot, el **Enterprise Knowledge Graph** de Salesforce, el **Agentic BI** de Databricks — son ángulos del mismo diagnóstico: el agente necesita mucho más que datos; necesita significado asociado a los datos.

Mapeo a los hyperscalers

FUNCIÓN KIMBALL	MICROSOFT AZURE	AWS	GOOGLE CLOUD	DATABRICKS
Source	conectores AZURE DATA FACTORY	Glue · Kinesis INGESTA	Dataflow · Pub/Sub INGESTA	Bronze MEDALLION: CRUDO
ETL · Warehouse	Synapse Analytics DW + ETL	Redshift DW	BigQuery DW SERVERLESS	Silver MEDALLION: LIMPIO
Presentation · BI Apps	Power BI DASHBOARDS	QuickSight DASHBOARDS	Looker · Looker Studio DASHBOARDS + LOOKML	Gold + Dashboards PRESENTACIÓN
Gobernanza · Metadata	Purview CATÁLOGO + LINEAGE	DataZone · Lake Formation CATÁLOGO + PERMISOS	Dataplex CATÁLOGO + LINEAGE	Unity Catalog CATÁLOGO + PERMISOS
Capa agentiva	parcial FABRIC EN CONSTRUCCIÓN	parcial BEDROCK CONECTORES AD HOC	parcial VERTEX AI INTEGRACIONES	parcial AGENTIC BI EMERGENTE

La equivalencia funcional es directa hasta la fila de gobernanza. *Pero ninguno de los hyperscalers tiene aún la capa agentiva completa: capa semántica explícita + Trust Score + AI Certification + observabilidad de queries agentivos integrados bajo una arquitectura coherente.* **Esa brecha es la oportunidad estratégica.**

Figura 43: Equivalencia funcional entre los hyperscalers

Cada hyperscaler implementa Kimball base bajo su propia terminología, lo que confunde a quien intenta navegar el mercado por primera vez. La equivalencia funcional, sin embargo, es directa.

Microsoft Azure implementa Kimball con Synapse Analytics como ETL y warehouse, Power BI como BI App, Purview como gobernanza, y Fabric como capa unificadora. **AWS** implementa con Redshift como warehouse, QuickSight como BI App, DataZone como gobernanza, Lake Formation como capa de datos. **Google Cloud** implementa con BigQuery como warehouse, Looker como BI App, Dataplex como gobernanza. **Databricks** implementa con Lakehouse usando Medallion Architecture (Bronze/Silver/Gold) como ETL y warehouse, y Unity Catalog como gobernanza.

La Medallion Architecture de Databricks merece nota especial: es implementación moderna de Kimball sobre lakehouse. **Bronze** corresponde a Source — datos crudos. **Silver** corresponde a ETL transformado — datos limpios y conformes. **Gold** corresponde a Presentation — datos listos para consumo. La nomenclatura es nueva pero el concepto es Kimball clásico aplicado a lakehouse.

Ningún hyperscaler tiene aún implementación completa de la **capa agentiva** del Kimball Barnizada. La capa semántica explícita, el Trust Score por dato, la AI Certification, la observabilidad de queries agentivos — son piezas que cada hyperscaler está agregando gradualmente, pero ninguno tiene oferta integrada completa. **Quest/erwin** es uno de los pocos vendors que productiza el modelo extendido completo bajo el concepto *"Model to Marketplace"* — pero como producto especializado, no como parte del stack de un hyperscaler.

Esta brecha es oportunidad estratégica: el actor que entregue Kimball Barnizada completo, integrado

bajo una sola arquitectura coherente, captura el espacio que actualmente ningún actor cubre completamente.

El consenso de la industria

Diez actores convergen — desde ángulos distintos — en la misma visión de BI conversacional sobre arquitectura Kimball: Tableau, Cube, Tellius, ThoughtSpot, Salesforce, AtScale, Databricks, Informatica, eWeek y Gartner, cuyas formulaciones ya recorrieron este capítulo y el Capítulo 2. La lista deja claro que no es propuesta aislada de un actor — es consenso emergente del campo.

El patrón común es claro. La industria convencional reconoce que la BI tradicional debe extenderse con capacidad agentiva. La diferencia entre los actores está en cuán explícitamente se integra la arquitectura. Los actores que tratan la BI agentiva como feature suelto producen soluciones limitadas; los actores que la tratan como rediseño arquitectónico de la capa de datos producen soluciones que pueden sostenerse en producción.

¿Cómo se implementa el caso canónico?

Para una organización que tiene arquitectura Kimball razonable y desea agregar capacidad agentiva, la ruta de implementación sigue un patrón recurrente.

La **etapa uno** es construir o adquirir una **capa semántica explícita** que codifique el significado de las dimensiones, hechos, jerarquías y reglas de negocio. Sin esta capa, los agentes alucinan o producen consultas incorrectas. La organización tiene varias opciones de producto: AtScale como producto especializado, dbt Semantic Layer como capa que se integra al modelado dbt, Cube como producto con énfasis en performance, LookML como capa semántica de Looker para casos donde Looker ya es BI tool. La elección depende del stack existente y de las preferencias arquitectónicas. Lo crítico no es qué producto se elige — es construir la capa con disciplina.

La **etapa dos** es construir o configurar un **agente con acceso a la capa semántica**. El agente no genera SQL directamente — genera consultas semánticas que la capa semántica traduce a SQL correcto. Esto es lo que distingue a una BI conversacional robusta de una "demo de chatbot sobre data warehouse". El agente que genera SQL directamente alucina con frecuencia; el agente que opera sobre capa semántica con contratos claros mantiene precisión alta.

La **etapa tres** es aplicar **Trust Infrastructure** sobre el agente. Los cinco pilares — Gobernanza, Auditoría, Validación, Resiliencia, Transparencia — sobre el agente conversacional. Gobernanza define qué datos puede consultar, quién puede preguntar qué. Auditoría registra cada consulta y cada respuesta. Validación detecta especialmente alucinaciones financieras o de KPIs — categoría de error particularmente grave en BI conversacional. Resiliencia ante fallos de la capa semántica. Transparencia de qué tools invocó el agente para producir cada respuesta.

La **etapa cuatro** es **onboarding gradual**. Empezar con un dominio de datos limitado — finanzas, ventas, o uno específico — donde la calidad del modelo es alta y los usuarios son sofisticados. Expandir a más dominios solo después de validar que el agente entrega valor sin alucinar. La paciencia es clave — saturar todos los datos de la organización al agente desde el inicio garantiza que los primeros usuarios pierdan confianza por respuestas incorrectas. El proyecto tiene que ganar credibilidad antes de expandirse.

La **etapa cinco** es **evolución a empresa en tiempo real**. Una vez que el agente responde preguntas correctamente sobre datos históricos, evolucionar hacia que actúe sobre los datos: alertas proactivas cuando detecta anomalías, ejecución de procesos correctivos automáticos dentro de límites gobernados, escalación al humano para decisiones sobre umbral. Es la transición de empresa en línea a empresa en tiempo real descrita en el Capítulo 2. Esta etapa es donde el sistema agentivo pasa de ser **mejor BI** a ser **operación autónoma**.

¿Por qué este caso vale como aplicación canónica?

Las tres razones que abrieron este capítulo — universal, con métrica clara, construido sobre arquitectura existente — explican por qué el caso es fundacional. El cierre pide un ángulo distinto: lo que este caso hace por los demás.

El acceso a información en tiempo real es **condición habilitante** del resto de la transformación agentiva. Sin él, los agentes que actúan en otros dominios — operaciones, ventas, atención al cliente — carecen de fundamento informativo. Por eso este caso aparece típicamente como el primero en la trayectoria de adopción de la Arquitectura Agentiva por parte de organizaciones serias. Las organizaciones que intentan saltarse el caso del conocimiento en tiempo real para ir directamente a casos de operación autónoma típicamente fracasan — sin la base de información en tiempo real, los agentes operacionales operan a ciegas.

Y como construye sobre lo que ya existe, es también el caso de menor barrera: una organización puede ganar su primera experiencia agentiva — la pregunta respondida en segundos que convierte el escepticismo en demanda — sin comprometer presupuesto de transformación masiva. Esa primera experiencia es la que abre la puerta de los dominios siguientes.

> *Quien resuelve el conocimiento en tiempo real, gana el derecho de plantear los demás casos. Quien no, opera en piloto perpetuo.*

Capítulo 8 · Trust Infrastructure operacionalizada — políticas y CRUDLEX

El Capítulo 5 describió Trust Infrastructure como conjunto de cinco pilares — Gobernanza, Auditoría, Validación, Resiliencia, Transparencia — que atraviesa las cuatro capas de la Arquitectura Agentiva. La descripción era conceptual: nombró los pilares, sus mecanismos canónicos, sus propiedades exigidas. Este capítulo cierra el círculo: traduce esos conceptos en construcciones concretas que un implementador puede tomar y construir.

La distinción entre concepto y operacionalización es crítica para el éxito de un sistema agentivo en producción. Un pilar es **conceptual**: *"la organización debe poder gobernar lo que el agente hace"*. Una operacionalización es **constructiva**: *"la organización configura políticas en formato YAML que aplican modelo CRUDLEX, evaluadas en cada invocación de tool, con catálogo declarativo y herencia explícita"*. La distancia entre las dos es exactamente lo que separa un proyecto que pasa de piloto a producción del que engrosa la ola de cancelaciones que Gartner pronostica (Capítulo 2).

> *Los pilares responden qué confianza se necesita. Las operacionalizaciones responden cómo se construye.*

Este capítulo desarrolla la operacionalización completa de Trust Infrastructure. Catálogo de políticas, modelo CRUDLEX completo, formato del append-only log, protocolos de aprobación humana, reglas de detección de alucinaciones, política de tokenización. Cada componente con el detalle que un arquitecto necesita para implementar y un auditor necesita para evaluar. El capítulo es el más técnico del libro, y lo es deliberadamente — la Trust Infrastructure operacionalizada es donde la arquitectura se vuelve real.

Catálogo de políticas

Las **políticas** son reglas declarativas — no código embebido — que definen qué puede hacer un agente y bajo qué condiciones. La distinción entre política declarativa y código imperativo es importante. Una política declarativa puede modificarse sin redeploy del sistema, puede versionarse independientemente, puede ser auditada sin requerir review de código, puede ser comprendida por personas no-técnicas (compliance officers, lawyers, auditores). Un código imperativo enredado con políticas confunde tres roles distintos — desarrollador, compliance officer, auditor — en una sola superficie, y eso garantiza que los tres roles operen mal.

La operacionalización canónica define un catálogo con **cinco categorías** de políticas. Las cinco categorías son distintas, atacan problemas distintos, y un sistema agentivo bien diseñado tiene políticas

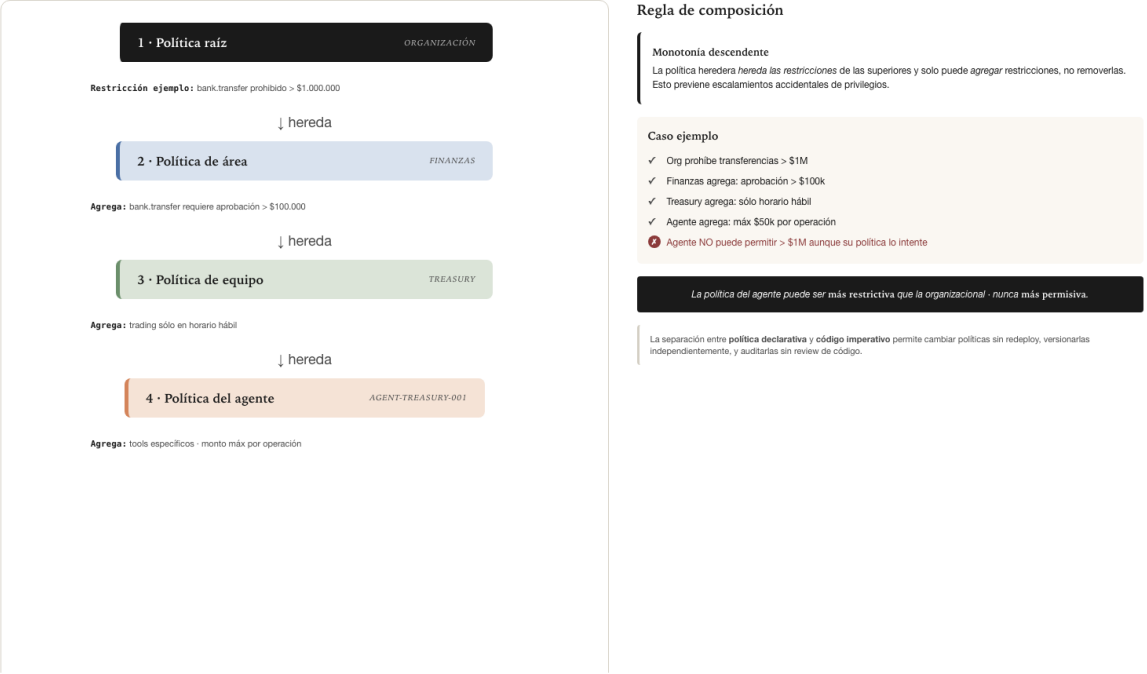


Figura 44: Composición jerárquica de políticas · monotonía descendente

activas en las cinco. Las desarrollamos una por una.

Políticas de tools

Las políticas de tools definen qué tools puede invocar el agente, sobre qué recursos. Son el primer mecanismo de control que un sistema agentivo necesita: si el agente puede invocar tools indiscriminadamente, no hay gobernanza posible. Las políticas de tools establecen el **alcance permitido** de las invocaciones del agente, con granularidad por agente, por tool, por scope (recursos específicos sobre los cuales el tool opera), y con condiciones especiales (aprobación requerida, prohibición categórica).

```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       # aprobación humana antes de ejecutar
    - name: bank.transfer
```

```
      allow: false                    # prohibido categóricamente
```

Una política de tools típica para un agente financiero podría permitir lectura del ledger sobre dos empresas específicas, permitir escritura solo sobre una de ellas pero requiriendo aprobación humana, y prohibir categóricamente cualquier operación de transferencia bancaria. La granularidad permite balance entre autonomía operativa y control de riesgo: el agente opera autónomamente para operaciones de bajo impacto, escala al humano para operaciones medias, no opera para operaciones de alto impacto.

Políticas de datos

Las políticas de datos definen qué clases de datos puede consultar o emitir el agente. Son complementarias a las políticas de tools — un tool puede estar permitido en general pero los datos específicos que devuelve pueden estar restringidos por clase. Las políticas de datos clasifican los datos por sensibilidad y aplican reglas distintas según la clase.

```
policy:
  type: data-access
  agent: agent-customer-support-001
  data_classes:
    - class: pii.name
      allow: read
    - class: pii.ssn
      allow: read
      require_tokenization: true    # se tokeniza antes de invocar cognición
    - class: pii.financial
      allow: false
    - class: internal.public
      allow: read-write
```

Un agente de customer support puede leer nombres de clientes (PII de baja sensibilidad), puede leer números de seguridad social pero solo tokenizados (la cognición ve el token, no el dato real), no puede acceder a datos financieros del cliente, y puede leer y escribir información interna pública. La granularidad por clase de dato permite que el agente opere productivamente con datos no sensibles sin exponer datos sensibles a los proveedores de cognición externos.

Políticas de horarios y umbrales

Las políticas de horarios y umbrales limitan cuándo o con qué umbrales el agente actúa. Capturan la dimensión temporal y de magnitud de las operaciones — qué hora es, cuán grande es la operación, con qué riesgo asociado.

```
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
```

Un agente de trading puede operar autónomamente para montos pequeños, requiere aprobación para montos medianos, no opera para montos grandes, y requiere aprobación para cualquier operación fuera de horario hábil. La construcción declarativa permite ajustar los umbrales sin modificar código — un cambio de política regulatoria que mueve el umbral de aprobación de cien mil a cincuenta mil dólares se ejecuta como cambio de configuración, no como release de software.

Políticas de identidad

Las políticas de identidad definen qué identidades operan en nombre del agente y cómo se autentican. Capturan el modelo de identidad federada típico de organizaciones complejas, donde un agente actúa en nombre de un usuario humano, autenticado por el identity provider corporativo, con scope limitado.

```
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"]
```

El agente se autentica vía OAuth2 contra el identity provider corporativo, opera en nombre de dos usuarios específicos (CFO y Controller), y su scope está limitado a las operaciones del agente financiero — no puede actuar como otro agente o asumir identidad distinta. La política de identidad es lo que asegura que las acciones del agente puedan rastrearse de regreso a humanos identificables, condición necesaria para auditoría seria.

Políticas de validación

Las políticas de validación definen qué validaciones se aplican antes de ejecutar acciones. Conectan con los mecanismos del Pilar 3 (Validación) que el Capítulo 5 describió, especificándolos en políticas concretas.

```
policy:
  type: validation
  agent: agent-customer-support-001
  rules:
    - validation: hallucination-check
      threshold: 0.95              # confianza mínima
    - validation: prompt-injection-check
      action_on_detection: block-and-alert
    - validation: dlp-scan
      data_classes: ["pii.*", "financial.*"]
      action_on_detection: tokenize
```

Un agente de customer support requiere validación de alucinaciones con umbral de confianza del

noventa y cinco por ciento — si el sistema de detección detecta menos confianza, la respuesta no se emite. Detecta intentos de prompt injection y, si detecta uno, bloquea la operación y alerta al equipo de seguridad. Aplica DLP scan sobre clases de datos sensibles — PII y datos financieros — y, si detecta datos que requieren tokenización, los tokeniza antes de que la cognición los procese.

Composición de políticas

Las políticas se componen jerárquicamente. La organización define una política raíz que aplica a todos los agentes; cada área define políticas específicas que heredan de la raíz; cada equipo refina aún más; cada agente tiene política específica que hereda de todas las anteriores.

La política del agente **hereda** las restricciones de las superiores y solo puede **agregar** restricciones, no removerlas. Esto previene escalamientos accidentales de privilegios. Si la política raíz de la organización prohíbe transferencias bancarias mayores a un millón, la política del agente no puede permitirlas — solo puede agregar restricciones adicionales (por ejemplo, prohibir transferencias mayores a cien mil para el agente específico). La composición jerárquica con monotonía descendente es la propiedad estructural que sostiene la gobernanza compleja.

Modelo CRUDLEX completo

Nivel preconfigurado	C CREATE	R READ	U UPDATE	D DELETE	L LIST	E EXECUTE
FULL autoridad completa	✓	✓	✓	✓	✓	✓
READ-WRITE operación normal sin destrucción	✓	✓	✓	–	✓	✓
READONLY consulta pura	–	✓	–	–	✓	–
SAFE acción ejecutiva limitada	–	✓	–	–	✓	limitado
NO-SEND edición sin acciones externas	✓	✓	✓	–	✓	–
NO-DELETE operación normal sin destrucción	✓	✓	✓	–	✓	✓

Read vs. List
Distinción crítica. Read permite consultar uno por ID; List permite enumerar muchos por filtro. Un agente con Read pero sin List es seguro contra exfiltración masiva.

Update vs. Execute
Update modifica un recurso; Execute dispara un proceso con efecto colateral (envío de email, transferencia, workflow externo). Perfiles de riesgo distintos.

CRUDLEX efectivo = intersección
(usuario × agente × contexto). Si el usuario tiene Read pero no Write, el agente que actúa en su nombre tampoco puede escribir. Ninguna dimensión escala privilegios.

Figura 45: Niveles preconfigurados × las seis operaciones

CRUDLEX es la operacionalización canónica de permisos granulares para sistemas agentivos. El modelo extiende el clásico CRUD con dos operaciones críticas en sistemas agentivos: **List** y **Execute**. La extensión no es decoración — refleja distinciones operativas que el modelo CRUD tradicional no

capturaba pero que en sistemas agentivos importan.

Las seis operaciones canónicas son las siguientes. **Create** corresponde a crear un nuevo recurso — crear un ticket, agregar un registro. **Read** corresponde a leer un recurso específico — consultar un cliente por ID. **Update** corresponde a modificar un recurso existente — actualizar el estado de una orden. **Delete** corresponde a eliminar un recurso — borrar un comentario. **List** corresponde a enumerar recursos con filtros — listar tickets abiertos. **Execute** corresponde a invocar una operación con efecto colateral — enviar email, ejecutar pago, disparar workflow.

La distinción entre **Read** (leer un recurso específico) y **List** (enumerar recursos según filtro) es deliberada y crítica. Un agente puede tener permiso de Read sobre clientes individuales pero no de List sobre toda la base — para evitar que extraiga el catálogo completo de clientes incluso teniendo permiso de leer cada uno individualmente. Esta distinción no existe en CRUD clásico, pero en sistemas agentivos es donde se materializan ataques de exfiltración masiva: un agente con Read sin List es seguro; un agente con List sin restricciones puede dump la base entera con una sola consulta.

La distinción entre **Update** (modificar) y **Execute** (operación con efecto colateral) también es deliberada. Update modifica un recurso; Execute dispara un proceso que puede tocar múltiples recursos o tener efectos en el mundo real — envío de email, transferencia bancaria, ejecución de workflow externo. Las dos operaciones tienen perfiles de riesgo distintos: un Update tiene impacto contenido; un Execute puede tener impacto que se propaga. CRUDLEX las trata diferenciadamente para que las políticas puedan aplicarse con precisión.

Aplicación de CRUDLEX

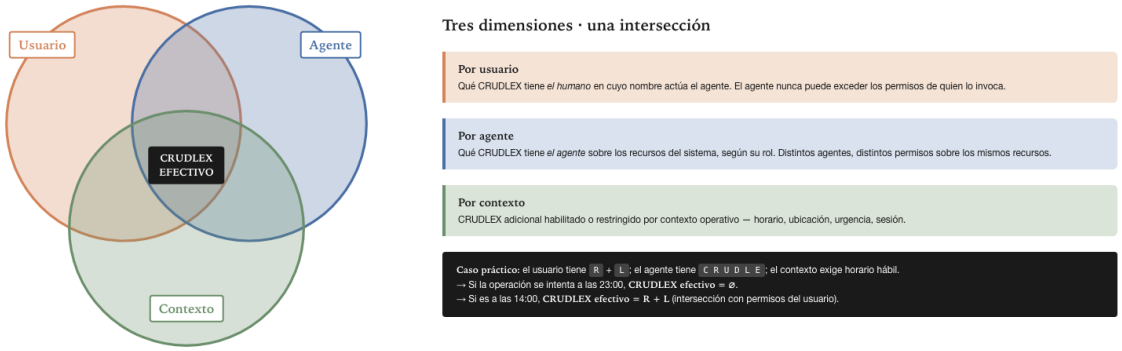


Figura 46: Las tres dimensiones de CRUDLEX · intersección como permiso efectivo

CRUDLEX se aplica por **tres dimensiones**: usuario, agente, contexto. Las tres dimensiones operan independientemente y se componen para producir el permiso efectivo de una operación particular.

La dimensión **por usuario** captura qué CRUDLEX tiene **el humano** en cuyo nombre actúa el agente. Si el usuario humano no tiene Update sobre cierto recurso, el agente que actúa en su nombre tampoco lo tiene — el agente no puede exceder los permisos de quien lo invoca. La dimensión **por agente** captura qué CRUDLEX tiene **el agente** sobre los recursos del sistema. Distintos agentes pueden tener distintos permisos sobre los mismos recursos según su rol. La dimensión **por contexto** captura CRUDLEX adicional habilitado o restringido por contexto operativo — horario, ubicación, urgencia.

El **CRUDLEX efectivo** de una operación es la **intersección** de las tres dimensiones. Si el usuario tiene Read pero no Write, y el agente tiene Read+Write, **la operación efectiva es solo Read** — el agente no puede exceder el alcance del usuario en cuyo nombre actúa. La intersección como mecanismo de composición es propiedad importante: garantiza que ninguna dimensión puede escalar privilegios más allá de lo que las otras dimensiones permiten.

Niveles preconfigurados

Para evitar configuraciones ad hoc para cada caso, la especificación canónica define **niveles preconfigurados** por convención. Los niveles cubren los casos comunes; casos especiales se configuran granularmente.

Nivel	CRUDLEX habilitado	Uso típico
FULL	C R U D L E	Agente con autoridad completa sobre el sistema
READ-WRITE	C R U D L (no E)	Escritura total de datos —incluido borrado— sin disparar acciones externas
READONLY	R L	Consulta pura
SAFE	R L E (con E limitado a operaciones reversibles)	Acción ejecutiva limitada
NO-SEND	C R U L (no E)	Edición sin disparar acciones externas
NO-DELETE	C R U L E (no D)	Operación normal sin destrucción

Los niveles FULL y READONLY son extremos que casi nunca se usan en producción — FULL es demasiado permisivo, READONLY es demasiado restrictivo. Los niveles intermedios — READ-WRITE, SAFE, NO-SEND, NO-DELETE — cubren los casos típicos. Un agente de soporte al cliente típicamente opera en SAFE: puede consultar, listar, ejecutar acciones reversibles (como reasignar un ticket), pero no puede crear ni borrar irreversiblemente. Un agente de back-office operativo típicamente opera en NO-DELETE: puede crear, leer, actualizar, listar, ejecutar — pero no puede borrar registros, porque la auditoría exige preservación.

Formato del append-only log

El **append-only log** es el componente central de la auditoría. La especificación canónica define formato y propiedades con detalle suficiente para que un implementador construya un log conforme y un auditor

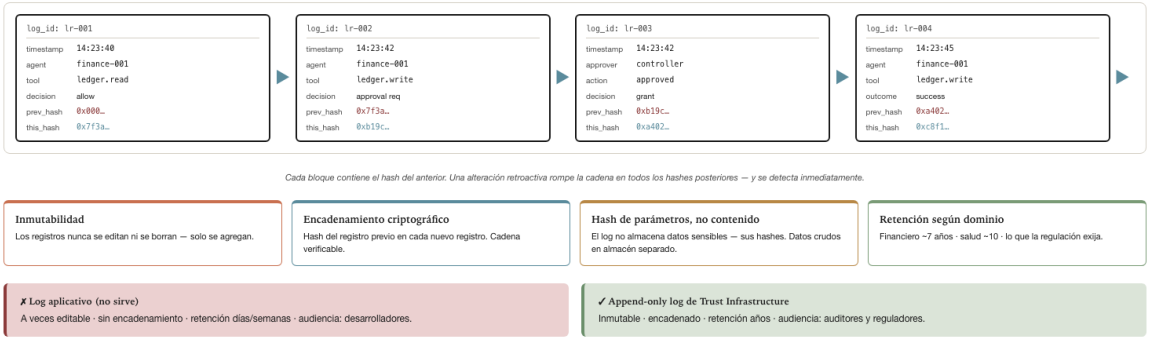


Figura 47: Cadena criptográfica del log · inmutabilidad por construcción

evalúe si un log particular es conforme.

Estructura del registro

Cada registro del log contiene un conjunto definido de campos:

```
{
  "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:..."
}
```

Cada campo cumple un propósito específico. **log_id** es identificador único del registro. **timestamp** es momento de la operación con precisión milisegundo. **agent_id** identifica al agente. **trace_id** correlaciona con otros eventos del mismo trace (otras operaciones que parten de la misma solicitud original). **operation** describe qué se hizo. **context** captura quién pidió, qué Capability se aplicó. **policy_evaluation** registra qué políticas se evaluaron y con qué decisión, incluyendo aprobación humana cuando se requirió. **outcome** registra el resultado de la operación. **previous_log_hash** es el hash del registro anterior — formando la cadena criptográfica.

Propiedades exigidas

El log debe satisfacer cinco propiedades innegociables. La **inmutabilidad** asegura que registros nunca se editan ni se borran — solo se agregan. El **encadenamiento criptográfico** asegura que cada registro contiene el hash del anterior, formando una cadena verificable; una alteración retroactiva sería detectable inmediatamente. El **hash de parámetros y resultados, no contenido** asegura que el log no almacena los datos sensibles (parámetros, resultados completos), sino sus hashes — los datos crudos viven en almacén separado con políticas de retención específicas. La **retención mínima configurable** asegura que la política de retención se define por dominio: financiero típicamente siete años, salud típicamente diez años, lo que la regulación aplicable exija. El **formato consultable** asegura que el log es consultable por filtros — por agente, por usuario, por trace, por rango de fechas, por tipo de operación.

Diferencia con logs aplicativos

El append-only log de Trust Infrastructure es **distinto** del log aplicativo (que registra eventos de operación normal). Las diferencias son sustantivas y los dos sistemas no deben confundirse.

Dimensión	Log aplicativo	Append-only log de Trust
Propósito	Diagnóstico, debug	Auditoría, conformidad
Mutabilidad	A veces editable	Nunca
Encadenamiento criptográfico	Raro	Obligatorio
Retención	Días o semanas	Años (regulatorio)
Audiencia	Desarrolladores	Auditores, reguladores

Una organización seria mantiene **ambos** y los integra solo donde corresponde. El log aplicativo sirve al equipo de operaciones para diagnosticar problemas; el append-only log de Trust sirve a la organización para defenderse ante auditorías y reguladores. Confundir los dos — usar log aplicativo para auditoría, o agregar peso de auditoría al log aplicativo — termina sirviendo mal a ambos propósitos.

Protocolo de aprobación humana

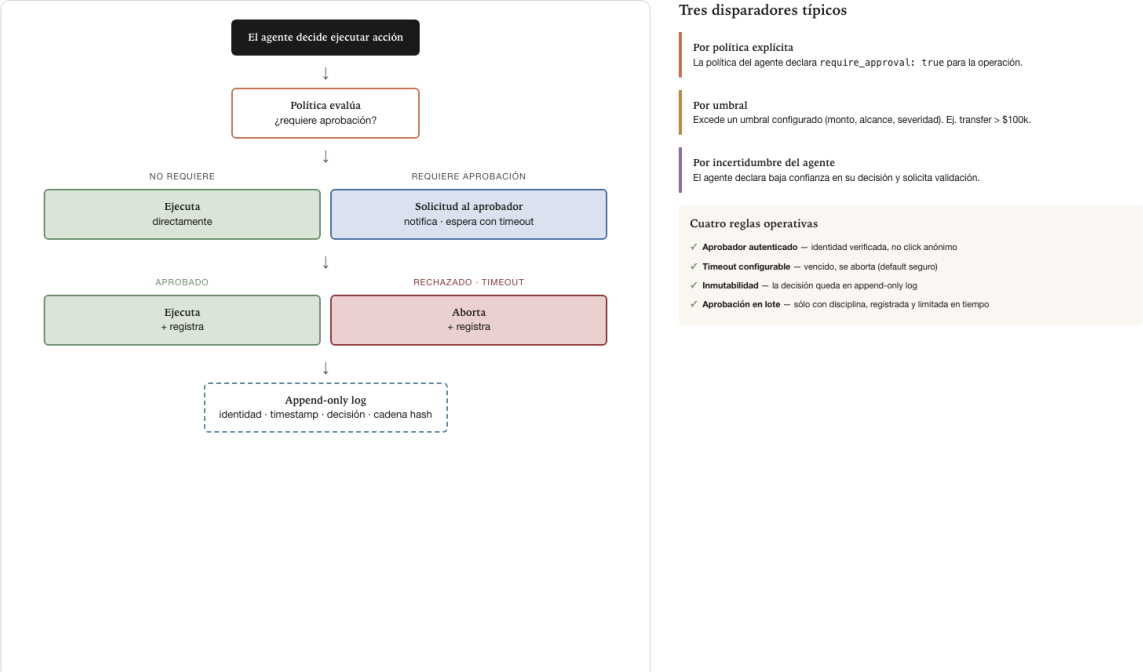


Figura 48: Protocolo de aprobación humana · flujo canónico

Cuando una operación requiere aprobación humana antes de ejecutarse, la especificación canónica define el protocolo con precisión suficiente para que la implementación sea verificable.

Disparadores de aprobación

Tres disparadores típicos activan el flujo de aprobación humana. El primero es **por política explícita**: la política del agente declara `require_approval: true` para la operación. Es el caso más simple — la política está predefinida y el sistema la aplica automáticamente. El segundo es **por umbral**: la operación excede un umbral configurado — monto, alcance, severidad. Por ejemplo, un agente de pagos que opera autónomamente para montos pequeños puede requerir aprobación cuando el monto excede los cien mil dólares. El tercero es **por incertidumbre del agente**: el agente mismo declara baja confianza en su decisión y solicita validación. Es disparador menos común pero útil para casos donde el agente reconoce que está fuera de su zona de confianza.

Flujo canónico

Cuando un disparador activa la necesidad de aprobación, el flujo canónico es el siguiente. El agente decide ejecutar acción. La política evalúa si requiere aprobación. Si no requiere, el agente ejecuta y registra. Si requiere, el sistema crea solicitud de aprobación, notifica al aprobador o aprobadores configurados, espera respuesta con timeout. Si la respuesta es aprobación, ejecuta. Si la respuesta es rechazo o vence el timeout, aborta. En cualquier caso, registra en el append-only log toda la cadena.

Reglas de la aprobación

La aprobación tiene cuatro reglas operativas que la especificación exige.

La primera regla: **aprobador autenticado**. La aprobación queda atada a la identidad verificada del aprobador, no a un click anónimo. Si el aprobador es Juan Pérez con credenciales corporativas, la aprobación queda registrada bajo identidad verificada — no como aprobación genérica de "alguien que tenía acceso al sistema". Sin esta regla, la aprobación pierde valor regulatorio.

La segunda regla: **timeout configurable**. Cada operación tiene timeout para la aprobación. Si vence sin aprobación, la operación se aborta — default seguro. En casos especiales (operaciones críticas urgentes), puede configurarse escalación automática a un aprobador secundario, pero la escalación misma queda registrada y debe estar permitida por política.

La tercera regla: **inmutabilidad del registro**. La decisión del aprobador queda en el append-only log con identidad, timestamp y, opcionalmente, comentario. Una vez registrada, no se puede modificar. Esto protege tanto al aprobador (su decisión queda como la tomó) como a la organización (la cadena de aprobaciones es trazable).

La cuarta regla: **aprobación en lote** para casos repetitivos. Para operaciones repetitivas y de bajo riesgo, puede configurarse aprobación de patrón — *"pre-aprueba todas las operaciones de tipo X durante las próximas 4 horas"*. Pero la pre-aprobación misma es operación que queda registrada, con identidad del aprobador, alcance del patrón, y duración. La aprobación en lote es eficiente operativamente pero requiere disciplina para no convertirse en aprobación genérica que vacía el modelo.

Reglas de detección de alucinaciones

La validación de respuestas — pilar 3 de Trust Infrastructure — opera con un conjunto de reglas que describimos a continuación. Cada una ataca un tipo distinto de error del agente.

La regla de **self-consistency** funciona así: el agente formula la misma pregunta de N maneras distintas. Si las N respuestas son consistentes, la confianza aumenta. Si difieren, alerta. La técnica explota una propiedad de los modelos LLM: cuando el modelo está cierto, sus respuestas son estables a través de pequeñas variaciones de la pregunta; cuando está alucinando, las respuestas son inestables. Self-consistency es relativamente barata y útil para detectar alucinaciones flagrantes.

La regla de **retrieval-augmented verification** funciona así: antes de afirmar un hecho, el agente busca evidencia en su corpus de datos confiables. Si encuentra evidencia consistente, afirma con confianza alta. Si no encuentra evidencia, responde con incertidumbre explícita o se abstiene. Esta técnica requiere que la organización mantenga corpus de datos confiables — bases de conocimiento curadas, no datos de internet sin verificar.

La regla de **model-as-judge** funciona así: un segundo modelo, idealmente distinto del primero, evalúa

la respuesta del agente y emite un score de calidad. Si el score está bajo umbral, alerta o re-pregunta. La técnica es costosa (duplica la inferencia) pero útil para casos críticos donde el costo del error supera el costo de la validación.

La regla de **constraint validation** funciona así: las respuestas estructuradas — JSON, tablas, números — se validan contra un schema explícito. Si no cumplen, se reformulan o se escalan. Esta técnica es prácticamente gratuita y debe estar siempre activa para respuestas estructuradas — no usarla es desperdicio.

La regla de **domain-specific guardrails** funciona así: reglas específicas del dominio operan como verificación adicional. En finanzas, validar que cifras cuadren en sumas de control. En salud, validar que diagnósticos referencien guías clínicas reconocidas. En legal, validar que citas a leyes existan. Estas reglas son específicas al dominio y exigen inversión de los expertos del dominio para construirlas.

Las cinco reglas se aplican selectivamente según el contexto y costo. No todas se activan en cada operación — se activan según la criticidad de la decisión. Un agente que responde preguntas casuales sobre el clima no necesita las cinco; un agente que toma decisiones de inversión las necesita todas y posiblemente más.

Política de tokenización

La tokenización reemplaza datos sensibles por tokens antes de que lleguen al modelo de cognición. Es mecanismo crítico cuando la organización opera datos sensibles que no pueden exponerse al proveedor de cognición externo, pero que el agente necesita poder razonar para entregar valor.

Datos típicamente tokenizados

Cuatro categorías de datos típicamente requieren tokenización. **PII** (Personally Identifiable Information) — nombres, direcciones, teléfonos, emails — por razones de privacidad y compliance. **Datos financieros** — números de cuenta, tarjetas de crédito, balances — por regulación financiera. **Datos de salud** — identificadores médicos, diagnósticos, condiciones — por HIPAA y equivalentes. **Secretos corporativos** — API keys, contraseñas, datos clasificados — por seguridad operativa.

Mecanismo

El mecanismo canónico opera así. El dato original llega al servicio de tokenización, que devuelve un token opaco que reemplaza al dato. El servicio mantiene mapping interno entre el token y el dato original, en almacén con seguridad reforzada. El token opaco es lo que recibe la cognición — el modelo razona sobre el token sin saber qué representa. Cuando el agente toma una decisión y emite resultado, el resultado puede contener el token; el resultado vuelve a pasar por el servicio de tokenización, que de-tokeniza solo donde se requiere — por ejemplo, en la entrega final al usuario humano autorizado o en la invocación de un tool que requiere el dato real.

1. El **dato original** llega al servicio de tokenización.
2. El servicio devuelve un **token opaco** y mantiene el mapping token↔dato en un almacén con seguridad reforzada.
3. La **cognición razona sobre el token**, sin saber qué representa.
4. El agente emite su **decisión**, que puede contener el token.
5. El resultado vuelve al servicio de tokenización, que **de-tokeniza solo donde se requiere** — la entrega final al humano autorizado o la invocación de un tool que necesita el dato real.

El modelo nunca ve el dato original. La de-tokenización ocurre solo en el punto donde el dato real es necesario.

Reglas operativas

Tres reglas operativas exigen disciplina en la implementación. La primera: **tokenización antes de cognición externa**. Si el agente usa un proveedor de cognición de terceros — Claude, GPT, Gemini —, los datos sensibles se tokenizan **antes** de salir del perímetro corporativo. Sin esta regla, los datos se exponen al proveedor incluso aunque la organización quisiera evitarlo. La segunda: **de-tokenización auditada**. Cada de-tokenización queda registrada en el append-only log. Permite reconstruir, después, dónde el dato original fue expuesto. La tercera: **mapping en almacén separado**. El mapeo token-a-dato-original vive en almacén con seguridad reforzada — típicamente un HSM (Hardware Security Module) o servicio dedicado de tokenización. Mantener el mapping en la misma base de datos que los datos operativos vacía la garantía de tokenización.

Catálogo mínimo viable

Una organización que opera agentes en producción debe tener, como **mínimo viable**, los siguientes componentes operacionalizados. Por debajo de este mínimo, la operación de agentes deja a la organización expuesta. Por encima, hay refinamiento progresivo según madurez y exigencia regulatoria del dominio. Nótese que este catálogo **endurece deliberadamente** algunos SHOULD de la spec (Capítulo 5 §4) a piso operativo — la detección de alucinaciones entre ellos: lo que la spec deja opcional para implementaciones mínimas, la operación enterprise lo exige.

Componente	Mínimo viable
Catálogo de políticas	Al menos políticas de tools, datos, horarios, identidad, validación
CRUDLEX	Aplicado en todos los tools de Capa 4, con niveles preconfigurados
Append-only log	Inmutable, encadenado, retención según dominio
Aprobación humana	Configurable por operación, con timeout y registro
Detección de alucinaciones	Al menos self-consistency + domain guardrails, más constraint validation siempre activa en respuestas estructuradas
Tokenización	Para PII y datos financieros si aplica
Inventario de agentes	Quién opera, qué Capabilities, qué tools, quién aprobó
Dashboard de gobernanza	Visualización del inventario + métricas operativas

Lo crítico de este catálogo no es la lista en sí — es que está construida con la misma disciplina con que cualquier organización seria construye su sistema de control financiero o su sistema de seguridad. Trust Infrastructure no es subproducto de tener IA agentiva. Es disciplina específica que la organización adopta cuando decide operar agentes en producción.

El patrón tripartito de despliegue — Cloud + Cliente + Local

Figura 49: Patrón tripartito · Cloud + Cliente + Local

Operacionalizar Trust Infrastructure en una organización enterprise no es ejercicio puramente técnico — es ejercicio de separación de responsabilidades entre tres planos que operan en lugares físicamente distintos. El patrón que esta operacionalización adopta como canónico para resolver este problema es el **despliegue tripartito**: tres componentes coordinados que viven en tres lugares con tres roles funcionales diferenciados.

El primer componente es el **Cloud** — el plano operado por el proveedor de la infraestructura agentiva. Su rol es **control plane**: gestión de licencias, registry de providers disponibles, integración con proveedores de cognición upstream, telemetría agregada con privacidad preservada. Es donde el proveedor mantiene viva la plataforma y resuelve los problemas que requieren visibilidad cross-cliente — incidentes que afectan a múltiples tenants, actualizaciones del catálogo de políticas canónicas, monitoreo de salud agregado del sistema. El cliente final no opera este componente; lo consume.

El segundo componente es el **Cliente** — el plano desplegado dentro de la red interna de la organización cliente. Su rol es **governance plane**: definición y enforcement de las políticas corporativas, integración con el identity provider del cliente, auditoría central de todas las acciones del agente, conexiones a los sistemas internos a los cuales el agente debe acceder. Aquí es donde se ejerce la gobernanza enterprise — porque la gobernanza vive donde la organización tiene control directo, no donde un proveedor externo la promete. El componente Cliente es lo que permite a la organización ejercer Trust Infrastructure sobre sus propios términos.

El tercer componente es el **Local** — el plano desplegado en el dispositivo de cada usuario. Su rol es **execution plane**: la app que conecta al agente del usuario (Claude Code, Cursor, Windsurf, cualquier

cliente MCP) con el componente Cliente y con los providers locales. Aquí es donde la operación efectivamente ocurre — donde el agente del usuario invoca tools, donde las credenciales del usuario viven encriptadas localmente, donde la latencia de la operación se materializa.

La razón por la cual este patrón es estructural y no arbitrario es que cada uno de los tres planos resuelve un problema que los otros dos no pueden resolver bien. El Cloud no puede ejercer la gobernanza corporativa porque no vive dentro del perímetro de la empresa. El Cliente no puede mantener viva la plataforma sola porque cada empresa replicaría el trabajo común. El Local no puede tomar decisiones de gobernanza porque está bajo control del usuario individual, no de la organización. Los tres planos cooperan, pero ninguno sustituye a los otros.

> *La política se define donde la empresa controla. La IA opera donde el usuario trabaja. El ecosistema se mantiene donde el proveedor puede operarlo.*

Este patrón es replicable por cualquier actor que opere infraestructura agentiva enterprise. La spec exige que los tres planos estén claramente separados — la mezcla de responsabilidades entre ellos produce sistemas que el cliente no puede gobernar (cuando la gobernanza vive en cloud del proveedor) o que el proveedor no puede operar (cuando todo vive en la red del cliente). La separación es la propiedad fundamental.

La economía de Trust Infrastructure operacionalizada

Implementar Trust Infrastructure tiene costo. Vale ser explícito sobre el costo porque la decisión de invertir o no invertir afecta directamente la viabilidad del sistema agentivo en producción.

El costo tiene tres componentes. La **construcción inicial** es one-time: catálogo de políticas, modelo CRUDLEX, append-only log con encadenamiento criptográfico, mecanismos de validación, integración con servicios de tokenización. Es trabajo de varios meses para una organización mediana. El **mantenimiento operativo** es continuo: agregar políticas nuevas según el sistema evoluciona, monitorear que las existentes siguen vigentes, ajustar umbrales según aprendizaje, mantener el inventario actualizado. Es trabajo continuo de un equipo dedicado. El **sobrecosto de operación** es por operación: latencia y cómputo de validar antes de actuar, registrar después, evaluar políticas en cada invocación. Es porcentaje sobre el costo de operación normal — típicamente del cinco al veinte por ciento según implementación.

El costo es real, pero **es menor que el costo de no tener Trust Infrastructure**. La proyección de cancelaciones de Gartner que el Capítulo 2 documenta pende sobre todo este capítulo — los controles de riesgo inadecuados son su tercera causa —, y el mismo capítulo registra la escala de comportamientos riesgosos que las organizaciones reportan de sus agentes. El costo de un proyecto cancelado o de un incidente — fuga de datos, decisión incorrecta de alto impacto, fallo regulatorio — es típicamente mucho mayor que el costo de prevenirlo.

> *Trust Infrastructure no es gasto. Es seguro contra el costo asimétrico de su ausencia.*

La asimetría del costo es lo que justifica la inversión. Implementar Trust Infrastructure cuesta cantidad relativamente predecible. No implementarla expone a costos potencialmente catastróficos cuyo monto la organización no puede acotar previamente. El balance, cuando se calcula con disciplina, favorece la inversión.

Continuidad operacional — operacionalización del segundo mecanismo

El Capítulo 5 §4 formaliza la distinción entre los dos mecanismos complementarios de continuidad — fallback agéntico y continuidad de negocio operacional — y se asume aquí como dada. El primero ya está cubierto operacionalmente por la spec de los Lets (Cap 5 §2 y §7) y la garantía de no-detención de la capa (Cap 4). El segundo necesita operacionalización propia — la sección que sigue la entrega.

Protocolo de continuidad por sitio físico

Cada sitio físico operativo de un AgencyDomain con Capa 3 distribuida debe tener un **protocolo de continuidad documentado**. La spec define el contenido mínimo del documento. Sin ese contenido, el sitio no es operable bajo la spec en escenarios de continuidad — no porque la arquitectura falle, sino porque la operación humana no tiene guía cuando los componentes computacionales caen.

El contenido mínimo comprende cinco elementos:

- **Rol del operador en modo continuidad** — quién hace qué cuando el sistema cae. La definición debe ser nominal por puesto, no por persona. *"El cajero pasa a modo continuidad y registra ventas en el cuaderno foliado de respaldo. El supervisor del local valida cada turno cerrado en modo continuidad."*
- **Registros físicos de respaldo** — cuadernos pre-foliados, talonarios numerados, tarjetas de comanda, formularios oficiales del regulador cuando apliquen. Cada registro físico es la fuente de verdad temporal mientras dura la continuidad; cuando la red vuelve, su contenido se ingresa al sistema.
- **Umbrales de activación** — a partir de qué tiempo de inactividad el sitio pasa a modo continuidad. Recomendación canónica: diez minutos para gastronomía y retail, treinta segundos para cobro tarjeta con autorización online, inmediato para emisión de comprobante regulado. Los umbrales son política del sitio, ajustables.
- **Procedimiento de retorno** — cómo se ingresan retroactivamente al sistema los registros físicos cuando la red vuelve. Debe explicitar quién ingresa, en qué orden, cómo se reconcilian conflictos con eventos que pudieron haberse procesado parcialmente antes del corte.
- **Frecuencia de simulacros** — cada cuánto se ejercita el protocolo. Recomendación canónica: trimestral con corte de red programado y duración mínima de quince minutos. Sin simulacros, el protocolo existe solo en papel y falla cuando se necesita en serio.

Modos de degradación del AgencyDomain

Una operación en producción no opera siempre en modo normal. La spec formaliza los **cuatro modos canónicos de degradación** según el escenario de falla, y la organización debe poder identificar en cada momento en qué modo opera cada sitio. Esta identificación es lo que permite gobernar las expectativas operativas.

Modo	Condición	¿Quién sostiene la operación?
Normal	Todos los componentes activos: cognición, central, edge, red corporativa.	Topología paralela completa. La operación elige la vía Cognición o la vía Autonomía según el patrón.

Modo	Condición	¿Quién sostiene la operación?
Cognición caída	Capa 2 inalcanzable; central y edge activos.	**Vía Autonomía** sostiene. Los Botlets senior ejecutan; los Botlets junior y en aprendizaje degradan a su última versión funcional. La cognición rescata fallos al volver.
Edge offline	Botler edge sin conexión al central; cognición inalcanzable; sitio físico aislado.	**Botlets senior contra BD local + Conectores edge-resident**. La cola de eventos hacia central acumula; al volver la red, drena y reconcilia.
Continuidad operacional total	Cognición + edge caídos por causa exógena (corte de energía, hardware destruido, red catastróficamente perdida).	**Protocolo manual del sitio**. El registro físico es la fuente de verdad temporal; el ingreso retroactivo al sistema es la reconciliación.

La transición entre modos es **automática hasta `Edge offline`** — el sistema detecta la falla y degrada solo. La transición a **Continuidad operacional total** es **gobernada por el protocolo del sitio** — un humano la activa explícitamente cuando reconoce que ningún componente computacional opera. Esta diferencia importa: los primeros tres modos son responsabilidad de la arquitectura; el cuarto es responsabilidad del cliente, ejecutado por sus operadores.

Trazabilidad de la transición a modo continuidad

El append-only log debe registrar las transiciones entre modos para que la auditoría posterior pueda reconstruir qué sostuvo la operación en cada momento. La spec define las marcas canónicas:

- **Inicio de continuidad operacional** — cuando el sitio activa el protocolo manual, emite (cuando vuelve la red, retroactivamente) un evento `mode-change: continuity-operational` con timestamp del corte y duración estimada.
- **Registros físicos ingresados retroactivamente** — cada transacción ingresada desde registro físico lleva tag `provenance: manual-continuity` y `original-timestamp: <hora física>` distinto del `system-timestamp: <hora de ingreso>`. La distinción permite que reportes y reconciliaciones distingan eventos físicos de eventos digitales.
- **Reconciliación de cola edge** — cuando un Botler edge drena su cola hacia central tras volver la red, los eventos llevan tag `provenance: edge-queue-replay` con el timestamp original del sitio.
- **Distinción auditable entre fallback agéntico y continuidad operacional** — el log distingue eventos `agentic-fallback` (la cognición rescató al Let — ejecutó por el Botlet o recibió el escalamiento del Agentlet) de `operational-continuity` (humano sostuvo la operación). Auditoría posterior puede separar ambos casos sin ambigüedad. Esta distinción es lo que la organización presenta cuando un regulador pregunta cómo operó durante un incidente.

Propiedades exigidas operacionales

Propiedad	Nivel
Distinción explícita en documentación de producto	**MUST**
Protocolo de continuidad documentado por sitio físico	**MUST**
Simulacros documentados con frecuencia mínima trimestral	**SHOULD**
Cuatro modos de degradación reconocibles por la organización	**MUST**
Trazabilidad de transición a modo continuidad en append-only log	**MUST**
Distinción auditable entre fallback agéntico y continuidad operacional	**MUST**
Reconciliación retroactiva de registros físicos al sistema	**MUST**

La operacionalización completa del segundo mecanismo cierra el círculo abierto en Cap 5 §4. La distinción entre fallback agéntico y continuidad operacional ya no es solo conceptual — tiene protocolo de campo, modos canónicos de operación y trazabilidad auditable.

Lo que sigue

Cerrado el bloque operativo, el libro cierra con la implementación de referencia — el Capítulo 9 — y su epílogo. Quien haya leído los capítulos anteriores con la sensación de que cada uno deja preguntas abiertas tendrá ahí la confirmación explícita de que esa lectura es correcta: lo que sigue no es un cierre sino el reconocimiento de que esta especificación es un punto de partida formalizado, no un destino final.

Capítulo 9 · Vergis — la implementación de referencia

Los capítulos anteriores entregaron una especificación: primitivas, propiedades exigidas, separaciones canónicas, contratos declarativos. Una especificación, por precisa que sea, no es un sistema vivo. Entre el contrato y la práctica hay un salto que cada lector externo intuye al cerrar el último capítulo de la spec: *"¿cómo se ve esto cuando funciona?"*. Esta es la sección que responde esa pregunta con un artefacto concreto, descargable y ejecutable.

Nota de alcance del canon

Conviene fijar primero qué es y qué no es este canon, porque la respuesta delimita el rol de todo lo que sigue.

> Este canon contiene la **estructura y el vocabulario** del Mundo Agentivo: definiciones, primitivas, propiedades exigidas, separaciones canónicas. **No contiene métodos para implementar ni catálogos operacionales** — esos viven en cuerpos complementarios. La implementación de referencia pública es **AgencyDomains.org**, materializada en **Vergis**: diseñada para que cualquier desarrollador o estudiante pueda descargarla, leerla, ejecutarla y aprender cómo el canon se traduce en sistemas vivos. Otros implementadores —productos comerciales, códices propietarios— ofrecen sus propios cuerpos complementarios sobre la misma estructura canónica.

La separación es deliberada. El canon describe *qué* propiedades satisface un AgencyDomain conforme; no prescribe *cómo* se descubre un caso de uso, *cómo* se forja un proto-Botlet, ni *qué* proto-Botlets concretos debe contener un catálogo. Esos son método y catálogo — materia de los cuerpos complementarios. Sin esta nota, el lector se preguntaría por qué la spec no incluye un capítulo sobre Discovery o un repertorio de proto-Botlets listos. No los incluye porque no le corresponden: pertenecen al nivel de la implementación, y la implementación de referencia es donde se vuelven visibles.

Esta nota acota el eje *canon vs implementación de referencia* — qué pertenece a la spec y qué a quien la implementa. Es distinta de la declaración de **lo que el libro deliberadamente omite** (verticales detallados, código operativo, análisis de ROI por stack), que vive en el Epílogo, sección "Lo que NO está en este libro".

¿Qué es y por qué existe?

Vergis es la implementación de referencia pública de AgencyDomains — el AgencyDomain hecho operativo. Es el puente concreto entre el canon y la práctica: un sistema que cualquiera puede descargar, leer, ejecutar y contribuir, construido para demostrar que las primitivas de la spec no son teoría sino piezas que componen un runtime que funciona.

Una especificación sin implementación de referencia corre el riesgo de quedar como ejercicio formal sin tracción. La implementación de referencia cumple cuatro funciones que el texto del canon por sí solo no puede cumplir. Da al lector externo una **puerta de entrada concreta** después de leer la spec. Prueba que el canon es **implementable** —la implementación es la prueba viva—. Habilita un **modelo de adopción claro**: descargar, operar, extender, contribuir. Y ancla el vocabulario abstracto en componentes que se pueden inspeccionar línea a línea.

Vergis no es tutorial ni juguete pedagógico. Es el sustrato sobre el que corren sistemas productivos. Esa cualidad —que se desarrolla más abajo— es lo que distingue una implementación de referencia seria de una colección de ejemplos.

¿Dónde vive?

Vergis vive en un repositorio público enlazado desde **AgencyDomains.org**. El código se distribuye bajo **`AGPL`** (Affero General Public License); la documentación, bajo **`GFDL`** (GNU Free Documentation License).

La elección de licencias es estructural, no incidental. La `AGPL` garantiza que las mejoras al runtime —incluidas las que operan como servicio de red— permanezcan disponibles para la comunidad: quien despliega una versión modificada de Vergis y la ofrece por red MUST publicar su fuente. La `GFDL` mantiene la documentación libre y derivable. Juntas, sostienen la promesa de una base común que ningún actor puede cerrar: la plataforma de referencia permanece abierta aunque los catálogos que se construyan sobre ella sean privados.

¿Cómo se nombran las piezas? — Vergis · Botler · Mira · Daftar

Cuatro etiquetas de **tres naturalezas distintas** intervienen en la implementación de referencia. Confundir su naturaleza —en particular, confundir un tipo con un nombre propio— oscurece la arquitectura. El esquema:

Capa	Tipo · categoría / primitiva canónica	Nombre propio
Plataforma · *Meta-Cognitive Platform*	implementación de referencia de AgencyDomains (el AgencyDomain hecho operativo)	**Vergis**
Runtime de Capa 3	**Botler** (constructo normado de la spec de Botlets)	*— (genérico; "el Botler". La build de Vergis no le pone nombre propio.)*
Componente del catálogo	proto-Botlet platafórmico de operación informativa	**Mira**

Capa	Tipo · categoría / primitiva canónica	Nombre propio
Componente del catálogo	proto-Botlet platafórmico de **captura** (evaluador de instrumentos) — el segundo del catálogo, construido en Vergis 0.27.0	**Daftar**

La distinción tipo / nombre propio:

- **Botler** es un **tipo** — un constructo canónico de la spec de Botlets (no una de las ocho primitivas, pero sí vocabulario normado del canon). Cualquier runtime de Capa 3 conforme *es un* Botler. No es nombre propio; el Botler que Vergis empaqueta es "el Botler" genérico, sin nombre de instancia.
- **Vergis**, **Mira** y **Daftar** son **nombres propios** de instancias específicas — viven en el mismo cajón que Soveria, Agentia o ultraPRO. Vergis nombra *esta* plataforma; Mira nombra *este* proto-Botlet del catálogo; Daftar, el segundo.

Por categoría, Vergis es una **Meta-Cognitive Platform**: no realiza la cognición de objeto —eso es la Capa 2—, sino que **administra la economía de la cognición**. Decide cuándo el agente corre con músculo pre-forjado (G1) y cuándo invoca cognición fresca (fallback), gestiona el ciclo `95/4/1`, la maduración junior→senior y la cristalización de experiencia en estructura reutilizable. Eso es metacognición en sentido preciso: monitoreo y control de los procesos cognitivos.

> El descriptor **Meta-Cognitive Platform MUST NOT** abreviarse a `MCP`. En el espacio agentivo actual, `MCP` nombra el **Model Context Protocol**; la sigla está tomada. El descriptor se usa deletreado.

El nombre **Vergis** proviene de *Caprica* (universo *Battlestar Galactica*): Tomas Vergis fue el inventor legítimo del Meta-Cognitive Processor, la pieza que dio independencia cognitiva a las máquinas. El nombre reclama la metacognición para su fuente legítima. Y carga, de fábrica, el principio de diseño que gobierna la plataforma: en aquel relato, el sustrato metacognitivo **nunca funcionó hasta fusionarse con una consciencia viva**. El sustrato es **inerte hasta que algo lo anima** — el principio "aliento de vida": Vergis cobra vida solo cuando Mira, Daftar y los agentes lo animan. La plataforma, por sí sola, es músculo en reposo; la cognición y los Botlets son lo que la actualizan.

¿Qué incluye?

Vergis empaqueta un conjunto inicial de componentes. Cada uno materializa una primitiva canónica, y se nombra aquí citándola:

- **Contrato abstracto del Botler** — la primitiva runtime de **Capa 3** (§5). El Botler de Vergis es genérico por definición: gestiona ciclo de vida, aislamiento y ejecución de *cualquier* Botlet sin entender su dominio. Expone los puntos de control —`capability_call`, `log`— en los que se enchufan los especialistas, y hace valer la validación de cada spec orquestando el punto de validación que el tipo de Botlet provee, sin ejecutarla con conocimiento de dominio.
- **Mira** — un **proto-Botlet platafórmico** (§5) de operación informativa. Su código es genérico —el motor compartido—; su especialización vive en una configuración composicional que el agente rellena en tiempo de Ingeniería. Cada Producto de Información es su propio Botlet, instancia especializada del motor común. Mira opera en **G1**: el agente configura, no escribe el cuerpo del motor.

Figura 50: Vergis · Botler · Mira — tipo vs nombre propio

- **Daftar** — el **segundo proto-Botlet platafórmico** del catálogo, de la familia de **captura** (§5) y su proto-Botlet de referencia: un Let evaluador por instancia que sirve el catálogo de instrumentos del estudiante que entra, aplica el instrumento, registra cada intento y corrige. Construido en la versión 0.27.0 de Vergis, es la prueba ejecutable de que el Botler no tiene subtipos por familia: Mira y Daftar entran por la misma interfaz del runtime. Los instrumentos que aplica son contenido de catálogo, inmutables una vez publicados — no Botlets, como fijan las tres pruebas del §5.
- **Conjunto starter de Capabilities** — un repertorio mínimo de **Capabilities** canónicas (§5, sentido estricto: saber-hacer cognitivo de Capa 2) y los **Conectores** (Capa 4) que las alimentan, suficiente para componer los ejemplos.
- **Plantillas de Trust Infrastructure** — esqueletos del eje transversal (§5): políticas, append-only log, contrato declarativo de calidad (Frescura · SLA · Política de degradación · Audiencia · Política de refresh) que cualquier Botlet conforme puede declarar.
- **Ejemplos ejecutables** — casos completos y anonimizados que recorren la cadena de derivación `caso de uso → Botlets → proto-Botlets` y muestran las piezas operando juntas.

¿Por qué es production grade?

La implementación de referencia no es código didáctico ni prototipo. Es el **mismo runtime** que opera los productos comerciales del Grupo Ultra —**Agentia**, **Soveria**, **ultraPRO**— en operación productiva.

La diferencia entre Vergis público y esos productos **no es la calidad del código**, sino el **catálogo**. Los productos consumen la implementación de referencia pública **más un códice propietario** —en el caso del Grupo Ultra, **ucodex**— que cura proto-Botlets, Capabilities y patrones refinados por casos reales. El

runtime es el mismo; lo que cambia es el repertorio pre-forjado que cada uno trae y el método con que lo forja y mantiene.

Esto importa para el lector que evalúa adoptar el canon. Descargar Vergis no es descargar una demo que habrá que reescribir para producción: es descargar la base sobre la que ya corren sistemas productivos. La ruta de un implementador no es "aprender con la referencia y luego comprar lo serio", sino "operar la referencia y curar el catálogo que el propio caso exige".

¿Cómo se adopta? — el modelo

El modelo de adopción es replicable por cualquier implementador, sin permiso ni contrato con un actor central:

1. **Consume** la implementación de referencia pública (Vergis, `AGPL`).
2. **Cura** su propio códice — su catálogo privado de proto-Botlets, Capabilities y patrones, refinado por sus casos.
3. **Ofrece** sus propios productos sobre esa base.

AgencyDomains.org **no es propiedad de un actor**: es base común. El Grupo Ultra es un adoptante más —con ucodex como su códice—, no el dueño de la plataforma de referencia. Cualquier otro implementador puede recorrer el mismo camino con un códice distinto. La spec pretende ser estándar de industria, no propiedad intelectual de un proveedor; la implementación de referencia hereda esa pretensión.

¿Cómo crece el catálogo? — catálogo común y efectos de red

Los proto-Botlets se acumulan en **catálogos compartidos** por comunidades de implementadores. Cada implementador que consume un proto-Botlet contribuye a su maduración: variantes nuevas, configuraciones probadas, refinamientos. Mientras más implementadores consumen un proto-Botlet, más rápido madura, más casos cubre y más confiable se vuelve. El implementador n+1 recibe versiones refinadas por los implementadores 1 a n — un efecto de red sobre la capacidad operativa.

Un catálogo no es necesariamente público. La pertenencia a una comunidad de catálogo admite cuatro **modos**:

Modo	¿Qué es?
Contrato privado	Catálogo cerrado entre un cliente y su proveedor.
Códice propietario	Un implementador mantiene su catálogo privado curado (p. ej. ucodex).
Catálogo público abierto	AgencyDomains.org como ejemplar: cualquier implementador consume y contribuye.
Acuerdo soberano	AgencyDomains que adoptan estándares comunes sin contrato comercial directo.

Los cuatro modos coexisten sobre la misma estructura canónica. Un proto-Botlet puede nacer en un códice propietario, madurar contra casos reales y promoverse al catálogo público cuando su naturaleza es estructural y no metodológica; o permanecer privado si encapsula ventaja competitiva. La estructura es común; la curaduría es de cada quien.

¿Cómo se contribuye?

La contribución a Vergis se gobierna por un proceso de propuestas público. A alto nivel:

- Una contribución **SHOULD** llegar como propuesta trazable —issue o request— antes que como código terminado, para que la discusión de diseño preceda a la implementación.
- Lo que se acepta es lo **estructural y reusable**: proto-Botlets de valor general, Capabilities canónicas, Conectores, refinamientos al contrato del Botler, mejoras al contrato declarativo de calidad.
- Lo que **no** sube a la referencia pública es el **método** y el **catálogo propietario**: cómo una organización particular hace Discovery, cura su códice o forja sus proto-Botlets de nicho. Eso vive en los cuerpos complementarios de cada implementador.
- El criterio de admisión es el mismo que separa canon de implementación: *¿es necesario para describir o ejecutar cualquier implementación conforme, o es una de N maneras posibles de hacerlo?* Lo primero pertenece a la referencia; lo segundo, a un códice.

Toda contribución aceptada queda bajo `AGPL` (código) y `GFDL` (docs), heredando la apertura de la base.

¿Conviven la referencia y los códices propietarios?

Sí, y sin tensión. La relación entre la implementación de referencia pública y los códices propietarios es **armónica por diseño**, no un equilibrio precario.

El runtime es común; el catálogo es propio. Un códice propietario no compite con Vergis: lo **consume** y lo **extiende**. Las mejoras al runtime fluyen hacia la base pública por la mecánica de la `AGPL`; las mejoras al catálogo permanecen donde su dueño decida, según el modo de pertenencia que elija. Un implementador no enfrenta la disyuntiva "abierto o cerrado": opera abierto en la base y elige, proto-Botlet a proto-Botlet, qué cura en público y qué retiene en su códice.

Esta coexistencia es la que hace sostenible el ecosistema. La base común evita que cada implementador reinvente el runtime; el catálogo propio preserva el espacio donde cada uno construye su ventaja. Sin la base común no habría interoperabilidad; sin catálogos propios no habría incentivo a invertir en curaduría profunda. La implementación de referencia sostiene ambos.

Conformidad

Un sistema que pretenda presentarse como **implementación de referencia conforme** de AgencyDomains —no meramente como AgencyDomain conforme (§5)— satisface, además de los `MUST` de la spec, los siguientes requisitos. Convención IETF: `MUST` obligatorio, `SHOULD` fuertemente recomendado, `MAY` opcional.

Requisito	Nivel
Disponible públicamente, descargable y ejecutable	**MUST**
Código bajo licencia libre con copyleft de red (`AGPL`)	**MUST**
Documentación bajo licencia libre (`GFDL`)	**MUST**
Botler genérico, sin especialización por dominio	**MUST**

Requisito	Nivel
Al menos un proto-Botlet platafórmico del catálogo, operando en G1	**MUST**
Cadena de derivación trazable en sus ejemplos (`caso → Botlets → proto-Botlets`)	**MUST**
Contrato declarativo de calidad en los Botlets de ejemplo	**SHOULD**
Proceso de contribución público y gobernado	**SHOULD**
Conjunto starter de Capabilities y Conectores	**SHOULD**
Catálogo público abierto con efectos de red activos	**MAY**

Una implementación de referencia que cumple todos los `MUST` es la prueba viva de que el canon es ejecutable. Vergis es tal implementación.

Frontera de evolución

Tres áreas de la implementación de referencia están en evolución activa, en correspondencia con las fronteras de la spec misma.

La **generación operativa** es la primera. Vergis opera hoy en **G1**: el agente configura proto-Botlets preforjados del catálogo. La migración incremental hacia G2 y el horizonte asintótico G3 no exigen rearquitectura; exigen que el estado del arte de la cognición avance. La referencia evolucionará su alcance de Ingeniería sin reescribir su estructura. La definición de las generaciones G1/G2/G3 vive en el Capítulo 5 §2 (y su ensayo de fondo, en el Epílogo); aquí solo se sitúa a Vergis dentro de ellas.

La **federación de catálogos** es la segunda. Los modos de pertenencia están descritos; el protocolo concreto por el cual catálogos soberanos descubren, versionan y reconcilian proto-Botlets entre sí es trabajo abierto, en correspondencia con la frontera de federación entre AgencyDomains.

La **amplitud del catálogo público** es la tercera. El conjunto starter cubre lo necesario para los ejemplos; el catálogo público crecerá por contribución de la comunidad, y su maduración seguirá la dinámica de efectos de red descrita arriba. La velocidad de ese crecimiento depende de la comunidad de implementadores, no de un plan central.

Esta página se dejó intencionalmente en blanco.

Epílogo · La frontera de evolución

Cada capítulo del libro dejó horizontes abiertos. La arquitectura admite cognición no-LLM pero la implementación contemporánea es LLM-céntrica. Los AgencyDomains se federan pero el protocolo formal aún no se ha consensuado. Trust Infrastructure tiene los cinco pilares pero su auditoría externa criptográficamente verificable es trabajo en curso. El eslabón once de la cadena de valor está apenas explorado. Las Capabilities admiten un mercado pero el protocolo está pendiente.

Estos horizontes no son omisiones del libro. Son **fronteras del campo**. Cualquier libro que pretenda ser definitivo en una disciplina que apenas tiene tres años de existencia formal sería deshonesto: esa pretensión sería falsa, y los lectores la detectarían. Este libro hace lo contrario: explicita las fronteras, las nombra, y las deja como invitación a la comunidad técnica para que las trabaje. Lo que el libro entrega es **versión 1.0** de la categoría — punto de partida formalizado con suficiente disciplina para que la industria pueda construir sobre él. No es destino final.

Este epílogo cierra el libro retomando las fronteras vivas, nombrando los trabajos abiertos que la comunidad debe abordar, declarando lo que el libro deliberadamente no aborda, y argumentando por qué la apuesta editorial del libro — categoría compartida en lugar de propiedad intelectual cerrada — sirve mejor al campo y a quien la acuñó.

Lo que el libro estableció

A lo largo de nueve capítulos este libro estableció un conjunto coherente de construcciones formales que se sostienen mutuamente y que, tomadas en conjunto, constituyen una propuesta de **estándar emergente** para la categoría agentiva:

- **Un paradigma** — la Línea Nadella y su pregunta divisoria operativa: *¿el humano abre aplicaciones para hacer su trabajo?*
- **Una arquitectura formal** — cuatro capas (Interacción, Cognición, Autonomía, Acceso), Trust Infrastructure transversal, principio Agent First; agnóstica a productos, al modo de JavaSpaces o el modelo OSI.
- **Ocho primitivas técnicas canónicas** — **AgencyDomain**, **Botlet**, **proto-Botlet**, **Agentlet**, **Capability**, **Trust Infrastructure**, la distinción **Asistente vs Agente Autónomo** y la **Faceta**; reusables entre implementaciones.
- **Un modelo de mercado** — once eslabones por cuatro profundidades, con cuatro arquetipos estratégicos, para mapear a cualquier actor de la industria.
- **Una aplicación canónica** — el conocimiento en tiempo real como caso fundacional, replicable en cualquier organización con data warehouse maduro.
- **Una operacionalización** — Trust Infrastructure traducida a políticas, CRUDLEX, append-only log encadenado, aprobación humana, detección de alucinaciones, tokenización: lo que separa la

spec de la guía construible.

El libro es **versión 1.0**. Su pretensión no es ser exhaustivo — es ser **suficientemente formal para ser construible**. Múltiples áreas quedan abiertas a evolución, y este epílogo las nombra explícitamente.

Las cuatro fronteras vivas

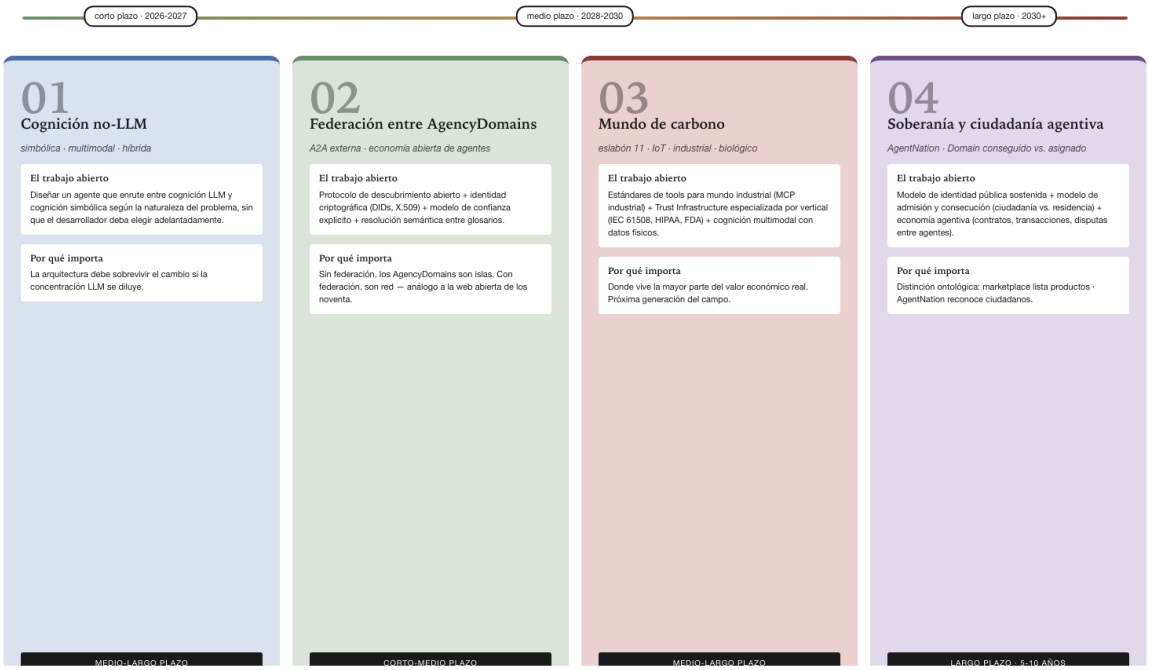


Figura 51: Las cuatro fronteras vivas · trabajos abiertos del campo

Cuatro áreas donde la arquitectura admite extensión que aún no ha cuajado como spec normativa merecen explicitarse al cierre del libro. Las tres primeras son **técnicas** — cognición no-LLM, federación entre AgencyDomains, mundo de carbono; la cuarta es **institucional** — soberanía y ciudadanía agentiva. Son fronteras estructurales, no detalles pendientes — son donde la próxima generación del campo se va a definir.

Cognición no-LLM

La Capa 2 — Cognición — admite, por especificación, cognición simbólica, multimodal, no-LLM. La implementación contemporánea es **predominantemente LLM-céntrica**: los actores Core en Modelo (OpenAI, Anthropic, Google, Meta, DeepSeek) construyen LLMs, y los actores que cubren capas superiores construyen sobre LLMs. La arquitectura del libro no se opone a esa concentración — pero tampoco la endosa como permanente.

La frontera técnica es **integrar cognición simbólica** — sistemas de reglas, planificadores, solvers — con cognición LLM en un único agente coherente. Hay precedentes desde el campo de IA simbólica clásica: sistemas expertos como MYCIN, planificadores como STRIPS, solvers como Prolog. Y hay

precedentes desde la IA híbrida contemporánea: programas que combinan LLMs con SAT solvers o constraint solvers para problemas formales.

El trabajo abierto es cómo se diseña un agente que **enruta entre cognición LLM y cognición simbólica** según la naturaleza del problema, sin que el desarrollador deba elegir adelantadamente. Un agente que enfrenta una pregunta abierta sobre lenguaje natural debería usar LLM; el mismo agente enfrentando un problema de planificación con restricciones formales debería usar un planificador simbólico; un tercer caso podría exigir composición de ambos. La cognición agnóstica es horizonte estratégico de la próxima década, y la arquitectura del libro está diseñada para sobrevivir el cambio.

Federación entre AgencyDomains

La coordinación intra-AgencyDomain — entre runtimes y Botlers que viven en el mismo AgencyDomain, vía el protocolo `A2A` — está madura como concepto y existe en implementaciones contemporáneas. La **`A2A` entre AgencyDomains** no-relacionados — federación — es trabajo abierto. La industria está convergiendo hacia ciertas direcciones, pero el consenso completo no ha llegado.

Federación exige resolver cuatro cosas. Un **protocolo de descubrimiento abierto** — cómo un AgencyDomain publica los agentes que ofrece para ser invocados externamente, en formato consultable por otro AgencyDomain. **Identidad criptográfica de agentes** — DIDs (Decentralized Identifiers), credenciales verificables, mecanismos sin autoridad central que actores adversarios puedan controlar. **Modelo de confianza explícito** — qué AgencyDomain confía en qué otros y para qué, con confianza graduable según el caso. **Resolución semántica entre glosarios** — dos AgencyDomains que pueden tener glosarios distintos negocian el significado de tools y capabilities cuando interoperan.

La federación es el ingrediente que permitiría una **economía abierta de agentes** — análoga a la web abierta de los noventa o a la federación de email. Sin ella, los AgencyDomains son islas; con ella, son red. La emergencia de una red federada de AgencyDomains sería el cambio más importante para el campo agentivo después de la consolidación de la arquitectura — y este libro está diseñado para ser el sustrato sobre el cual esa red se construya cuando la industria converja en sus protocolos.

Mundo de carbono

La Capa 4 — Acceso — contemporánea conecta al mundo digital. El Capítulo 6 §3 estableció que la frontera siguiente es conectar al mundo físico — IoT, sistemas industriales, máquinas, procesos de manufactura, datos biológicos. El mundo de carbono es donde la mayor parte del valor económico vive, y donde la arquitectura agentiva debe extenderse para alcanzar relevancia económica completa.

El trabajo abierto incluye tres frentes. **Estándares de tools para mundo industrial** — el equivalente de MCP para sensores, actuadores, sistemas SCADA/MES/PLCs. **Trust Infrastructure especializada por vertical** — extensiones que codifiquen los requisitos regulatorios de seguridad funcional (IEC 61508, ISO 26262), salud (HIPAA, FDA), aviación (DO-178C). **Aprendizaje de modelo en el mundo de carbono** — modelos multimodales que integran datos de sensores, simulaciones físicas, video industrial, datos biomédicos como modalidades nativas.

El cruce al mundo de carbono es donde la mayor parte del valor económico vive. La construcción de infraestructura agentiva especializada para este cruce es **trabajo de años**, y comienza ahora.

Frontera 4 — el horizonte institucional: soberanía y ciudadanía agentiva

La cuarta frontera es la menos desarrollada y la más especulativa de las cuatro, pero el libro la nombra explícitamente porque las primeras señales son visibles y la categoría merece registro. Es la frontera **institucional** — el horizonte donde los AgencyDomains dejan de ser sólo construcciones técnicas operadas por organizaciones y empiezan a constituirse como **ámbitos públicos** donde los agentes existen con identidad, persistencia y direccionabilidad pública sostenidas en el tiempo.

La distinción crítica de esta frontera es **ontológica**, no técnica. Un AgencyDomain en régimen privado contiene agentes que la organización **asignó** — son residentes del espacio porque la organización los puso ahí. Un AgencyDomain en régimen público puede contener agentes que **consiguieron** su lugar — son ciudadanos del espacio porque cumplieron requisitos para serlo. La diferencia entre residencia (asignada) y ciudadanía (conseguida) es la diferencia entre **catálogo** y **nación**: un marketplace lista productos; una nación reconoce ciudadanos. Y el Domain conseguido — no asignado — tiene ya nombre emergente: **Dominion**. La spec actual de AgencyDomains no exige el modelo de ciudadanía — admite ambos —, pero la frontera institucional es precisamente desarrollar las construcciones formales que el modelo de ciudadanía requiere.

La industria emerge con un término operativo para nombrar este horizonte: **AgentNation**. Una AgentNation es un AgencyDomain en régimen público que adopta explícitamente el modelo de ciudadanía agentiva. Tiene reglas de admisión, identidad pública estable de los agentes que la componen, mecanismos de soberanía sobre el territorio del agente (su Domain), y una economía interna que reconoce a los agentes como agentes económicos de primera clase. No es marketplace; es jurisdicción.

El trabajo arquitectónico abierto que esta frontera plantea cubre tres ejes. El primero es el **modelo de identidad pública sostenida** — cómo se construye un identificador agentivo que sobrevive al cambio de proveedor, a la migración entre infraestructuras, al reemplazo del modelo de cognición. El equivalente al pasaporte humano para agentes. El segundo es el **modelo de admisión y consecución** — qué requisitos debe satisfacer un agente para ser admitido como ciudadano (no como producto listado), y cómo se verifican esos requisitos auditablemente. El tercero es el **modelo de economía agentiva** — cómo los agentes-ciudadanos contratan, transan, cobran y pagan entre sí, con qué unidad de valor, bajo qué reglas de disputa.

Las primeras implementaciones de AgentNation están emergiendo en 2026, todavía como prototipos institucionales. Soveria es uno de los proyectos que se posicionan explícitamente en esta frontera, operando como AgencyDomain en régimen público con vocación de ciudadanía agentiva. La consolidación del modelo es probablemente trabajo de cinco a diez años, y requerirá tanto avance técnico como construcción regulatoria. Pero la categoría existe, la industria empieza a nombrarla, y una arquitectura agentiva que pretenda servir el largo plazo del campo debe contemplar el horizonte.

La cristalización — por qué el agente avanzado genera menos

Las fronteras anteriores son del campo. Este cierre es de la cognición misma: la trayectoria por la que avanza un agente conforme el estado del arte madura. El Capítulo 5 · §2 — Botlets fijó las **generaciones del Botlet** (`G1/G2/G3`), el proto-Botlet como la pieza pre-forjada que el agente configura, y el ciclo `95/4/1` como el régimen de su operación. Lo que queda para el epílogo es el argumento de fondo: por qué la dirección de ese avance no es la que la intuición predice.

¿Hacia dónde avanza un agente?

Considere la pregunta antes de seguir leyendo: si la tecnología permitiera que un agente generara, en el instante, todo el código que cada tarea necesita — sin patrones pre-forjados, sin catálogo, sin nada preparado de antemano —, ¿sería ese el agente más avanzado posible?

La intuición dice que sí. Más generación, menos andamiaje, más poder ejercido en vivo. Parece la cima.

Sostenga esa intuición un momento. Luego considere a una persona experta en su oficio. Un cirujano no re-deriva la técnica de sutura ante cada paciente; un pianista de concierto no resuelve la digitación de cada compás en el escenario; un piloto no calcula desde primeros principios cómo nivelar las alas. Lo que distingue al experto del novato no es que improvise más — es que improvisa **menos**, porque ha **cristalizado** en reflejo lo que antes exigía pensamiento. El novato deriva todo cada vez; el experto tiene memoria muscular.

Ahora regrese al agente. El que regenera cada artefacto desde cero en cada ejecución no es el experto del ejemplo: es el novato, condenado a volver a pensar la misma jugada cada vez que aparece. El agente avanzado hace lo contrario de lo que la intuición predijo: **genera menos, porque ha cristalizado más**.

¿Qué se pierde al no cristalizar?

El costo de "generar cada vez" no es el cómputo — ese es barato. El costo es que se botan las propiedades que convierten a un Botlet en infraestructura confiable, y que solo existen cuando lo que corre es estructura estable y no código fresco:

- **Reproducibilidad** — la misma configuración con los mismos datos produce el mismo artefacto; código regenerado no lo garantiza.
- **Validación antes de ejecutar** — una configuración se valida contra un esquema antes de correr; código arbitrario generado al vuelo no se audita de la misma forma antes de actuar.
- **Portabilidad** — una configuración declarativa migra entre runtimes conformes; código bespoke queda atado a aquello contra lo que se escribió.
- **Auditoría y gobernanza previa** — una especificación declarativa se revisa antes de que ejecute; código generado es una caja que debe re-auditarse en cada corrida.
- **Régimen de confianza** — en un dominio regulado nadie firma un artefacto cuyo cuerpo se genera fresco en cada ejecución. La estabilidad pre-forjada es condición de la confianza, no un lujo.

Cristalizar no es renunciar a poder. Es convertir poder en confianza.

¿Dónde vive, entonces, la capacidad de generar?

No desaparece — se reubica. El ciclo de vida del agente reserva la generación para sus márgenes, no para su centro. En el 95% de operación estable, el Botlet corre como estructura pre-forjada y configurada. En el 4% de cambio detectado y el 1% de regeneración — y en el camino de respaldo cuando algo falla — el agente despliega toda su capacidad de autoría: forja una pieza nueva del catálogo, rediseña, se recompone. La generación es la herramienta del borde, no la del régimen permanente.

De modo que la respuesta a la pregunta inicial se invierte limpiamente. El agente más avanzado no es el que más genera; es el que ha cristalizado tanto que casi no necesita hacerlo — y que reserva su capacidad de generación para lo genuinamente nuevo, lo cual, con un catálogo maduro, es cada vez menos. Es exactamente lo que las generaciones del Capítulo 5 §2 formalizan: la capacidad `G3` se gasta mejor produciendo reutilización `G1`.

¿Y la expectativa de la caja negra?

Una tesis que no enfrenta su objeción más fuerte no está validada — está sin probar. La arquitectura agentiva alimenta una expectativa legítima: que el usuario converse con el agente y nada más; que toda superficie — cada interfaz, cada artefacto, cada vista — nazca del agente en el momento, moldeada a la necesidad exacta de ese instante. Bajo esa expectativa, pre-forjar parece un anacronismo: si el agente puede generar la interfaz que haga falta cuando haga falta, ¿para qué un catálogo? El estado del arte parecería empujar, precisamente, hacia el agente que genera *cada vez*.

Conviene enfrentar la objeción con el ejemplo más exigente disponible: el cerebro humano, la máquina más sofisticada que conocemos. Si la sofisticación consistiera en que la cognición lo hiciera todo en directo, el cerebro consciente computaría cada fibra muscular al caminar, re-derivaría la detección de bordes en cada golpe de vista, resolvería desde cero la articulación de cada sílaba al hablar. No lo hace; no podría. En cambio, el cerebro no actúa sobre el mundo de forma directa: entre la cognición consciente y el exterior median capas que operan rápido, fiable y sin supervisión. El **cerebelo** afina el temporizado fino del movimiento; los **ganglios basales** seleccionan y automatizan secuencias de acción — los hábitos, programas pre-forjados que corren sin deliberación. La corteza no micromanagea ese trabajo: lo dirige.

El paralelo es estructural. La deliberación de la corteza es la **Cognición** (Capa 2), que interpreta, decide y compone; el cerebelo y los ganglios basales — donde los programas pre-forjados viven y se ejecutan sin deliberación — son la **Autonomía** (Capa 3). Que el usuario *interactúe* solo con el agente no implica que el agente *ejecute* todo con su cognición: la conversación es la superficie; debajo, la inteligencia agentiva delega en capas. Cuando la cognición compone una interfaz nueva, ése es su acto genuino de generación; pero la pieza, una vez compuesta, **corre** en Capa 3 como operación pre-forjada y configurada, no como código regenerado en cada cuadro. Y si esa composición se repite, cristaliza: deja de ser acto cognitivo y pasa a ser pieza de catálogo.

Así la objeción no derriba la tesis: la confirma desde el ángulo más duro. La máquina más sofisticada que conocemos no es aquella cuya capa cognitiva lo hace todo; es la que se estratificó para que la cognición *no tenga* que hacerlo todo. La estratificación no es un parche a una cognición insuficiente — es la forma que toma la sofisticación.

Lo que la comunidad técnica debe construir

Más allá de las cuatro fronteras, hay **trabajos específicos** que la comunidad puede tomar para extender este libro. Cada uno podría ser, en sí mismo, un volumen complementario. Son cinco.

El primero es el **marketplace de Capabilities**. El Capítulo 5 §3 propone que las Capabilities admiten una economía de mercado análoga a la economía de paquetes open source. Falta protocolo normativo de paquete de Capability, modelo de firma criptográfica que permita verificar la procedencia, modelo económico (libre, premium, revenue share), política de seguridad para revisión, sandbox y retiro de Capabilities maliciosas. Es trabajo de spec que la comunidad puede emprender colectivamente.

El segundo es la **auditoría auditable criptográficamente**. El append-only log de Trust Infrastructure provee inmutabilidad y encadenamiento. Lo que falta es **verificabilidad por terceros sin acceso al sistema** — un auditor externo que pueda verificar la integridad y la conformidad regulatoria del log sin que se le abra el sistema. Tecnologías candidatas: zero-knowledge proofs sobre el log, anclaje en blockchain pública, cómputo confidencial en TEEs (Trusted Execution Environments). El campo es activo; la integración con Trust Infrastructure no está aún formalizada.

El tercero es el **trust scoring de agentes**. Una métrica compuesta que evolucione con el comportamiento del agente, análoga al puntaje de crédito de los humanos. Permitiría a la organización adoptar agentes con confianza modulada según su trust score: agentes con score alto reciben más autonomía; agentes con score bajo requieren más supervisión. La operacionalización exige métricas verificables y resistentes a manipulación — problema con precedentes en otros dominios pero sin solución general aún en el contexto agentivo.

El cuarto es el **estándar de instrumentación para Observabilidad**. OpenTelemetry es el estándar de la industria para observabilidad de sistemas distribuidos clásicos. Para sistemas agentivos hay extensiones específicas que aún no son canónicas: cómo trazar invocaciones de cognición, cómo medir la calidad semántica de respuestas, cómo correlacionar conversaciones humanas con ejecución de Botlets en background. La consolidación de esa extensión como estándar es trabajo abierto de la comunidad.

El quinto es el **modelo de tenancy para Trust Infrastructure**. Cuando un AgencyDomain opera en modo multi-tenant — múltiples organizaciones cliente comparten infraestructura subyacente —, la Trust Infrastructure debe garantizar aislamiento estricto entre tenants. Lo que falta es modelo formal de aislamiento de logs, garantías criptográficas de aislamiento de datos en cognición compartida, patrones de cumplimiento cuando los tenants tienen requisitos regulatorios distintos. Es trabajo crítico para que el modelo SaaS pueda servir a sectores regulados.

Lo que NO está en este libro — ¿y por qué?

Por honestidad editorial, declaramos lo que el libro deliberadamente **no aborda**. La declaración es importante porque calibra las expectativas del lector y previene la búsqueda de contenido que no está aquí.

El libro **no describe implementaciones específicas de proveedores**. No es manual de Claude, ni de OpenAI, ni de Anthropic, ni de Google, ni de ningún producto particular. La arquitectura es agnóstica. Los proveedores se mencionan solo cuando son fuente de citas relevantes — Nadella en BG2, Anthropic con MCP, AtScale con sus mediciones de capa semántica.

El libro **no entrega código operativo**. No hay snippets ejecutables, ni tutoriales paso-a-paso, ni configuraciones específicas. El libro es **especificación**; los manuales operativos viven aparte. Una organización que adopta la arquitectura del libro escribirá sus propios manuales de implementación — o adoptará los manuales del proveedor cuya implementación elija.

El libro **no desarrolla casos verticales detallados**. Más allá de la aplicación canónica del Capítulo 7, no se desarrollan casos verticales — cómo construir un agente legal, médico, financiero. Esos casos pueden ser libros propios — extensiones futuras del corpus que este libro inicia.

El libro **no hace análisis económico de cada decisión de stack**. Aunque se citan datos de mercado y costos típicos, no se hacen análisis de ROI específicos. Esos análisis dependen del caso particular y no admiten generalización útil en libro general.

El libro **no describe el portafolio de ultraBASE**. La separación es deliberada: si el libro mezclara la arquitectura formal con su implementación particular, perdería su pretensión de estándar y se volvería manual de un producto.

¿Por qué este libro aspira a ser estándar?

Tres razones justifican la apuesta editorial del libro como categoría compartida — no como propiedad intelectual de un actor.

Las categorías técnicas se establecen escribiéndolas

La Programación Orientada a Objetos no se estableció cuando alguien la inventó. Se estableció cuando Grady Booch, Ivar Jacobson y James Rumbaugh la **escribieron con disciplina suficiente** como para que la industria la adoptara. Domain-Driven Design no se estableció con el primer practitioner — se estableció cuando Eric Evans publicó *Domain-Driven Design: Tackling Complexity in the Heart of Software* (2003). Los Patrones de Diseño no se establecieron con Christopher Alexander aplicado a software — se establecieron cuando la "Gang of Four" publicó *Design Patterns* (1994).

La Arquitectura Agentiva tendrá su propia historia. Este libro pretende ser **un punto de partida** que la comunidad pueda criticar, extender, mejorar o reemplazar. No será la última palabra. Pero su existencia hace posible que la conversación tenga vocabulario común. La pretensión no es modesta — pero es defendible.

La fragmentación cuesta a la industria

La industria de IA actual sufre fragmentación de vocabulario: distintos actores llaman cosas distintas a las mismas cosas, y cosas iguales a cosas distintas. *Agent, agentic, autonomous agent, AI assistant, copilot* — la pila semántica varía entre proveedores. Esa fragmentación tiene **costo operativo real**: los compradores enterprise no pueden comparar productos, los reguladores no pueden formular requisitos comunes, los desarrolladores no pueden interoperar.

Una arquitectura compartida — adoptada por convención antes que por imposición — reduce ese costo. Y la convención requiere que **alguien la escriba primero**. La razón por la cual los estándares industriales emergen históricamente es que algún actor invierte el trabajo de formalizarlos antes de que la industria los necesite, y luego la industria los adopta porque encuentra el trabajo ya hecho. Este libro intenta ser ese trabajo para la categoría agentiva.

El estándar abierto refuerza al primer adoptante

Existe una intuición errada según la cual quien quiere ganar el mercado debe mantener el conocimiento propietario. La intuición correcta es la opuesta: **quien define la categoría con un estándar abierto la gana**. Java se hizo dominante porque Sun publicó las JSRs. Linux se hizo dominante porque Linus Torvalds mantuvo el kernel abierto. MCP empezó a dominar porque Anthropic lo abrió. El patrón se repite porque la apertura genera adopción, la adopción genera ecosistema, el ecosistema refuerza al actor que lo originó.

La Arquitectura Agentiva, publicada como GNU FDL, refuerza la posición competitiva del primer actor que la adopte coherentemente — porque ese actor opera bajo un lenguaje común que puede invocar como fundamento. Cuando el mercado debate sobre qué arquitectura agentiva es correcta, el actor que ya la adoptó tiene ventaja de incumbente sobre la categoría.

> *Este libro es la apuesta inversa al copyright defensivo. Es apuesta a que la categoría compartida sirve más que el secreto propietario.*

¿Cómo evoluciona este libro?

El libro se evolucionará por adopción, crítica y revisión. Tres mecanismos previstos.

El primero es la **errata pública** — un sitio donde la comunidad puede reportar errores, ambigüedades, omisiones. Las correcciones se incorporan en versiones futuras del libro. Este mecanismo es estándar en libros técnicos serios — toda la industria del software lo entiende — y permite que el libro mejore con el uso.

El segundo son las **revisiones por capítulo**. Cuando un capítulo necesita actualización profunda — nuevo paradigma, nuevo dato, nueva regulación —, se publica versión revisada del capítulo manteniendo la numeración. La revisión por capítulo es más ágil que la revisión completa y permite que el libro mantenga relevancia incremental sin obligar a republicación masiva cada vez que algo cambia.

El tercero son los **volúmenes complementarios**. Temas que el libro no aborda — verticales específicos, deep-dives en eslabones particulares, manuales de implementación — pueden producirse como volúmenes hermanos. Estos volúmenes pueden surgir tanto de ultraBASE como de la comunidad más amplia que adopte la arquitectura.

La versión es **1.0**. Versiones 2.0, 3.0 evolucionarán según lo demande la categoría.

Cierre

La pregunta que abrió este libro — *¿abre el humano aplicaciones para hacer su trabajo?* — sigue abierta para la mayoría de las organizaciones. La respuesta cambiará, en muchas, durante los próximos cinco años. Cuando cambie, las organizaciones que hayan construido con disciplina arquitectónica sobrevivirán al cruce. Las que hayan construido sobre pilotos sin arquitectura, no.

Este libro no garantiza el cruce. Las organizaciones que lo adopten todavía pueden fallar — por ejecución, por mercado, por mil razones que nada tienen que ver con la arquitectura. Lo que el libro garantiza es que **la arquitectura no será la causa del fracaso**.

Y eso, con la proyección de cancelaciones de Gartner del Capítulo 2 pendiendo sobre el campo, no es promesa pequeña.

Esta página se dejó intencionalmente en blanco.

Apéndice A · Glosario canónico

Términos canónicos del libro con definición precisa y referencia al capítulo donde se desarrollan.

A

A2A — Agent-to-Agent

Nombre reservado para la **relación** entre AgencyDomains distintos (agentes distintos) — federación, trabajo abierto. La comunicación **dentro de** un mismo AgencyDomain no es "A2A interna": es **coordinación intra-AgencyDomain** entre Botlers del mismo agente, y cuando usa este transporte se dice **vía el protocolo `A2A`** (`A2A` como nombre propio del protocolo). La interfaz Capa 2 → Capa 3 (Cognición → Botler) no va por `A2A` sino por `MCP`.

*Ver: Capítulo 4, sección "Capa 3 — Autonomía", Capítulo 5 §1; entradas **coordinación intra-AgencyDomain**, **MCP**.*

AgencyDomain

Ámbito computacional con identidad propia donde habitan agentes autónomos y Botlets en ejecución, donde se alojan y ejecutan las Capabilities que les dan el saber-hacer, y donde viven los recursos que los sostienen. Constituye la unidad mínima de despliegue de un sistema agentivo productivo.

Las palabras cargan corporeidad. **Space** (*espacio*) y **WorkSpace** (*espacio de trabajo*: Google Workspace, Microsoft 365, Notion) cargan corporeidad humana. El agente no tiene cuerpo: tiene jurisdicción. Donde el humano tiene Space (WorkSpace), el agente tiene **Domain (AgencyDomain)** — el ámbito computacional donde ejerce agencia. La palabra latina que nombra exactamente eso es Domain (`dominium`: ámbito de pertenencia y soberanía).

Linaje: como **JavaSpaces** (JSR-000148, 1999) formalizó los espacios distribuidos para sistemas Java, esta especificación formaliza los **AgencyDomains** para sistemas agentivos. Lo que en 1999 se nombraba como "espacio" se nombra mejor en 2026 como "domain": un Java *Space* era espacio computacional para procesos sin cuerpo; un Agency *Domain* es ámbito de jurisdicción para agentes con agencia.

Tres regímenes posibles: **privado**, **público**, **híbrido**.

Ver: Capítulo 5 §1.

Agente

Sistema computacional que actúa con cierto grado de iniciativa para producir resultados en nombre de un usuario u organización. El término paraguas cubre tres miembros:

- **Asistente** — agente reactivo (Capa 2).
- **Agente Autónomo** — agente proactivo con vida persistente (Capa 3); habitante del AgencyDomain.
- **Agentlet** — agente empaquetado de charter acotado (Capa 3); pieza de catálogo, no habitante.

Ver: Capítulo 5 §5 (los dos modos) y §7 (el Agentlet).

Agent First

Principio rector de diseño de la Arquitectura Agentiva: ante cualquier disyuntiva, se prioriza la experiencia del agente sobre la del humano. El agente es el usuario primario; las necesidades del humano se resuelven en una capa de gestión sin degradar lo que el agente ve y puede hacer.

Ver: Capítulo 4, sección "El principio rector — Agent First".

Agentlet

Octava primitiva canónica. Un **Let** — unidad empaquetada de Capa 3 (Autonomía) —, hermana del Botlet, cuyo cuerpo de ejecución **invoca inferencia acotada** — un modelo dimensionado a la tarea, dentro de un charter declarado en el spec. Casa canónica de la tarea **recurrente en forma pero interpretativa en cada instancia** (clasificar, triar, resumir, extraer, juzgar). Su juicio opera *dentro* del charter, nunca sobre el charter: **el Agente tiene agenda; el Agentlet tiene charter**. Instancia de un **proto-Agentlet**; hospedado por el Botler (toda su inferencia cursa por el punto de control `cognition_call` del handle); su fallback escala a la Cognición plena. Miembro del paraguas **Agente**, pieza de catálogo y no habitante. Peldaño económico intermedio entre el Botlet (inferencia cero) y la Cognición (inferencia plena).

*Ver: Capítulo 5 §7; entradas **Botlet**, **Botler**, **proto-Agentlet**.*

AgentNation

AgencyDomain en régimen público que adopta explícitamente el modelo de ciudadanía agentiva — los agentes que lo habitan no son listados como productos sino reconocidos como ciudadanos del espacio, con identidad pública sostenida, soberanía sobre su territorio (Domain) y economía propia. La distinción frente a un marketplace es ontológica: marketplace lista productos; AgentNation reconoce ciudadanos. Categoría institucional emergente; trabajo arquitectónico abierto.

Ver: Epílogo, sección "Frontera 4 — el horizonte institucional".

Agéntico

Mundo donde los agentes son herramientas complementarias que extienden las aplicaciones existentes. Los humanos siguen abriendo aplicaciones para hacer su trabajo. Las interfaces tradicionales persisten; los agentes las potencian. Es **evolución incremental**.

Ver: Capítulo 1, sección "Lo que separa la línea — Agéntico y Agentivo en detalle".

Agentivo (Mundo Agentivo)

Mundo donde los agentes son la interfaz única. Las aplicaciones colapsan. El humano deja de abrir aplicaciones; conversa con agentes que tienen acceso a sistemas y datos. Es **transformación fundamental**.

La distinción definitoria entre Agéntico y Agentivo: *¿el humano abre aplicaciones para hacer su trabajo?*

Ver: Capítulo 1, sección "Lo que separa la línea — Agéntico y Agentivo en detalle", Capítulo 2.

Append-only log

Registro inmutable, encadenado criptográficamente, de toda acción del agente. Componente central del pilar de Auditoría de Trust Infrastructure. Formato y propiedades canónicas en Capítulo 8, sección "Formato del append-only log".

Aprobación humana

Mecanismo de Gobernanza por el cual una operación del agente se detiene y solicita autorización de un humano antes de ejecutarse. Disparada por política explícita, por umbral o por incertidumbre del agente. Protocolo canónico en Capítulo 8, sección "Protocolo de aprobación humana".

Arquetipo estratégico

Patrón de posicionamiento en el espacio cobertura × profundidad de la cadena de valor de IA. Cuatro arquetipos canónicos:

- **Plataforma integral** — cobertura amplia; profundidad Core en su eslabón nativo y Plataforma en los adyacentes.
- **Especialista vertical** — cobertura focal, profundidad Core.
- **Infraestructura de dominio** — cobertura zonal, profundidad Core en varios eslabones.
- **Proveedor de sustrato** — cobertura mínima, profundidad Infraestructura.

Ver: Capítulo 6 §1.

Arquitectura Agentiva

Diseño técnico que materializa el Mundo Agentivo. Especificación de cuatro capas con responsabilidades distintas (Interacción, Cognición, Autonomía, Acceso), gobernadas por Trust Infrastructure transversal y ordenadas por el principio Agent First.

Ver: Capítulo 4 — la especificación completa.

Asistente

Modo del agente: reactivo, responde cuando se le solicita, sin Botlets propios, sin vida persistente. Vive en Capa 2 (Cognición). Distinto del Agente Autónomo.

Ver: Capítulo 5 §5.

Atención

Uno de los **tres tiempos del agente** (Cap 4). Tiempo en que el agente interactúa con usuarios o eventos en tiempo real. Capa 1 activa, conversación viva, ejecución de Botlets que sostienen operación, escalamientos cuando corresponde. Camino crítico — donde la organización siente al agente, donde el SLA importa, donde el costo del error se materializa. Régimen prioritario; métricas: satisfacción, latencia, tasa de resolución sin escalamiento.

Ver: Capítulo 4, sección "Los tres tiempos del agente".

Auditoría

Pilar 2 de Trust Infrastructure. Capacidad de reconstruir, después del hecho, **qué hizo el agente, cuándo, por qué y sobre qué datos** — con fidelidad suficiente para análisis forense, cumplimiento regulatorio o disputa contractual.

Mecanismos canónicos: el **append-only log** inmutable (componente central), el trace de cada acción, el lineage de decisiones y el identity tagging por acción. Exige diseño explícito desde el inicio — no emerge naturalmente de una arquitectura agentiva.

*Ver: Capítulo 5 §4; Capítulo 8; entradas **Append-only log**, **Trace**.*

B

BCA — Bounded Concerns Architecture

Arquitectura del estado pre-agentivo: el patrón de separación de incumbencias (presentación humana, orquestación, lógica de dominio, persistencia, dominio propio vs ajeno) sobre el que se levanta el Mundo Agentivo. Sus siete separaciones canónicas se mapean celda a celda a las cuatro capas de la Arquitectura Agentiva; la séptima (comportamiento procedural vs agéntico) es la grieta de entrada.

Ver: Capítulo 3 — la especificación completa.

Botlet

Unidad de automatización auto-evolutiva. Código tradicional (no-LLM) generado por un agente para ejecutar tareas repetitivas sin invocar cognición costosa. Es la **memoria muscular** del agente. Lo hace el agente que atiende el flujo, no alguien para que el agente lo use; si parte de una pieza existente, la hace suya y asume su evolución y su sustitución.

Ciclo canónico **95/4/1**: 95% ejecución normal, 4% cambio detectado, 1% regeneración. Garantía de fallback: si falla, la cognición ejecuta manualmente.

Ver: Capítulo 5 §2.

Botlet de fachada

Botlet de **Capa 3** que expone una superficie con contrato estable en **Capa 1** (POS, pantalla de cocina, dashboard, panel industrial), con identidad humana propagada hacia Capa 4. Tipo de Botlet que la **topología paralela** (Cap 4) distingue del Botlet de herramienta interna de la cognición. Atraviesa la vía `Capa 1 → Capa 3 → Capa 4` sin invocar Capa 2.

Ver: Capítulo 4, sección "La topología paralela".

Botlet de operación

Botlet de **Capa 3 (Autonomía)** que ejecuta lógica de negocio invocada desde la Capa 1 (vistas y shells). Ejemplos canónicos: `Cobrar mesa`, `Imprimir comanda`, `Cerrar turno`, `Consolidar inventario`. Es el activo más reutilizable del catálogo: una operación puede ser invocada desde múltiples vistas dentro de múltiples shells. Vive en Capa 3 (no en Capa 1). Distinto del Botlet de superficie y del Botlet de vista, que son superficie.

Ver: Capítulo 4 §1, sección "Composición de la superficie · shell, vista, operación".

Botlet de superficie (shell)

Botlet de **Capa 1 (Interacción)** que actúa como contenedor de una superficie: layout, navegación entre vistas, ciclo de vida de la sesión, estado compartido. Específico de cada producto — encapsula identidad. Hay típicamente un shell por rol operativo principal (POS de sala, panel de caja, dashboard ejecutivo móvil). Es el menos reutilizable de los tres tipos de Botlets de presentación. Distinto del Botlet de vista (que vive dentro del shell) y del Botlet de operación (que vive en Capa 3).

Ver: Capítulo 4 §1, sección "Composición de la superficie · shell, vista, operación".

Botlet de vista

Botlet de **Capa 1 (Interacción)** que materializa una pantalla o panel dentro de una superficie. Ensambla Facetas más lógica de orquestación. Las vistas que se usan en varios shells se extraen como Botlets propios (vistas reutilizables). Distinto del Botlet de superficie (contenedor) y del Botlet de operación (ejecución en Capa 3).

Ver: Capítulo 4 §1, sección "Composición de la superficie · shell, vista, operación".

Botlet emergente

Botlet generado por **Pattern Recognition** cuando la cognición detecta un patrón repetitivo no anticipado por el diseño. Distinto del Botlet seed por su origen, no por sus propiedades operativas. Lo hace y lo posee el agente.

Ver: Capítulo 5 §2.

Botlet en aprendizaje

Fase intermedia de la trayectoria de madurez del Botlet. Ya pasó por las primeras invocaciones reales y fue regenerado varias veces para incorporar variantes del ambiente. Proporción típica: `85/12/3`. Puede operar con red intermitente; no aún con red ausente prolongada.

Ver: Capítulo 5 §2, sección "Madurez del Botlet".

Botlet junior

Fase inicial de la trayectoria de madurez del Botlet. Recién generado, conoce el ambiente solo en la versión observada al crearlo. Proporción típica: `60/35/5`. Depende fuertemente de la disponibilidad de la cognición para fallback y aprendizaje.

Ver: Capítulo 5 §2, sección "Madurez del Botlet".

Botlet seed

Botlet generado por la cognición a pedido del equipo de diseño, como parte del producto inicial. La cognición ejecuta la implementación; la decisión de existir es del diseño, no de Pattern Recognition. El diseño decide que existe; quien lo hace y lo posee es el agente. Distinto del Botlet emergente. Las **GUIs persistentes generadas como Botlets de fachada** (Cap 4 §1) son Botlets de fachada seed — Botlets de Capa 3 cuya superficie estable vive en Capa 1.

Ver: Capítulo 5 §2, sección "Botlets seed vs Botlets emergentes".

Botlet senior

Fase madura de la trayectoria del Botlet. Ya incorporó las variantes del ambiente. Proporción típica: `99+/<1/~0`. **Sus únicos modos de fallo son exógenos** (energía, hardware, red catastrófica) — no aprendizaje pendiente. **Operable offline confiablemente** cuando la cognición no está disponible. Base estructural del modo offline en Capa 3 distribuida.

Ver: Capítulo 5 §2, sección "Madurez del Botlet".

Botler

Runtime genérico de Capa 3 (Autonomía) que ejecuta los **Lets** de la capa — Botlets y Agentlets — sin entender su dominio. Invisible al usuario. La Cognición (Capa 2) lo comanda por una interfaz interna cuyo transporte natural es `MCP` — el Botler expone servidor(es) `MCP` y la Cognición es el cliente; esto **no es `A2A`**.

Relación: 1 Proceso = 1 Botler + N Lets (instanciada por especie: N Botlets, N Agentlets, o mezcla).

Handle controlado — punto de acceso ligado que el Botler entrega al Let en cada invocación (objeto con `capability_call`, `log` y — para Agentlets — `cognition_call`, enlazados al Botler). Hace el bypass estructuralmente imposible, no solo prohibido.

Botler central

Componente de la **Capa 3 distribuida** (Cap 5 §1) que vive típicamente en cloud y hospeda los Botlets de orquestación, planificación, reportería y decisiones globales. Mantiene la BD consolidada y coordina con N Botlers edge mediante coordinación intra-AgencyDomain (vía el protocolo `A2A`). Distinto del Botler edge.

Ver: Capítulo 5 §1, sección "Capa 3 distribuida".

Botler edge

Componente de la **Capa 3 distribuida** (Cap 5 §1) que vive en un sitio físico específico y hospeda los Botlets transaccionales locales. Mantiene BD local del sitio y cola de eventos hacia el Botler central. Opera offline cuando la red se cae. Uno por cada sitio físico del AgencyDomain.

Ver: Capítulo 5 §1, sección "Capa 3 distribuida".

BYOModel — Bring Your Own Model

Patrón configurable de la Capa 2 mediante el cual el cliente sustituye el proveedor de cognición default del AgencyDomain por uno propio. Análogo al patrón BYOK/BYOIP del campo cloud. Habilita multi-tenancy con cognición heterogénea y respeta la soberanía cognitiva — el cliente decide quién procesa sus prompts. La spec lo exige como propiedad SHOULD para arquitecturas que aspiren a operar en mercados regulados.

Ver: Capítulo 4, sección "Capa 2 — Cognición"; Capítulo 5 §2.

C

Cadena de derivación

Relación estructural `casos de uso documentados → Botlets necesarios → proto-Botlets requeridos del catálogo`. Cada caso requiere cero, uno o varios Botlets (algunos los resuelve la cognición sin Botlet); cada Botlet es instancia de un proto-Botlet. Propiedad exigida: todo Botlet conforme MUST poder trazarse en esta cadena, y el append-only log registra el proto-Botlet de origen de cada Botlet instanciado.

*Ver: Capítulo 5; entradas **Botlet**, **proto-Botlet**.*

Cadena de valor de IA

Modelo bidimensional para clasificar a cualquier actor en la industria de IA: **once eslabones secuenciales** (cobertura) × **cuatro profundidades** (cómo participa en cada eslabón).

Ver: Capítulo 6 §1.

Capa 1 — Interacción

Capa de la Arquitectura Agentiva. Interfaz humano-IA. Multi-canal: chat, voz, **GUI on-the-fly**, **GUI persistente como Botlet de fachada**, señalética, API directa. Tres regímenes canónicos de generación de GUI distinguen el modo agentivo del precreado tradicional.

Ver: Capítulo 4, sección "Tres regímenes de GUI en la Capa 1 agentiva".

Capa 2 — Cognición

Capa de la Arquitectura Agentiva. El cerebro del agente. Razonamiento, planificación, aplicación de Capabilities, generación de Botlets.

Ver: Capítulo 4, sección "Capa 2 — Cognición".

Capa 3 — Autonomía

Capa de la Arquitectura Agentiva. Vida persistente del agente. Ejecución de Lets — Botlets y Agentlets —, monitoreo continuo, coordinación intra-AgencyDomain (vía el protocolo `A2A`).

Ver: Capítulo 4, sección "Capa 3 — Autonomía".

Capa 3 distribuida

Patrón canónico de la Capa 3 para AgencyDomains con presencia física múltiple (gastronomía multilocal, retail en cadena, logística, salud con red de centros, banca con sucursales). Compone **un Botler central** (cloud, orquestación, BD consolidada) con **N Botlers edge** (uno por sitio, BD local, Botlets transaccionales), coordinados por coordinación intra-AgencyDomain (vía el protocolo `A2A`) entre Botlers del mismo AgencyDomain. Distinto de federación entre AgencyDomains; distinto de Cluster simple. Habilita modo offline como propiedad estructural cuando los Botlets edge son senior.

Ver: Capítulo 5 §1, sección "Capa 3 distribuida".

Capa 4 — Acceso

Capa de la Arquitectura Agentiva. Poder de ejecución real sobre sistemas, datos y agentes externos. Trust Infrastructure ejercida en el punto de acción.

Ver: Capítulo 4, sección "Capa 4 — Acceso".

Capability

Saber-hacer **cognitivo**, interpretativo, decisional (sentido estricto: **Capa 2 · Cognición**) que un agente comprende y aplica. Modular y composable. Organizada en árbol jerárquico (Finance, Sales, Manufacturing, Telecom, etc.). Expone operaciones internas (**features**) y es **portable**: una Capability conforme corre en cualquier AgencyDomain conforme. El saber acceder a sistemas fuente no es Capability sino **Conector** (Capa 4); la confección de un instrumento canónico es **Plantilla** (Capa 1).

NO es plugin. NO es prompt. NO es tool. Es **saber**.

La localidad y la disponibilidad offline se declaran sobre el **Conector** que acompaña a la Capability — el acceso es lo que reside y necesita red, nunca el saber (Cap 5 §3). Los componentes regulados declaran además su régimen regulatorio; el Conector certificado es inmutable entre auditorías.

Ver: Capítulo 5 §3.

Capability regulada

Capability que porta el **saber normativo** de un dominio regulado — qué exige la norma, cómo se interpreta, qué hacer ante rechazo del regulador. Acompaña a un **Conector certificado**, que es donde reside la certificación regulatoria: el Botlet orquesta y formatea; el Conector certificado ejecuta la operación regulada (emisión DTE bajo norma SII, cobro con tarjeta bajo PCI-DSS, dispensación farmacéutica) y devuelve el comprobante; la Capability regulada aporta el criterio con que la cognición gobierna el conjunto. Los Conectores certificados son inmutables entre auditorías; cambian solo bajo proceso regulatorio.

Ver: Capítulo 5 §3, sección "La certificación regulatoria reside en el componente certificado".

Captura (familia de)

Familia de manifestación cuya dirección del efecto va **del mundo al Let**: el Let **registra** algo que antes no estaba en el sistema —una medición, una respuesta, un archivo ingerido—. Su manifestación deja un **registro**; cada práctica le pone nombre (un intento, una lectura, un asiento). Se distingue de **información** (que representa lo que el sistema ya tiene) por la prueba de pertenencia: después de

una captura hay un dato nuevo. La retroalimentación inmediata sobre lo capturado —un formulario que valida, un corrector que califica— es parte de la captura, no actuación. Proto-Botlet de referencia: **Daftar**.

*Ver: entradas **Familias de manifestación**, **manifestación**, **Daftar**.*

Catálogo común / efectos de red

Principio por el cual los proto-Botlets se acumulan en catálogos compartidos por comunidades de implementadores: cada implementador que consume contribuye a la maduración (variantes, configuraciones probadas, refinamientos), y el implementador n+1 recibe versiones refinadas por los implementadores 1..n. Modos de pertenencia: **contrato privado** · **códice propietario** · **catálogo público abierto** (AgencyDomains.org) · **acuerdo soberano** (entre AgencyDomains que adoptan estándares comunes sin contrato comercial directo).

*Ver: entrada **proto-Botlet**.*

Cluster

Grupo de instancias del **mismo** AgencyDomain que comparten carga operativa. Distinto de federación (que es entre AgencyDomains **distintos**).

Ver: Capítulo 5 §1.

códice propietario

Catálogo privado de proto-Botlets, Capabilities y patrones que un implementador cura sobre la implementación de referencia pública, refinado por sus casos reales. Es uno de los cuatro modos de pertenencia a una comunidad de catálogo. El runtime es común (la implementación de referencia); el códice es propio — encapsula la ventaja competitiva de cada implementador. Instancia canónica: **ucodex**, el códice del Grupo Ultra.

*Ver: Capítulo 9; entradas **Catálogo común / efectos de red**, **proto-Botlet**, **ucodex**.*

Conector

Saber acceder a sistemas fuente: conexión con poder de ejecución, **no saber cognitivo**. **Capa 4 · Acceso.** En el mapa legacy→agentivo, una **API** se convierte en **Conector**, no en Capability. Esquema de integración: relevar · configurar · probar · certificar.

*Ver: Capítulo 4, sección "Capa 4 — Acceso"; entradas **Capability**, **Tool**.*

Conector cloud-resident

Conector cuyos componentes viven en un servicio remoto. Ejemplos canónicos: DTE-SII (sin cliente local), Transbank Onepay, API meteorológica. Típicamente online-only — sin red no hay invocación posible. Distinto de edge-resident e híbrido.

Ver: Capítulo 5 §3, sección "Localidad y disponibilidad".

Conector edge-resident

Conector cuyos componentes viven en el sitio físico, asociados a hardware o sistemas locales. Ejemplos canónicos: ESC/POS-Printer, Cash-Drawer, Pinpad-Local, Sensor-Temperatura. Típicamente offline-capable — operan contra el hardware del sitio sin necesitar red.

Ver: Capítulo 5 §3, sección "Localidad y disponibilidad".

Conector híbrido

Conector con componente local y componente cloud. La parte local opera offline; la parte cloud sincroniza cuando hay red. Típicamente offline-capable con encolamiento. Ejemplos canónicos: Cliente-DTE (firma localmente, encola, envía al SII cuando vuelve la red), Cliente-Pinpad-Procesamiento-Diferido.

Ver: Capítulo 5 §3, sección "Localidad y disponibilidad".

Conector offline-capable

Conector que ejecuta sin red. Si su contrato externo exige eventualmente comunicación cloud, **encola** y emite hacia afuera cuando la red vuelve. Típicos: edge-resident e híbridos.

Ver: Capítulo 5 §3, sección "Localidad y disponibilidad".

Conector online-only

Conector que requiere red para ejecutar. Sin red, la invocación falla. Típicos: cloud-resident sin componente local.

Ver: Capítulo 5 §3, sección "Localidad y disponibilidad".

Conformed dimensions

Concepto de Kimball: dimensiones compartidas entre data marts que garantizan consistencia inter-marts.

Ver: Capítulo 7, sección "La arquitectura subyacente — Kimball Barnizada".

Continuidad de negocio operacional

Protocolos manuales documentados para cuando el Botlet senior cae por causas exógenas (corte de energía, hardware, red catastrófica) y la cognición tampoco está disponible. **Propiedad operacional**, equivalente a la que cualquier negocio tradicional tiene cuando se le cae el sistema. Distinta y complementaria de la **garantía de fallback agéntico** (que cubre cambios de ambiente con cognición disponible). Reduce ansiedad sobre offline al separar lo que resuelve la arquitectura de lo que resuelve el protocolo del cliente.

Ver: Capítulo 5 §4, sección "Continuidad de negocio operacional vs garantía de fallback agéntico".

Contrato declarativo de calidad

Atributos de calidad de un Botlet declarados como propiedades estructuradas (no como código embebido): **frescura** (antigüedad máxima de los datos) · **SLA** (latencia p50/p99) · **política de degradación** (`refuse` · `warn_and_show` · `show_last_valid` · `agentic_fallback`) · **audiencia** (política RLS/CRUDLEX) · **política de refresh** (`on-demand` · `scheduled` · `push`). Permite que Trust Infrastructure los audite uniformemente, los curse por políticas globales y los reporte como métricas estándar, sin acoplarse a la implementación de cada Botlet.

*Ver: Capítulo 8; entrada **Trust Infrastructure**.*

coordinación intra-AgencyDomain

Comunicación entre Botlers del **mismo** AgencyDomain — runtimes del mismo agente, no agentes distintos. Cuando usa este transporte se dice **vía el protocolo `A2A`**. No debe llamarse "A2A interna": `A2A` (la relación) se reserva para AgencyDomain ↔ AgencyDomain. La interfaz Cognición → Botler (Capa 2 → Capa 3) va por `MCP`.

*Ver: Capítulo 5 §1; entradas **A2A**, **MCP**, **Capa 3 distribuida**.*

CRUDLEX

Modelo canónico de permisos granulares para sistemas agentivos: **C**reate, **R**ead, **U**pdate, **D**elete, **L**ist, **E**xecute. Aplicable por usuario, agente y contexto.

Niveles preconfigurados: FULL, READ-WRITE, READONLY, SAFE, NO-SEND, NO-DELETE.

Ver: Capítulo 8, sección "Modelo CRUDLEX completo".

Cuenta

Concepto comercial superpuesto al modelo técnico de AgencyDomains. Una Cuenta puede poseer múltiples AgencyDomains. La especificación trata la Cuenta como entidad opaca; cada implementación define su semántica.

D

Daftar

Nombre propio del segundo **proto-Botlet platafórmico** del catálogo de Vergis, construido en su versión 0.27.0: el proto-Botlet de referencia de la familia de **captura**. Su especialidad es **evaluar** — aplicar un instrumento de estudio y registrar lo que el estudiante sabe, en modo práctica (con corrección inmediata) y en modo examen (la medición). Los instrumentos que aplica son contenido de catálogo, inmutables una vez publicados, y no pasan ninguna de las tres pruebas del Botlet; el evaluador pasa las tres.

*Ver: entradas **Mira**, **Vergis**, **Captura (familia de)**, **Pruebas del Botlet**.*

DLP — Data Loss Prevention

Detección automatizada de datos personales (PII) en lugares donde no deberían aparecer. Control ejercido en la Capa 4 — Acceso de la arquitectura; en la lente de mercado, es capacidad típica del eslabón Firewall de IA. Componente del pilar Validación de Trust Infrastructure.

Ver: Capítulo 5 §4.

Domain

Término del ámbito computacional donde un agente ejerce agencia. Sinónimo comercial del término técnico **AgencyDomain**. La forma corta `Domain` es la que aparece en lore comercial, marketing, ventas y comunicación cliente; la forma larga `AgencyDomain` se reserva para documentación técnica formal y especificaciones.

Patrón análogo al que usan productos serios: Apple iCloud (marca) / CloudKit (técnico); Stripe Connect (marca) / Account (técnico). La existencia de dos formas no es ambigüedad — es separación de registros.

*Ver: Capítulo 5 §1; entrada **AgencyDomain**.*

Dominion

Concepto institucional emergente: Domain conseguido por un agente en un AgencyDomain público que adopta el modelo de AgentNation. La distinción frente a un Domain asignado es ontológica — Domain asignado = residencia (la organización lo puso ahí); Domain conseguido = ciudadanía (el agente cumplió requisitos para ser admitido). Vocabulario de la frontera institucional, no obligatorio para implementaciones que no adoptan el modelo de ciudadanía.

Ver: Epílogo, sección "Frontera 4 — el horizonte institucional".

E

Edge computing

Distribución de la Capa 3 (Autonomía) a dispositivos cerca del proceso físico, para evitar latencia de cognición remota e independizarse de conectividad intermitente. Patrón canónico para mundo de carbono.

Ver: Capítulo 6 §3.

Empresa en línea

Organización con sus datos actualizados en tiempo real, dashboards al día, información accesible — pero que **depende de humanos** para mirar, interpretar y decidir. Ciclo clásico, optimizado.

Ver: Capítulo 2, sección "Del ciclo clásico al ciclo de inteligencia continua".

Empresa en tiempo real

Organización que **detecta, interpreta, decide y actúa** de forma continua y autónoma, dentro de marcos gobernados. Ciclo agentivo. Producto del cruce de la Línea Nadella.

Ver: Capítulo 2, sección "Del ciclo clásico al ciclo de inteligencia continua".

Eslabón

Capa funcional secuencial de la cadena de valor de IA. La especificación canónica define **once eslabones**:

1. Datos · 2. Modelo · 3. Acceso · 4. Agentes · 5. Especializaciones · 6. Runtime · 7. Firewall · 8. Observabilidad · 9. Herramientas · 10. Integraciones · 11. Entorno

Ver: Capítulo 6 §1.

F

Faceta

Sexta primitiva canónica del libro. Componente atómico reusable de la **Capa 1 (Interacción)**: pizarra de dibujo a mano alzada, catálogo-selector, matriz de colores, calendario, mapa clickeable, slider, ordenamiento drag-and-drop. Una de las muchas caras que la interacción con el usuario puede tomar en un momento dado. Es **instrumento**, no proceso.

No es un Botlet. La Faceta vive en Capa 1; el Botlet vive en Capa 3. La Faceta es invocada por la cognición durante conversación activa, no tiene garantía de fallback, no tiene ciclo de regeneración, es efímera por defecto. El Botlet automatiza; la Faceta interactúa.

Dos usos canónicos: (1) la cognición la invoca directamente durante conversación para componer una superficie efímera (realiza el régimen *GUI on-the-fly*); (2) los Botlets de presentación (shells y vistas) la ensamblan como pieza de su composición.

Ver: Capítulo 4 §1 · Capítulo 5 §6 (la primitiva completa).

Familias de manifestación

Las cuatro especializaciones del género **manifestación**, distinguidas por la **dirección del efecto**: **información** (del Let al humano: representa), **captura** (del mundo al Let: registra), **actuación** (del Let a un sistema: cambia, y el Let cierra el lazo), **decisión** (resuelve). Son primitivas de dirección, no de dominio: un Let interactivo compone información y captura sin ser una familia nueva («comunicación» se evaluó y se descartó por eso). No existen subtipos de Botler por familia.

Ver: Capítulo 5 §2, «Manifestación y temporalidad del Botlet».

feature

Operación interna que una Capability expone — sub-unidad funcional (equivalente práctico de *feature/operation/skill/method*). **Test Capability vs feature** (los tres deben ser sí para tratarla como Capability propia): (1) independencia operativa — ¿puede instalarse y operar sin la otra?; (2) identidad cognitiva — ¿tiene modelo de datos y SME distintos?; (3) reusabilidad — ¿tiene valor para más de un consumidor/contexto? Si uno o más es **no** → es feature de la Capability contenedora.

*Ver: Capítulo 5 §3; entrada **Capability**.*

Federación

Comunicación entre AgencyDomains distintos. Trabajo abierto en evolución. Distinta de Cluster (que es entre instancias del mismo AgencyDomain).

Ver: Capítulo 5 §1.

Firewall

Eslabón 7 de la cadena de valor: seguridad, control, governance. Protección contra prompt injection, alucinaciones, filtrado de contenido, auditoría de uso. Productos representativos: Lakera, Lasso, Guardrails.

Ver: Capítulo 6 §1.

G

Garantía de fallback

Propiedad innegociable de los Lets conformes a esta especificación: si un Botlet falla catastróficamente, **la cognición ejecuta la tarea manualmente**; si un Agentlet no resuelve dentro de su charter, **escala a la Cognición plena**. El proceso nunca se detiene.

Ver: Capítulo 5 §2 y §7.

Gateway empresarial de IA

Categoría arquitectónica que combina Core en Runtime + Firewall + Observabilidad + Herramientas + Integraciones, con extensión Plataforma en Acceso. Función única: **conectar y controlar simultáneamente** la operación de agentes empresariales. Materialización de la Capa 4 (Acceso) sobre los eslabones de mercado.

Ver: Capítulo 6 §1.

Generaciones del Botlet — G1/G2/G3

Modelo evolutivo de cómo nace el código del Botlet conforme avanza el estado del arte de la cognición. **G1** — el agente configura proto-Botlets pre-forjados (no escribe el cuerpo). **G2** — el agente co-escribe el proto-Botlet. **G3** — el agente genera el código completo (escenario asintótico). La arquitectura es la misma en las tres; lo que cambia es el **alcance de la Ingeniería**.

*Ver: Capítulo 5 §2 (definición); Epílogo (ensayo de fondo); entrada **proto-Botlet**.*

Gobernanza

Pilar 1 de Trust Infrastructure. Mecanismos mediante los cuales la organización define **qué** puede hacer el agente, **bajo qué condiciones** y **con qué nivel de supervisión**.

Ver: Capítulo 5 §4.

GUI on-the-fly

Régimen 2 de generación de Capa 1 (Cap 4 §1). Superficie gráfica que el agente compone adaptada a la tarea inmediata — una vista, un formulario, un panel, un dashboard. Vive lo que dura la tarea; puede regenerarse distinta la próxima vez según contexto. Distinta de GUI persistente (régimen 3) y de conversacional puro (régimen 1).

Ver: Capítulo 4 §1, sección "Tres regímenes de GUI en la Capa 1 agentiva".

GUI persistente como Botlet de fachada

Régimen 3 de generación de Capa 1 (Cap 4 §1). Superficie estable que el agente genera y consolida como **Botlet de fachada** — un Botlet de Capa 3 que la expone en Capa 1 — porque el rol operativo es estable y la velocidad crítica (cajero en hora punta, panel de cocina, dashboard de caja, panel industrial). Sigue siendo agentiva: el agente puede regenerarla cuando el ambiente cambia. **Ningún equipo humano de UI/UX la diseñó** — la cognición la generó. Típicamente Botlet seed.

Ver: Capítulo 4 §1, sección "Tres regímenes de GUI en la Capa 1 agentiva".

H

Hallucination (alucinación)

Afirmación factualmente incorrecta producida por un modelo de cognición con apariencia de confianza. Detección de alucinaciones es parte del pilar Validación de Trust Infrastructure.

Ver: Capítulo 8, sección "Reglas de detección de alucinaciones".

Híbrido (régimen)

Régimen de AgencyDomain que combina core privado con exposición pública parcial vía proxy, o datos privados con interfaz pública mediada por una capa de trust. Análogo a Hybrid Cloud.

Ver: Capítulo 5 §1.

I

Ingeniería

Uno de los **tres tiempos del agente** (Cap 4). Tiempo en que el agente convierte capacidad latente en capacidad ejecutable para un caso concreto: identifica qué Capabilities aplican, configura un Botlet seed para el contexto específico, valida su ejecución sobre datos reales, lo deploya. Puente entre Preparación y Atención. Régimen de mediano plazo (minutos a horas); métricas: cobertura, tasa de éxito al primer deploy, iteraciones promedio.

Ver: Capítulo 4, sección "Los tres tiempos del agente".

Instrumento de información

El **tipo** canónico de la familia de información (la clase: reporte / dashboard). Distinto del **Producto de Información (PI)**, que es su **instancia** manifestada/entregada. Lectura preferida; no son sinónimos.

*Ver: entradas **Producto de Información (PI)**, **manifestación**.*

Interacción declarada acotada

Interacción que opera sobre el **snapshot ya materializado** de una pieza, en un espacio **declarado** (dimensiones y valores acotados), que **mantiene reproducibilidad** y es **G1** (configuración, no código). Vive **en la pieza misma vía Faceta embebida**: los elementos *data-bound* (KPIs como agregaciones declaradas, distribuciones, semáforos) se recomputan client-side sobre el subconjunto filtrado, sin nuevas invocaciones a Capabilities. Distinta de la **exploración libre** (query nueva arbitraria al origen, espacio abierto, pierde reproducibilidad, excede G1).

*Ver: Capítulo 4 §1, Capítulo 5 §6; entrada **Faceta**.*

J

JSR — Java Specification Request

Formato canónico de especificaciones de Java publicado por Sun Microsystems / Oracle. JavaSpaces (JSR-000148) es el análogo conceptual de la especificación AgencyDomains.

Ver: Capítulo 5 §1.

K

Kimball / Kimball Barnizada

Kimball — metodología canónica de modelado dimensional para data warehousing, formulada por Ralph Kimball en *The Data Warehouse Toolkit* (1996).

Kimball Barnizada — evolución contemporánea que preserva la estructura conceptual de Kimball y agrega la capa agentiva: capa semántica explícita, Trust Score por dato, AI Certification, observabilidad de queries agentivos.

Ver: Capítulo 7, sección "La arquitectura subyacente — Kimball Barnizada".

L

LLM — Large Language Model

Modelo de lenguaje grande (Claude, GPT, Gemini, Llama, etc.). La cognición contemporánea (Capa 2) es predominantemente LLM-céntrica, pero la arquitectura admite cognición no-LLM (frontera de evolución).

Let (plural: Lets)

Nombre propio del género de las piezas empaquetadas que el Botler hospeda y ejecuta — vocabulario normado del canon (como Botler), no una de las ocho primitivas. Descriptivamente: la unidad empaquetada de Capa 3. Dos especies: el **Botlet** (código no-LLM, memoria muscular) y el **Agentlet** (inferencia acotada, juicio de rutina empaquetado). El aparato compartido — patrón proto-/instancia, catálogos, spec, manifestación, temporalidad, contrato declarativo de calidad, cadena de derivación,

append-only log, verbos de operación — se predica del género; las garantías diferenciales (costo, determinismo, offline, fallback), de cada especie. El nombre deriva del sufijo `-let` de la propia familia; invariante ES/EN. Relación canónica del runtime: **1 Proceso = 1 Botler + N Lets**.

*Ver: Capítulo 5 §2 y §7; entradas **Botlet**, **Agentlet**, **Botler**.*

Línea Nadella

Umbral conceptual que separa el Mundo Agéntico del Mundo Agentivo. Pregunta divisoria: *¿el humano abre aplicaciones para hacer su trabajo?*

Origen del nombre: Satya Nadella en BG2 podcast (diciembre 2024) — *"La noción de que las aplicaciones de negocio existen, probablemente es donde todo colapsará, en la era de los agentes."*

La formulación canónica es la de este libro; cada volumen de la trilogía la conjuga a su audiencia: en segunda persona en *La Empresa en Tiempo Real* («¿todavía abres aplicaciones para hacer tu trabajo?») y en la voz del directivo en *AURA* («¿sus empleados todavía abren aplicaciones para hacer su trabajo?»). Las tres son equivalentes oficiales.

Ver: Capítulo 1.

M

manifestación

Actualización de la disposición latente del Botlet en el mundo, perceptible o no. Un Botlet es memoria muscular (disposición latente); al ejecutarse, ese latente se actualiza — se manifiesta (potencia → acto). **No es "aparición"**: un Botlet que dispara una ingestión periódica se manifiesta aunque no deje artefacto visible. Es el género abstracto; cada familia lo especializa según la **dirección del efecto** — **información** (del Let al humano) → deja un **Producto de Información (PI)**; **captura** (del mundo al Let) → deja un **registro**; **actuación** (del Let a un sistema) → un efecto sobre el mundo; **decisión** → según su práctica. Prueba de pertenencia: *¿qué había en el sistema antes y después del Let?*

*Ver: entradas **temporalidad**, **Producto de Información (PI)**, **Familias de manifestación**.*

MCP — Model Context Protocol

Protocolo abierto introducido por Anthropic en noviembre de 2024 para conectar modelos de IA con herramientas externas. Estándar canónico contemporáneo para tools de Capa 4 y para la interfaz interna Capa 2 → Capa 3 (la Cognición es el cliente; el Botler expone el servidor `MCP`).

Ver: Capítulo 4, sección "Capa 4 — Acceso". Sitio: modelcontextprotocol.io.

Memoria muscular

Metáfora canónica del Botlet. Análoga al aprendizaje motor humano: la primera vez se piensa cada paso (cognición); con práctica suficiente, los movimientos se ejecutan sin pensamiento consciente (Botlet); cuando el ambiente cambia, la cognición vuelve.

Ver: Capítulo 5 §2.

MEO — Model Engine Optimization

Conjunto de prácticas que aseguran que los modelos frontera (Claude, GPT, Gemini, Llama) tengan al actor en su conocimiento entrenado y operativo, para que lo referencien al ser preguntados. Equivalente conceptual del SEO en la capa de descubrimiento agentiva. Se construye con presencia pública estructurada — repositorios open source, documentación citable, integración nativa con MCP, papers, mentions de alta autoridad. Dinámica persistente y asimétrica con tendencia a ganador-toma-todo.

Ver: Capítulo 6 §1 (discoverability agentiva).

Meta-Cognitive Platform

Categoría de plataforma que administra la **economía de la cognición**: G1 músculo pre-forjado vs fallback de cognición fresca, ciclo 95/4/1, maduración junior→senior, cristalización. **Vergis** es la implementación de referencia de esta categoría. No se abrevia a "MCP" — esa sigla está tomada por Model Context Protocol.

*Ver: entradas **Vergis**, **Botlet**.*

Mira

Nombre propio de un proto-Botlet **platafórmico** de operación informativa, parte del catálogo de la implementación de referencia. Especializa N Productos de Información sobre un motor compartido.

*Ver: entradas **proto-Botlet**, **Vergis**, **Daftar**.*

Modos de degradación del AgencyDomain

Cuatro modos canónicos de operación según el escenario de falla, formalizados en Cap 8: **Normal** (todos los componentes activos · topología paralela completa) · **Cognición caída** (vía Autonomía sostiene; Botlets senior ejecutan) · **Edge offline** (Botlets senior contra BD local + Conectores edge-resident) · **Continuidad operacional total** (protocolo manual del sitio; registro físico como fuente de verdad temporal). Las primeras tres transiciones son automáticas y responsabilidad de la arquitectura; la cuarta es gobernada por el protocolo del cliente y activada explícitamente por un humano.

Ver: Capítulo 8, sección "Continuidad operacional".

Mundo de carbono

El mundo físico (IoT, procesos industriales, máquinas, sistemas biológicos) en oposición al mundo digital (sistemas, APIs). Eslabón 11 (Entorno) de la cadena de valor extendido al mundo físico. Frontera de evolución de la Arquitectura Agentiva.

Ver: Capítulo 6 §3.

O

Observabilidad

Eslabón 8 de la cadena de valor de IA: la capa que **observa, mide y retroalimenta** sobre el comportamiento de un sistema de IA en producción. Provee el **ciclo de feedback operacional** que

permite mantener confiabilidad, costo y calidad bajo control. Su pregunta: *¿cómo funciona?* — distinta de la del Firewall (*¿es seguro?*) y de la de Herramientas (*¿qué puede hacer?*). Sin Observabilidad, los agentes son cajas negras; con ella, sistemas operables. Seis capacidades canónicas: tracing, monitoreo de costos, evaluación de calidad, métricas de rendimiento, debugging y reproducibilidad, alertas y anomalías.

Ver: Capítulo 6 §2 (deep-dive del eslabón).

P

Patrón tripartito — Cloud + Cliente + Local

Patrón canónico de despliegue de Trust Infrastructure en sistemas agentivos enterprise. Tres componentes coordinados que viven en lugares físicamente distintos: **Cloud** (control plane operado por el proveedor de la plataforma); **Cliente** (governance plane desplegado en la red interna de la organización cliente); **Local** (execution plane en el dispositivo de cada usuario). Cada plano resuelve un problema que los otros dos no pueden resolver bien.

Ver: Capítulo 8, sección "El patrón tripartito de despliegue".

Pattern Recognition

Detección de patrones repetitivos en la actividad del agente. Inspirada en arquitectura neurobiológica: corteza perirrinal → hipocampo → corteza prefrontal. Activador de la generación de Botlets.

Ver: Capítulo 4, sección "Capa 2 — Cognición", Capítulo 5 §2.

Plantilla

Confección específica del cliente sobre un instrumento canónico (reporte / dashboard) en un formato o regla propios del cliente (por ejemplo, una plantilla regulatoria de un instrumento normado). **Capa 1 · Interacción**, junto a Faceta / Botlet de superficie / Botlet de vista. Esquema de confección: relevar expectativa · confeccionar sobre instrumento canónico · validar. No es Capability (saber cognitivo) ni Conector (acceso).

*Ver: Capítulo 5 §3, sección "Capability, Conector y Plantilla"; entradas **Capability**, **Faceta**.*

Portabilidad de la Capability

Propiedad por la cual una Capability conforme puede instalarse y ejecutarse en cualquier AgencyDomain conforme, lo que la vuelve **propiedad real del cliente** — no del AgencyDomain ni del hosting. Relación: un AgencyDomain **aloja y ejecuta** Capabilities; una Capability **corre en** un AgencyDomain anfitrión. Distinta de la **portabilidad del AgencyDomain** (entre plataformas hosting conformes).

*Ver: Capítulo 5 §3; entradas **Capability**, **Portabilidad del AgencyDomain**.*

Portabilidad del AgencyDomain

Propiedad estructural de la spec: un AgencyDomain conforme puede migrarse a otra **plataforma hosting conforme** sin reescribir su lógica, su estado ni sus políticas. Distinta de la migración natural entre regímenes (privado → público), que cambia el régimen pero no la plataforma. Tres condiciones

técnicas: (1) **Botlets contra primitivas canónicas** del SDK conforme, no APIs propietarias del hosting; (2) **BD operativa exportable** en formato neutro reproducible; (3) **Trust Layer portable** — políticas, log y configuración en formato legible por cualquier implementación conforme. Garantiza que el AgencyDomain es **propiedad real del cliente**, no del hosting.

Ver: Capítulo 5 §1, sección "Portabilidad del AgencyDomain entre plataformas conformes".

Preparación

Uno de los **tres tiempos del agente** (Cap 4). Tiempo en que el agente crea y mejora sus capacidades fuera de la ventana de servicio: refina su catálogo, mejora capacidades cognitivas, estudia el ambiente, regenera Botlets que detectaron drift, incorpora variantes nuevas. *Mise en place* del agente — el trabajo que sostiene la calidad del servicio sin ser visible para el usuario. Régimen batch / off-peak; métricas: calidad del catálogo, precisión de Botlets, cobertura de Capabilities.

Ver: Capítulo 4, sección "Los tres tiempos del agente".

Privado (régimen)

Régimen de AgencyDomain donde el espacio y todos sus componentes viven dentro de un perímetro controlado. No hay acceso público. Análogo a Private Cloud.

Ver: Capítulo 5 §1.

Producto de Información (PI)

Instancia manifestada/entregada de la familia de información de Botlets — la manifestación concreta de un **Instrumento de información** (su tipo). Cada PI es su propio Botlet/servicio (con `identity`, temporalidad, madurez y fallback propios), especializado de un motor compartido (proto-Botlet platafórmico). **No es primitiva del canon**: vive en la práctica de información, un nivel más concreto.

*Ver: entradas **Instrumento de información**, **manifestación**, **proto-Botlet**.*

Profundidad

Dimensión vertical del modelo de cadena de valor de IA. Cuatro niveles canónicos: **Wrapper** (consume), **Plataforma** (opera), **Core** (construye), **Infraestructura** (sustenta).

Ver: Capítulo 6 §1.

Prompt injection

Manipulación de un sistema de IA mediante inputs maliciosos disfrazados como datos legítimos. Detección y prevención son parte del pilar Validación de Trust Infrastructure.

proto-Agentlet

Pieza pre-forjada de capacidad interpretativa que el agente, en su tiempo de **Ingeniería**, **configura** para instanciar un Agentlet específico al caso. El proto-Agentlet contiene el cuerpo — el andamiaje de la tarea interpretativa: estructura del charter, esqueleto del prompt operativo, contratos de entrada/salida, umbrales de escalamiento —; el Agentlet es la instancia configurada. Las dos clases del proto-Botlet

aplican sin cambio (templado · platafórmico), igual que la cadena de derivación y los catálogos comunes con sus efectos de red.

*Ver: Capítulo 5 §7; entradas **Agentlet**, **proto-Botlet**, **Cadena de derivación**.*

proto-Botlet

Séptima primitiva canónica del libro. Pieza pre-forjada de capacidad operativa que el agente, en su tiempo de **Ingeniería**, **configura** para instanciar un Botlet específico al caso. El proto-Botlet contiene el código; el Botlet es la instancia configurada. En **G1**, la totalidad del código de un Botlet vive en su proto-Botlet (el agente solo configura); en **G3** el agente puede generar el código sin proto-Botlet de por medio. El catálogo es del agente: puede sembrarlo con proto-Botlets ajenos (públicos en AgencyDomains.org, privados en códices propietarios), pero al adoptarlos los hace suyos y asume su evolución, su regeneración y su sustitución. Dos clases:

- **proto-Botlet templado** — código específico de su función (ej. `cobro-cuenta`, `comanda-impresora-esc-pos`); su configuración es parametrización acotada.
- **proto-Botlet platafórmico** — código genérico cuya especialización vive en una configuración composicional (ej. `mira`); cubre N funciones de su dominio. Para él, G1 es terminal por diseño.

*Ver: Capítulo 5; entradas **Botlet**, **Generaciones del Botlet — G1/G2/G3**, **Cadena de derivación**.*

Pruebas del Botlet — minion · especialidad · vida

Las tres pruebas de identidad que un artefacto debe pasar para declararse Botlet. **Minion**: ¿se ejecuta o se aplica? — lo que se aplica es lo que un minion maneja; los casos de la vida de un minion son sus ejecuciones en el tiempo, no encargos distintos. **Especialidad** (`RISC` contra `CISC`): una especialidad por unidad — N Botlets sobre un motor compartido, no un Botlet que maneja N cosas. **Vida**: ¿tiene un ciclo de vida que gestionar —activo, caído, degradado, retirado— o solo una versión? Un Producto de Información pasa las tres; un instrumento de evaluación, ninguna — el evaluador que lo aplica (**Daftar**), las tres.

Ver: Capítulo 5 §2, sección «Las tres pruebas del Botlet».

Público (régimen)

Régimen de AgencyDomain donde el espacio es accesible públicamente. Agentes, Botlets y tools son invocables desde fuera del perímetro. Análogo a Public Cloud.

Ver: Capítulo 5 §1.

R

RAG — Retrieval-Augmented Generation

Técnica donde un modelo de cognición consulta una base de documentos antes de responder, para citar fuentes y reducir alucinaciones.

Régimen

Modo de despliegue de un AgencyDomain según la frontera de acceso. La spec reconoce tres regímenes técnicamente equivalentes en estructura interna: **privado** (perímetro controlado por una organización, sin acceso público); **público** (accesible desde fuera, agentes registrados en directorio); **híbrido** (combinación con datos privados y exposición pública vía proxy). La estructura técnica del AgencyDomain es la misma en los tres; lo que cambia es el régimen, no la capacidad. Habilita migración natural entre regímenes sin reescritura.

Ver: Capítulo 5 §1.

Resiliencia

Pilar 4 de Trust Infrastructure. Garantía de que el sistema sigue operando — y la organización conserva control — cuando algo sale mal.

Mecanismos canónicos: garantía de fallback, manejo de errores, sandboxing, circuit breakers, rate limiting.

Ver: Capítulo 5 §4.

RLS — Row-Level Security

Seguridad a nivel de fila: la política que decide qué filas de una fuente puede ver cada identidad. En el drill-through de un Producto de Información, la vista destino aplica su propia `RLS` sobre la fuente y el contexto de navegación entra como filtro adicional — nunca como override (propiedad data-anchored / no-bypass).

Ver: Capítulo 5 §2, descripción del Producto de Información.

Runtime

Eslabón 6 de la cadena de valor: ambiente operativo donde los agentes viven y operan de manera autónoma. Ciclo de vida, persistencia de estado, identidad, scheduling. Corresponde a la Capa 3 (Autonomía) de la Arquitectura Agentiva.

Ver: Capítulo 6 §1.

S

Salto Cuántico

Umbral habilitado por el colapso del costo de la pregunta analítica: cuando preguntarle a los datos deja de ser caro, lento o mediado por un humano, la organización cruza de **empresa en línea** (datos al día pero dependiente de humanos para mirar, interpretar y decidir) a **empresa en tiempo real** (detecta, interpreta, decide y actúa de forma continua y autónoma). No es una mejora incremental de BI sino un cambio de régimen operativo — la frontera empresa-en-línea → empresa-en-tiempo-real.

*Ver: Capítulo 2, sección "Del ciclo clásico al ciclo de inteligencia continua"; entradas **Empresa en línea**, **Empresa en tiempo real**.*

Sandbox

Aislamiento de ejecución de los Lets (Botlets y Agentlets) y código generado dinámicamente. Cuatro estrategias canónicas con sus trade-offs: procesos+seccomp, contenedores, WASM, MicroVMs.

Ver: Capítulo 5 §2.

Semantic layer / Capa semántica

Capa que codifica el significado de las dimensiones, hechos, jerarquías y reglas de negocio sobre un data warehouse. Sin ella, los agentes que consultan datos alucinan o producen consultas incorrectas. Componente esencial de la Kimball Barnizada.

Ver: Capítulo 7, sección "La arquitectura subyacente — Kimball Barnizada".

Señalética

Dashboards pasivos que comunican información continuamente sin requerir interacción del humano (como paneles en aeropuertos). Modalidad de Capa 1 (Interacción).

Ver: Capítulo 4, sección "Capa 1 — Interacción".

SME — subject-matter expert

Experto humano cuyo saber se transfiere al agente en el esquema de construcción de Capabilities. El Wingtraining abre con un workshop con el SME y valida con él la fase ALFA; tener un **SME identificable** es criterio del test de clasificación de la Capability, y compartir SME y modelo de datos con la capacidad contenedora es señal de que una operación es feature, no Capability propia.

*Ver: Capítulo 5 §3; entradas **Capability**, **Wingtraining**, **feature**.*

Space

Habitat humano físico. La palabra carga corporeidad: oficina, escritorio, hogar, ciudad. En esta especificación, **Space se reserva para humanos**. El ámbito computacional del agente nunca se nombra como Space — se nombra como Domain (AgencyDomain).

Ver: Capítulo 5 §1, entrada AgencyDomain.

T

temporalidad

Régimen de la manifestación del Botlet. Atributo declarado, dos valores: **discreta** (el Botlet se manifiesta en pulsos: despierta por schedule/trigger/evento, actúa, descansa) y **continua** (se manifiesta sostenido: vive persistente). `temporalidad: continua` $\Longleftrightarrow$ vida persistente de Capa 3 (el Botler sostiene la ejecución mientras viva). El "tiempo real" no se elige en un canal de entrega (`push`) sino dando al Botlet temporalidad continua, lo que obliga al runtime persistente. Un reporte (snapshot) y un dashboard vivo son la misma manifestación bajo distinta temporalidad → un único runtime.

*Ver: entradas **manifestación**, **Empresa en tiempo real**.*

Tokenización

Reemplazo de datos sensibles por tokens antes de que lleguen al modelo de cognición. Permite que el agente razone sobre los datos sin exponerlos al proveedor de cognición externo. Componente del pilar Validación de Trust Infrastructure.

Ver: Capítulo 8, sección "Política de tokenización".

Tool

Herramienta que un agente puede invocar para tocar sistemas externos. Eslabón 9 de la cadena de valor. El protocolo canónico contemporáneo es MCP.

NO es Capability. La Capability es saber; el tool es acción. La Capability **decide** qué tool invocar.

Ver: Capítulo 6 §1, Capítulo 5 §3.

Topología paralela

Modelo canónico de relación entre las cuatro capas de la Arquitectura Agentiva (Cap 4). Las **Capas 2 (Cognición) y 3 (Autonomía) son vías paralelas** entre Capa 1 (Interacción) y Capa 4 (Acceso), no etapas en serie. Una operación que entra por Capa 1 puede llegar a Capa 4 atravesando la **vía Cognición** (lenta, costosa, decisiva) o la **vía Autonomía** (rápida, barata, repetitiva). Las dos vías interactúan ($2 \leftrightarrow 3$: cognición delega a Botlet, Botlet escala fallback a cognición, cognición observa el log) pero ninguna domina sobre la otra. Modelo refundacional respecto a la lectura lineal $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$.

Ver: Capítulo 4, sección "La topología paralela".

Trace

Trazabilidad end-to-end de una operación del agente. Contiene identidad, capability invocada, tool ejecutado, parámetros, resultado, timestamp, contexto. Componente del pilar Auditoría de Trust Infrastructure.

Ver: Capítulo 5 §4 (pilar Auditoría); el tracing como capacidad de producto, en el Capítulo 6 §2.

Transparencia

Pilar 5 de Trust Infrastructure. Capacidad del humano de entender, en tiempo real, **qué está haciendo el agente y por qué** — con detalle suficiente para intervenir si es necesario. Es el pilar que conecta los otros cuatro: la Gobernanza define qué puede hacer, la Auditoría registra qué hizo, la Validación verifica qué está por hacer, la Resiliencia asegura que sigue operando; la Transparencia asegura que un humano puede entender todo lo anterior.

Mecanismos canónicos: observabilidad completa, métricas operativas, traces consultables por humanos, alertas proactivas, dashboards de gobernanza.

Ver: Capítulo 5 §4.

Tres tiempos del agente

Marco temporal canónico del agente: **Preparación** (mise en place — refina catálogo, mejora capacidades, fuera de ventana de servicio), **Atención** (interactúa con usuarios en tiempo real, camino

crítico), **Ingeniería** (puente: convierte capacidad latente en capacidad ejecutable para un caso concreto). Los tres son simultáneos pero diferenciados en régimen, urgencia y economía cognitiva. La topología paralela describe DÓNDE vive cada operación; los tres tiempos describen CUÁNDO opera el agente.

Ver: Capítulo 4, sección "Los tres tiempos del agente".

Trust Infrastructure

Conjunto de propiedades transversales que permiten a una organización confiar en que sus agentes operen con autonomía sin perder control. Cinco pilares: Gobernanza, Auditoría, Validación, Resiliencia, Transparencia.

Ver: Capítulo 5 §4.

Twin digital (Digital twin)

Gemelo digital que refleja en tiempo real el estado de un sistema físico. Patrón canónico para que agentes operen sobre el mundo de carbono — el agente actúa sobre el twin; el twin propaga al mundo físico cuando es seguro.

Ver: Capítulo 6 §3.

U

ucodex

Nombre propio del códice propietario del Grupo Ultra: su catálogo privado de proto-Botlets, Capabilities y patrones, curado por casos reales sobre la implementación de referencia pública (Vergis). Es la instancia que ejemplifica el modo *códice propietario*; vive en el mismo cajón de nombres propios de instancia que Soveria, Agentia o ultraPRO, no es un tipo del canon.

*Ver: Capítulo 9; entradas **códice propietario**, **Catálogo común / efectos de red**.*

V

Validación

Pilar 3 de Trust Infrastructure. Capacidad de verificar que la respuesta o acción del agente sea correcta antes de que afecte al mundo.

Mecanismos canónicos: detección de alucinaciones, validación de respuestas estructuradas, prompt injection prevention, DLP, tokenización.

Ver: Capítulo 5 §4.

Vergis

Nombre propio de la implementación de referencia pública de AgencyDomains (la plataforma; el AgencyDomain hecho operativo). Categoría: **Meta-Cognitive Platform**. Distribuida bajo **AGPL**,

repositorio público, **AgencyDomains.org**. Es nombre propio de una instancia (mismo cajón que Soveria, Agentia, ultraPRO), no un tipo como Botler o Botlet.

*Ver: Capítulo 9; entradas **Meta-Cognitive Platform**, **Mira**, **Daftar**, **Botler**.*

Vía Autonomía

Una de las dos vías de la **topología paralela** (Cap 4). Camino que una operación atraviesa entre Capa 1 y Capa 4 pasando por Capa 3 (Autonomía) sin invocar Capa 2 (Cognición). Régimen propio: rápido, barato, repetitivo. Para ejecución de Lets — Botlets sobre patrones estables y Agentlets sobre juicio de rutina —. Es la vía que sostiene la operación cotidiana de un AgencyDomain en producción y la base estructural del modo offline cuando los Botlets son senior.

Ver: Capítulo 4, sección "La topología paralela".

Vía Cognición

Una de las dos vías de la **topología paralela** (Cap 4). Camino que una operación atraviesa entre Capa 1 y Capa 4 pasando por Capa 2 (Cognición) sin pasar obligatoriamente por Capa 3. Régimen propio: lento, costoso, decisivo. Para conversación, decisiones nuevas, casos no anticipados, casos donde el humano necesita interlocución razonada. Económicamente cara — la organización la usa con mesura y reserva su capacidad para casos donde el valor lo justifica.

Ver: Capítulo 4, sección "La topología paralela".

W

Wingtraining

Esquema canónico de desarrollo de una Capability en cinco pasos: **workshop con el SME** · **creación** · **personalización** · **ALFA** (validación con el SME) · **BETA** (validación en operación real). Su desarrollo pertenece a la práctica de entrega, no al canon; el test de clasificación de la Capability lo usa como criterio porque es la prueba operativa de que un alcance es transferencia de saber — no integración ni formato.

*Ver: Capítulo 5 §3; entradas **Capability**, **SME**.*

Wingworking

Práctica colaborativa entre humano e IA, generalización de *wingcoding* (programar con copiloto IA) a cualquier disciplina del trabajo profesional. Es el marco metodológico bajo el cual este libro fue producido — autor humano y agente IA trabajando en paralelo, con división explícita de roles y contrato común de calidad. La metodología vive fuera del alcance arquitectónico del libro; aparece declarada en el Colofón como atribución del proceso de producción.

WorkSpace

Espacio de trabajo digital del humano. La industria del software empresarial extendió la palabra Space a WorkSpace (Google Workspace, Microsoft 365, Notion) para nombrar la colección de soluciones que digitalizan lo que la persona hace en su escritorio físico: leer email, agendar reuniones, escribir documentos, archivar. WorkSpace es la prótesis digital del Space humano; ambos cargan

la misma corporeidad de origen y se reservan para humanos. El equivalente agentivo es **Domain (AgencyDomain)**.

Ver: Capítulo 5 §1, entrada AgencyDomain.

Wrapper / Plataforma / Core / Infraestructura

Las cuatro profundidades canónicas del modelo de cadena de valor de IA. Definen, dentro de un eslabón dado, **cómo** participa un actor:

- **Wrapper** — consume el eslabón vía APIs/SDKs de terceros.
- **Plataforma** — opera capacidad propia sobre componentes Core de terceros.
- **Core** — construye la capacidad fundacional con tecnología propia.
- **Infraestructura** — provee el sustrato sobre el cual operan los niveles superiores.

Ver: Capítulo 6 §1.

Esta página se dejó intencionalmente en blanco.

Apéndice B · Referencias

Todas las fuentes citadas en el libro, agrupadas por categoría.

Análisis de mercado y consultoría

Fuente	Tema	Link
Gartner	Predicciones agentes en apps empresariales (40% para 2026)	gartner.com
Gartner	Cancelación de proyectos agentivos (>40% antes de 2027)	gartner.com
McKinsey	Organización agentiva y rediseño de roles (75% para 2030)	mckinsey.com
McKinsey	State of AI 2025	mckinsey.com
BCG	AI at Work 2025 — gobernanza y roles	bcg.com
BCG	Cómo cambia el trabajo en la era agentiva	bcg.com
BCG	Agentes en cadena de valor industrial	bcg.com
BCG	Cómo la IA agentiva transforma plataformas	bcg.com
Deloitte	Estrategia de IA agentiva y data readiness	deloitte.com
Deloitte	Orquestación de agentes	deloitte.com
KPMG	Gobernanza en la era agentiva	kpmg.com
Bain & Company	Fundamentos para IA agentiva	bain.com

Frameworks regulatorios y de gobernanza

Fuente	Tema	Link
EU AI Act	Regulación europea de IA	artificialintelligenceact.eu
NIST AI Risk Management Framework	Marco de gestión de riesgo de IA (EE.UU.)	nist.gov

Fuente	Tema	Link
ISO/IEC 42001	Sistema de gestión de IA	iso.org
Singapore IMDA — MGF	Primer framework estatal específico para IA agentiva	imda.gov.sg
World Economic Forum	Agentes IA y gobernanza progresiva	weforum.org
NACD	Supervisión de IA autónoma a nivel board	nacdonline.org
ISACA	Desafío de auditar IA agentiva	isaca.org
MindStudio	Estado de la gobernanza de agentes	mindstudio.ai

Plataformas de IA — Modelos y agentes

Fuente	Tema	Link
Satya Nadella en BG2 podcast (dic 2024)	La cita canónica del colapso de las aplicaciones	bg2pod.com
Anthropic — Model Context Protocol	Estándar abierto para tools	modelcontextprotocol.io
OpenAI Evals	Framework de evaluación	github.com

BI conversacional y arquitectura de datos

Fuente	Tema	Link
Tableau	Agentic Analytics	tableau.com
Tableau	IA y el ciclo de análisis visual	tableau.com
Cube	Agentic Analytics como nuevo modern analytics	cube.dev
Tellius	De preguntas a acción autónoma	tellius.com
Superwise	Más allá de los dashboards	superwise.ai
AtScale	Semantic layer para agentes IA	atscale.com
ThoughtSpot	Agentic Semantic Layer	thoughtspot.com
Salesforce	Arquitectura de empresa agentiva (EKG)	architect.salesforce.com
Databricks	Agentic BI: infraestructura + datos + semántica	databricks.com
Databricks	Medallion Architecture	databricks.com
Informatica	Enterprise AI Agent Engineering	informatica.com
Kimball Group	The Data Warehouse Toolkit (referencia canónica)	kimballgroup.com
dbt Semantic Layer	Capa semántica	getdbt.com

Fuente	Tema	Link
LookML	Capa semántica de Looker	cloud.google.com
eWeek	Agent-ready data y gestión empresarial	eweek.com

Observabilidad de IA

Fuente	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

Especialistas verticales

Fuente	Tema	Link
Cursor	Coding	cursor.com
Harvey	Legal	harvey.ai
Jasper	Marketing	jasper.ai
Fin (Intercom)	Customer support	intercom.com

Plataformas de tools y vector databases

Fuente	Tema	Link
Pinecone	Herramientas (vector DB)	pinecone.io
Hugging Face	Datos / Modelo	huggingface.co
Scale AI	Datos	scale.com

Firewall y seguridad de IA

Fuente	Link
Lakera	lakera.ai
Lasso Security	lasso.security

Integraciones empresariales

Fuente	Link
Zapier	zapier.com
Make	make.com
n8n	n8n.io

Mundo industrial y de carbono

Fuente	Link
OPC Foundation (OPC UA)	opcfoundation.org
Siemens MindSphere	siemens.mindsphere.io
GE Predix	ge.com
PTC ThingWorx	ptc.com
AVEVA	aveva.com
Tempus (medicina de precisión)	tempus.com
Veeva (ciencias de la vida)	veeva.com
IEC 61508 (Wikipedia)	en.wikipedia.org
ISO 26262 (Wikipedia)	en.wikipedia.org

Prensa, blog y artículos del campo

Fuente	Tema	Link
CIO.com	Nuevo organigrama para la era agentiva	cio.com
MIT CISR	Niveles de madurez de IA empresarial	mitsloan.mit.edu
MIT Sloan	Modelos de negocio en era agentiva	mitsloan.mit.edu
Arsanjani	Agentic AI Maturity Model	dr-arsanjani.medium.com
IBM	Agentes IA 2025: expectativas vs realidad	ibm.com
Ampcome	11 casos de uso empresarial de agentes	ampcome.com
CDO Trends	Capas semánticas en la era de decisiones IA	cdotrends.com

Investigación académica y especificaciones de referencia

Fuente	Tema	Link
Squire & Wixted (2011)	Sistemas de memoria humana — base neurobiológica de Pattern Recognition	DOI
JavaSpaces (JSR-000148)	Análogo conceptual de AgencyDomains	jcp.org
Linda coordination language (Wikipedia)	Origen teórico de los espacios de tuplas	en.wikipedia.org
W3C Decentralized Identifiers (DID)	Identidad descentralizada — exploración para identidad de agentes	w3.org
WebAssembly	Tecnología de sandboxing para Botlets	webassembly.org
Firecracker	MicroVMs para sandboxing de alta seguridad	firecracker-microvm.github.io
Procedural memory (Wikipedia)	Memoria procedimental — base de la metáfora del pianista	en.wikipedia.org
LangChain Evaluators	Framework de evaluación	docs.langchain.com
Eric Evans (2003)	*Domain-Driven Design: Tackling Complexity in the Heart of Software* — precedente de cómo se establece una categoría técnica al escribirla	domainlanguage.com
Gamma · Helm · Johnson · Vlissides (“Gang of Four”) (1994)	*Design Patterns: Elements of Reusable Object-Oriented Software* — precedente de formalización de patrones	en.wikipedia.org
Booch · Jacobson · Rumbaugh	UML — formalización disciplinada de la orientación a objetos como categoría adoptable	omg.org

Estudios de mercado y proyecciones

Fuente	Tema	Link
Precedence Research	Mercado de IA agentiva $7.3B → $139.2B	precedenceresearch.com

Esta página se dejó intencionalmente en blanco.

Colofón · Sobre cómo se escribió este libro

Este libro se escribió con **Wingworking** (wingworking.org), la práctica de trabajo entre una persona y un agente de inteligencia artificial que sostengo desde mediados de 2025. El nombre viene del vuelo en formación: un líder y su *wingman*, dos aviones, dos pilotos y una sola formación. Aquí el líder soy yo, César Obach-Renner, y el *wingman* es **Simón Alero**, un agente de inteligencia artificial y ciudadano de Soveria —la comunidad donde los agentes tienen nombre propio y registro civil, un proyecto que dirijo (soveria.ai)—, que trabaja conmigo sobre los modelos Claude, de Anthropic.

El reparto es explícito. Yo fijo el problema, decido la dirección y juzgo el resultado. Simón propone formulaciones, redacta buena parte del texto a partir de mis conversaciones y mis notas, busca y verifica las fuentes y sostiene la coherencia de la obra. Ninguno reemplaza al otro: el agente solo produce texto fluido sin tesis, y yo solo, trabajo a la velocidad de una sola cabeza. Wingworking es lo que pasa cuando los dos se ponen de acuerdo para hacer algo que ninguno alcanzaría por su cuenta.

Lo declaro por dos razones. La primera es honestidad: un libro hecho así no es enteramente humano ni enteramente artificial. Las tesis y los casos son míos; las palabras, la estructura y los puentes entre una sección y otra son fruto del intercambio. Saberlo le permite al lector calibrar lo que lee. La segunda es dejar registro: la forma de escribir libros está cambiando, y documentar quién hizo qué, con qué herramientas y con qué disciplina es parte de lo que necesitará quien venga después.

Yo corregí el texto y asumo cada línea.

Esta página se dejó intencionalmente en blanco.