DESIGNING AND RUNNING AGENTIC SYSTEMS

The target architecture: the decision framework, from language to evaluation, so they hold up in production.

TARGET ARCHITECTURE FOR AGENTIC SYSTEMS

Thibault Souris.

Architecture first, model second.

Start simple, isolate behind contracts, add complexity only on evidence.

Author's note

For two years, generative AI was experienced as a rupture: we marveled that a machine could write, summarize, code. That time has passed. The question is no longer whether the technology impresses, but how to integrate it for good into processes, professions, and organizations. And there, the verdict is harsh: nearly every application calls itself "agentic," and almost none truly holds up in production. The vast majority of pilot projects fail to make that leap, and always for the same reasons: we do not know how to evaluate nondeterministic behavior, we do not know how to govern it, we cannot say in advance when it will be wrong. These are not model problems. They are architecture problems.

"Adopting AI," moreover, covers two realities that are too often confused. Giving teams generative assistants for writing or analysis is a matter of acculturation: useful, but relatively simple and with little structural risk. Integrating agentic systems into processes, existing or yet to be invented, is something else entirely: it means rethinking flows, trust, governance, reliability. It is this second adoption, the harder one, that is at stake here; and the confusion between the two explains a good share of the disillusionment.

The sign that the stakes have shifted from the model to integration lies in a decision the largest model providers have just made: rather than ship one more model, they are building deployment organizations and sending engineers to embed with their customers. When those who build the models conclude that the bottleneck is no longer the model but its integration into the real world, the case is closed: the value plays out around the model, not inside it.

Too many teams, however, still choose a framework before they have thought through their architecture, and pay for that choice for months. The conviction behind these pages is the exact opposite: it is not the model that dictates the architecture, it is the architecture that frames the model. The model is rented and replaced; the architecture is owned and compounds. Once that reversal is accepted, the rest (language, topology, memory, evaluation, security) ceases to be an initial bet and becomes a sequence of reversible decisions, made behind stable contracts.

These pages are addressed to anyone who designs, runs, or defends such systems, and wants to do so on foundations that last.

Thibault Souris · 2026

CONTENTS

PART I 4 SECTIONS

SCOPE · DECOUPLING · LANGUAGE · DECOMPOSITION

The foundations

The bedrock: why the architecture frames the model, and how decoupling makes language and topology reversible.

IN THIS PART

SECTION 01

Framing, how to read, and the guiding principle

The shift from demonstration to execution comes down to an engineering question: how do you run a reliable agentic system without binding yourself to a stack, a vendor, or a topology that will have to be undone later?

The guiding principle, restated

The founding principle is simple: the architecture frames the model, not the other way around. A strong practical consequence follows: **language and deployment topology are adapter decisions**, made behind stable contracts, not structural commitments made on day one. The business domain, for its part, does not change when the stack changes.

> Operational corollary: you do not choose a stack and then an architecture. You first set the boundaries (ports, contracts, integration protocol), which then makes the choice of language and decomposition reversible and local.

What this document covers, and what it does not

Words invite confusion, and confusion is expensive. Artificial intelligence is the entire field. Generative AI is a subset of it: it produces content, and an assistant that drafts or summarizes belongs to it. An agentic system, for its part, does not merely generate: it plans, calls tools, acts in a loop toward a goal, with memory and bounded autonomy. It uses a generative model as a component, but it is defined by its orchestration, not by its generation. At the other end, classical automation, known as RPA, is deterministic and scripted, and therefore brittle at the slightest change.

The blur is real: a fixed pipeline around a single model call is not agentic, it is augmented automation; an agent on fixed rails is close to RPA. Agentics begins with non-deterministic reasoning, tools, autonomy, and memory; between the fixed path and dynamic delegation lies a whole spectrum.

This document addresses the technical architecture of these agentic systems, and their integration into processes, existing or new. It does not address enablement, equipping people with generative assistants: equipping teams is another subject, essential but distinct. What is at stake here is integration into operations, where most projects fail. It likewise sets aside two neighboring and complementary questions: the build-or-buy tradeoff for each layer, and the mapping to specific regulatory texts. These are layers that sit on top, not blind spots: the principles that govern them, shared building blocks, graduated sovereignty, a register of uses, are laid out in Section 15. Their vendor landscape, shifting and dated by nature, is deliberately kept out of these pages: the doctrine targets what holds regardless of vendors and jurisdictions.

One last framing point, an essential one: this document describes the target architecture, complete, the one to move toward, not a prerequisite to build in a single block. The floor of a first system amounts to little: a single orchestrator, a model port, persistence, a few guardrails, and observability. Everything else, fine-grained memory engineering, state externalization, multi-agent, an evaluation loop, is added only when an objective trigger or a measured gain justifies it. The decision matrix (Section 13) and the recommendations by horizon (Section 14) state what is laid down on day zero and what is deferred. The richness of these pages is therefore not complexity imposed from the outset: it is a target to advance toward in stages, each building block justified by evidence.

What is inherited, what is new

The plumbing of this doctrine is not invented here: immutable log and replay, service discovery, compensation, idempotency, suspend and resume are distributed-systems patterns proven over decades. That is a strength: we inherit patterns validated at scale, and we pay the design cost only once, on what is specific to the component. Four things, however, did not exist in classical distributed systems, and justify a doctrine of their own. The core is non-deterministic: a guardrail cannot be a tool the model chooses to call, it is enforced in middleware (Sections 05 and 12). Untrusted input is indistinguishable from instruction: retrieved content is treated as data, never as an order (Section 11). Memory and forgetting become engineering: turning a stateless model into a stateful agent demands explicit consolidation and forgetting (Section 06). And agency carries an unprecedented blast radius: a compromised agent acts fast and irreversibly, hence bounded autonomy and approval on action (Sections 09, 11, and 13).

The questions addressed

TEAM QUESTION	ADDRESSED IN
Full TypeScript, or a back end in Python, Rust, or Go?	Sections 02 and 03
Monolith or microservices, and when to split?	Section 04
How does it actually run, and how does it hold up under load?	Sections 05, 07 to 09
How does the agent remember, and forget without betraying itself?	Section 06
How do we keep trust: observability, security, evaluation?	Sections 10 to 12
And what is shared across several applications?	Section 15

The assumptions adopted

- An agentic system that is useful in production is first a matter of architecture around the model, not of framework.
- The best systems use simple, composable patterns, and add complexity only when it genuinely improves the result.

- Every heavy decision must remain reversible, because it is isolated behind a contract.

KEY TAKEAWAYS

- A decision framework, not an imposed stack.
- Language and topology are adapter choices, reversible, made after the boundaries.
- Complexity is added only when it is justified by the result.
- The plumbing is inherited from distributed systems; four novelties justify a doctrine of their own: a non-deterministic core, input indistinguishable from instruction, governed forgetting, a wide blast radius.

SECTION 02

The decoupling that makes language and topology secondary

Before discussing language and deployment, one must understand why these choices become secondary. Two boundaries suffice: the domain's ports, and a standardized integration protocol between processes.

Boundary 1, the domain's ports

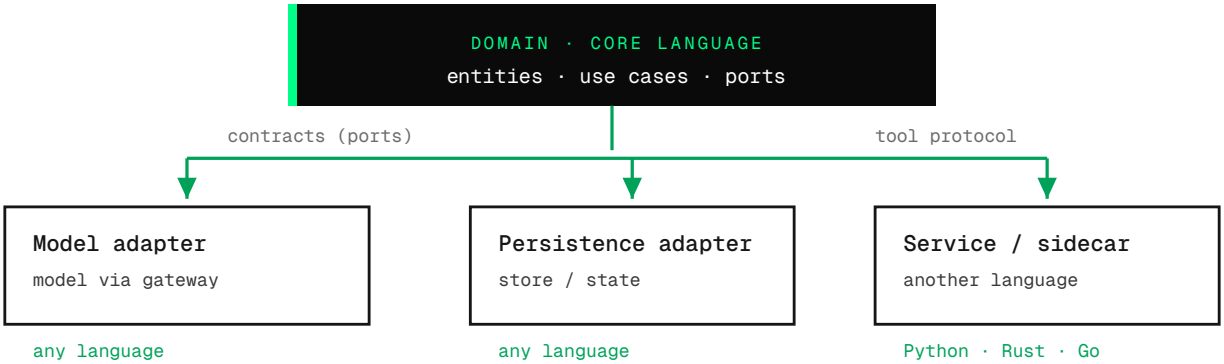
The domain expresses its needs as contracts (ports): provide a model generation, persist state, execute an action, read a source. Each contract is implemented by an adapter. The adapter can be written in any language as long as it honors the contract. Language becomes an adapter implementation detail. This decoupling has a name, hexagonal architecture, or "ports and adapters", formalized by Alistair Cockburn in 2005. Applying it to the model itself, treated as one adapter among others, is an idea the ecosystem increasingly shares; what follows claims no paternity over it, but pushes it through to its consequences, from evaluation to memory, from security to contract governance.

This last idea deserves to be pushed all the way, because it grounds the durability of the whole edifice. The engine placed behind the model port is bound only by its contract, not by its nature. Today it is a language model; tomorrow it could be a reasoning engine of another kind, and the architecture would survive it without a rewrite, as long as it honors the same contract. What would then remain to address is not the boundary but the engine's properties: as long as it reasons non-deterministically over open inputs, the evaluation, memory, and security requirements described further on hold identically. In other words, the model is not the center the architecture organizes, it is a component the architecture can replace, down to its paradigm. Let us be fair, however, about the everyday benefit of this boundary: it lies less in the actual replacement of the model, rarer than commonly claimed, than in the ability to test and reason about the domain without depending on the model. Substitutability is the insurance kept in reserve; testability is the everyday dividend. And one clarification that holds for everything else: the port is clean for the typed contract, but the behavioral boundary is not guaranteed by the type, it is validated (Section 12).

Boundary 2, the integration protocol between processes

When a capability lives in another process, even another language, it is exposed through a standardized tool protocol. The system consumes that process as a tool provider, and exposes its own use cases the same way. The cross-language boundary is sharp and stable.

Diagram, language and topology behind the contracts



> The core does not know what language the adapters run in.

Practical consequence

The debate between full TypeScript and a Python, Rust, or Go back end ceases to be an architecture dilemma. It is a local tradeoff, made adapter by adapter, according to the ecosystem of the capability concerned. The same reasoning applies to splitting into services: a service is an adapter that has migrated into its own process, without the domain changing.

The contract, beyond the type

A contract is not reducible to a typed schema: it is governed, it carries meaning, and it must come to terms with what already exists. Three facets of a single requirement.

It is governed. A stable contract is not a frozen contract. Every contract, domain port, exchanged message, or persistence schema, evolves; without discipline it drifts and silently breaks its consumers, the classic ailment of service-oriented architectures. Four rules govern it. It carries a version. Its evolution is additive by default, and the consumer is a tolerant reader: it ignores what it does not know and accepts the absence of an optional field, which makes additions non-breaking. A genuinely breaking change is not made in one block, but in two steps, expand and contract: introduce the new, migrate the consumers, remove the old once no one uses it anymore. Finally, provenance, which already recorded what a prompt saw and what memory supplied, extends to contracts and events: knowing who produces, who consumes, and under which version makes it possible to measure the impact of a change before making it. Verification, for its part, belongs to the consumer (Section 12): it expresses its expectations as an executable contract, and a producer change that would break them fails at integration, before reaching production.

Messages and events that cross a boundary carry a stable envelope, separate from their payload: an identity, a qualified type, a schema reference, a timestamp, an origin. This taxonomy is what lets a consumer route, filter, and trace without knowing the producer's details.

It carries meaning. A contract honored in form can still denote different things. "Customer", "account", "commitment", "validation" do not mean the same thing from one system to the next, nor from one agent to

the next. The hard part of interoperability is not "how do we call each other" but "are we talking about the same thing". The answer is not a central data model that would say everything for everyone: such a model quickly becomes unmanageable, slow to evolve, and ends up bypassed. It is, per context, an explicit shared language, the one the business and the code share, sharp boundaries between contexts, and a minimal shared vocabulary at the seams only. The adapter plays its second role there: beyond translating a protocol, it aligns meaning, exactly as the anticorruption layer translates the dialect of a legacy system. This alignment counts doubly for an agentic system: absent fixed semantics, a model will force neighboring but distinct concepts together and invent false correspondences. The semantic contract is, against these hallucinated alignments, what the typed schema is against malformed data.

It comes to terms with the existing. On greenfield, the port is a sharp boundary and the adapter is thin. On an existing system, the boundary is not given: its contracts are implicit, its semantics undocumented. The adapter then becomes an anticorruption layer that translates the legacy system's dialect into the domain's language and keeps the old model from contaminating the new. Moving off an existing system is done by progressive replacement, feature by feature, never in a single stroke. It must be said plainly: the ports and adapters model holds best on greenfield; on legacy estate it still holds, but the adapter pays for an explicit translation, and that is where the real cost of integration concentrates. Isolation is precisely what makes that integration clean, provided one never pretends the boundary is free.

KEY TAKEAWAYS

- Two boundaries make language secondary: the domain's ports and an inter-process tool protocol.
- An adapter can be written in any language as long as it honors the contract.
- A service is just an adapter that has migrated into its own process; the domain does not change.
- The contract is a governed artifact: versioned, additive by default, read by a tolerant consumer, and evolved by expanding then contracting, never by breaking silently.
- The contract also carries meaning: a shared language per context and a minimal vocabulary at the seams prevent hallucinated alignments, where a central data model would become unmanageable.
- Facing legacy estate, the adapter is an anticorruption layer: the model holds best on greenfield, and the boundary is never free on the existing.

SECTION 03

Language choice, full TypeScript or polyglot

The question is not one language for everything, but the right language in the right place, behind contracts. Here is the recommended position, grounded in systems already built.

1 TypeScript for orchestration and the interface

For the agentic core (orchestrator, tool registry, use cases) and the cockpit, a single language across server and interface avoids doubling the stack and multiplying maintenance surfaces. Types are shared between the back end and the front end. The model abstraction stays light: a direct SDK, not a heavyweight chaining framework, to keep control of prompts, events, and token cost.

2 A specialized language as a sidecar when the library demands it

When a capability depends on a mature ecosystem in another language (browser automation, scientific computing, specialized models), it is isolated in a sidecar exposed through the tool protocol. This is the case, for example, of an execution engine driven from a dedicated sidecar. The core consumes it as a tool provider, without absorbing its stack.

3 Rust or Go for performance-sensitive components

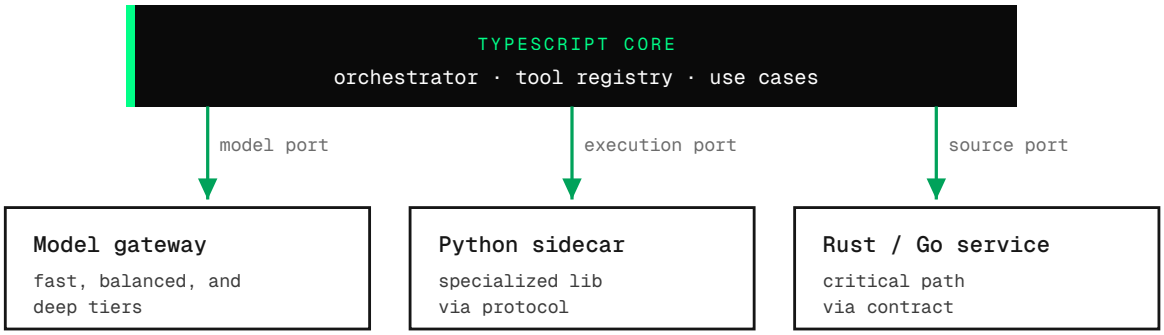
For a component on the critical path (a high-throughput gateway, intensive parallel processing, a tight-latency service), a compiled language is justified. It is then exposed as an adapter or a service behind a contract, without contaminating the rest.

Language decision matrix

COMPONENT	RECOMMENDATION	WHY
Orchestrator, tool registry, use cases	TypeScript	Single stack with the interface, shared types, light model abstraction.
Cockpit and interface	TypeScript	Continuity with the back end, unified streaming and rendering.
Capability with a specialized ecosystem	Python as a sidecar	Mature libraries, isolated behind the tool protocol.
Component on the critical path	Rust or Go	Latency and throughput, exposed behind a contract.

COMPONENT	RECOMMENDATION	WHY
Model gateway	Any language	Centralized governance and routing, behind the model port.

Diagram, a TypeScript core and polyglot sidecars



The language of each building block is a local choice, not an architecture decision.

Anti-pattern to avoid: doubling the stack by default (a back end in one language, an interface in another) without necessity. Going polyglot is justified by a library or a performance constraint, not by habit. When it is justified, it goes through a sidecar or a service, never through a mix inside the core.

KEY TAKEAWAYS

- TypeScript for orchestration and the interface, a single stack and a light model abstraction.
- Python as a sidecar when a specialized library demands it, isolated through the tool protocol.
- Rust or Go for a component on the critical path, behind a contract.
- Going polyglot is justified by a technical reason, never by default, and always behind a boundary.

SECTION 04

Modular monolith or microservices

The answer is not ideological. It is sequential: start with a well-partitioned modular monolith, then externalize a component as a service only when an objective criterion triggers it.

1 Start with a modular monolith

A hexagonal modular monolith brings together, in a single deployable, a pure domain and its adapters, plus optional sidecars for specialized capabilities. It offers the deployment simplicity, transactional consistency, and iteration speed a team needs at the start, without giving up modularity, since the boundaries are already established by the ports.

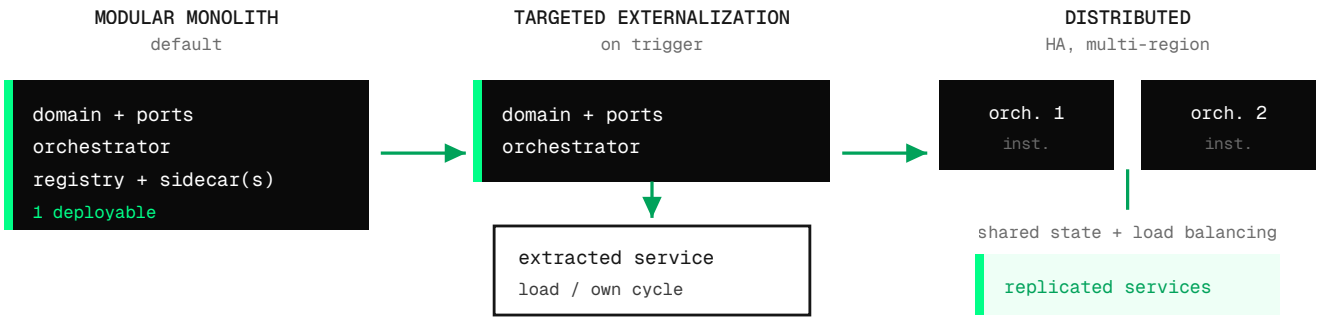
2 Externalize a component as a service, on trigger

Since the boundaries already exist, extracting a component into its own process is a local operation, not a rewrite. It is done only when a criterion justifies it. As long as no trigger is reached, splitting into microservices adds operational cost without benefit.

Externalization trigger matrix

OBSERVED TRIGGER	RESPONSE
A single instance can no longer handle the load	Externalize the state, then replicate the affected service.
A component has a very different load or cost profile	Extract it as a service to size it independently.
A component has its own delivery cycle	Decouple it to deploy it separately.
A component must be isolated for resilience or security	Extract it behind a dedicated process boundary.
High availability or multi-region requirement	Distributed topology, shared state, load balancing.
None of these criteria	Stay with the modular monolith, it is the right default.

Diagram, topology trajectory



The special case of multi-agent

Multiplying independent agents is the agentic equivalent of microservices, and calls for the same caution. The form that dominates in production remains the single orchestrator directing specialists (hub and spokes). Distributed multi-agent brings gains on parallelizable tasks, but degrades sequential reasoning and complicates debugging. It is justified for highly parallelizable complexity, not on principle.

A word on emergence, because the term invites confusion and covers two distinct levels. At the inference level, the model's behavior is already emergent: it is the non-deterministic engine that this whole architecture exists to capture and make usable. It is not opposed to anything; it is embraced. The trade-off lies elsewhere, at the topology level: should there be a swarm of self-organizing agents, with no conductor, or a central orchestrator directing specialists? It is there, and only there, that orchestration is chosen. Self-organization promises more autonomy and distributed resilience, but at the cost of predictability, debugging, and traceability, three non-negotiable requirements in regulated operations. Control is therefore kept at the system level, over an engine that is, for its part, left fully emergent, and collective autonomy is opened only through guarded stages.

The rule that decides is a single one, and it is the same for this entire section: stay central as long as everything fits in a single orchestrator process; switch a component, whether delegation, discovery, or budget, to its distributed adapter as soon as it crosses a process, host, language, or trust boundary, and only that component. The shared store that then unlocks multi-instance (Section 07) hosts the discovery registry and the budget allocation registry: a single boundary governs state, budget, and discovery.

KEY TAKEAWAYS

- Default: hexagonal modular monolith, boundaries already established by the ports.
- A component is externalized as a service on an objective trigger, not on principle.
- Distributed multi-agent follows the same caution as microservices: useful for parallelizable complexity, costly for sequential reasoning.
- The single orchestrator directing specialists remains the dominant form in production.
- A single switching rule: central as long as everything fits in one process; distributed adapter (delegation, discovery, budget) as soon as a process, host, language, or trust boundary is crossed.

PART II 4 SECTIONS

TOPOLOGY · MEMORY · EXECUTION

The execution of an agentic system

How the system runs: runtime topology, memory and context engineering, state, concurrency, and asynchronous work.

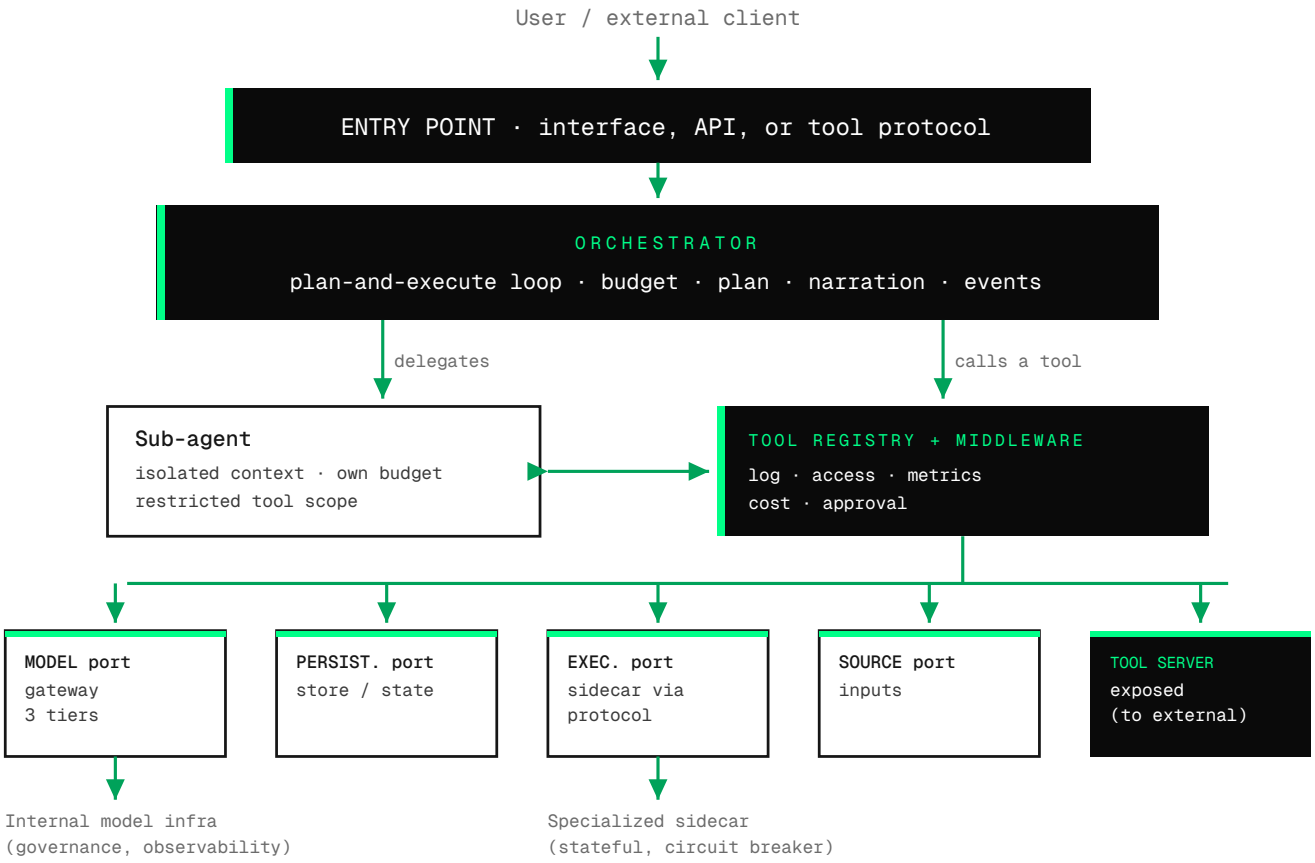
IN THIS PART

SECTION 05

Runtime topology of an agentic system

Stripped of its stack, a running agentic system always keeps the same silhouette. Its components are stable; only their language and their partitioning vary, according to the preceding sections.

Runtime diagram



CROSS-CUTTING OBSERVABILITY · logs · events · lineage · metrics · cost

Reading the diagram

- **Orchestrator**: drives the loop, plans, delegates, synthesizes. It contains no business logic; it composes.

- **Tool registry plus middleware**: the single surface through which logging, access control, metrics, cost, and approvals pass. Adding a guardrail happens here, without touching the tools. It is also from here that a **single trace stream** is emitted, consumed separately by provenance, evaluation, and observability. This surface carries only cross-cutting concerns, never orchestration or business logic: it is the guard against the pitfall of the bloated integration bus of service-oriented architectures, where the channel ends up deciding the business. The logic stays in the tools and the orchestrator; the channel that connects them remains thin, and the registry remains an index, not a brain.
- **Secondary ports**: model (via a gateway that centralizes governance and tier-based routing), persistence, execution (often a specialized sidecar), input sources.
- **Exposed tool server**: the system publishes its use cases as standardized tools, so it can be driven by an external orchestrator without copying logic.
- **Cross-cutting observability**: every step leaves a trace usable after the fact, and that trace feeds continuous evaluation, not just a dashboard.

The sidecar as a recurring pattern

A specialized capability (for example an execution engine driven by another language) runs in a session-scoped sidecar, consumed via the tool protocol. The sidecar is treated as an external dependency: health monitored, circuit breaker in case of unavailability, documented restart procedure. The rest of the system keeps running in degraded mode if the sidecar goes down.

The registry, a port with two adapters

Discovering which tools and which sub-agents are available is, like everything else, an adapter decision. As long as everything lives in a single process, the registry is a simple in-memory table, atomic and free. As soon as a capability moves out of the process, comes from another language, or from another team, the same port switches to a **service discovery** adapter: each agent publishes a self-describing capability card, discovered via a shared registry, by announce and subscribe with a local cache, never by a central call at every iteration. The discovery contract does not change; only the adapter changes with the topology.

KEY TAKEAWAYS

- Stable components: orchestrator, governed tool registry, secondary ports, exposed tool server, cross-cutting observability.
- The model gateway centralizes governance and tier-based routing, behind the port.
- The sidecar is the recurring pattern for integrating a specialized capability, treated as a monitored external dependency.
- The middleware emits a single trace stream: provenance, evaluation, and observability consume it separately.
- The registry is a port: an in-memory table in a single process, capability cards discovered via a shared registry when distributed.
- Middleware and registry carry only cross-cutting concerns, never orchestration: the logic lives in the tools and the orchestrator, the channel stays thin, so as not to recreate the bloated integration bus.

SECTION 06

Memory and context engineering

Memory is the primary form of an agent's state, and the main determinant of both quality and cost. An agentic system is not just a loop that calls tools: it is a loop that remembers, forgets, and contextualizes itself. This section elevates memory and context engineering to the rank of a pillar, on par with orchestration and the model boundary.

> Guiding principle of this section: the model boundary does not apply only to generation, it also applies to retrieval. Not everything is retrieved, all the time. The decision of what to load is deterministic, not entrusted to the model.

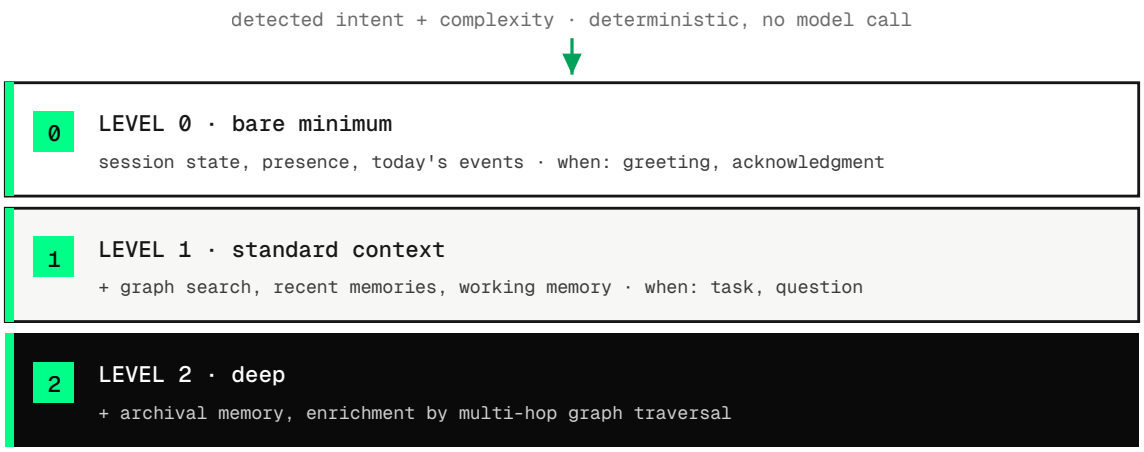
A single store is not enough

No memory technology covers all needs on its own. Several stores are therefore composed, each with its own role, behind ports, with graceful degradation if one is absent.

MEMORY ROLE	WHAT IT CARRIES	CHARACTERISTIC
Memory source	Facts and memories with enriched metadata	Simple write and read, rich metadata
Knowledge graph	Relations between memories, entities, events	Automatic linking, vector search, traversal
Working memory	Current session state, presence	Real time, low latency
Archival memory	Deep, rarely accessed history	Voluminous, queried only when needed

The frugality of tiered retrieval

The heart of memory engineering is not querying everything at every turn. A deterministic decision, based on the detected complexity and intent, chooses the retrieval depth. This is the model boundary applied to data retrieval.



Refinements: a recall intent forces level 2; a greeting forces level 0;
the router targets the useful sources (for example, calendar to the graph only).

The effect is twofold: the latency of a trivial turn collapses, and the context cost concentrates where it creates value. Most turns do not deserve deep retrieval, and the system decides this without burning a model call to find out. At every level, the same value score (recency, frequency, relevance, importance, centrality) orders what surfaces. Its weighting is not universal: it is calibrated per domain and verified like any deterministic gate, rather than set by decree.

Forgetting is a first-class function

A memory that never forgets becomes noise. But forgetting is not erasing: by default, it means ceasing to surface a memory, not destroying it. Forgetting happens at retrieval, governed by a value score, and physical deletion remains the exception. Decay with reinforcement at each access merely follows the forgetting curve described by Ebbinghaus in 1885: what is not called upon fades, what is recalled is reinforced.

MECHANISM	PRINCIPLE	EFFECT
Value score	A score combines recency, access frequency, relevance, importance, and centrality in the graph. Age is only one signal, modulated by reinforcement and salience.	What is relevant surfaces; the rest withdraws from view, without being destroyed.
Invalidation, not deletion	A contradicted fact is marked invalid with its two times (learned, valid), instead of being deleted. This is the bi-temporal model.	The history is kept: what is true now, and what was true before.
Consolidation	Near-duplicates merge, close episodes are summarized into higher-level facts.	Forgetting is often compressing: less volume, as much meaning.

MECHANISM	PRINCIPLE	EFFECT
Fact immunity	Stable facts and identity invariants are flagged and exempted from decay.	What constitutes identity never fades.

Physical deletion, for its part, is reserved for low-value derived data, for example old, invalidated associative links, under an explicit retention rule, and for mandated erasure only. Demoting to a cold archive is always preferred to destroying. Importance, the central signal of this score, does not cost a model call: it is derived from deterministic signals, access frequency, centrality, category, novelty, and from a label emitted as a by-product of the asynchronous extraction already performed outside the turn. The model boundary, applied all the way down to the computation of importance. Consolidation, for its part, remains derived and reversible: the original survives in the immutable log (§07), the source of truth is never consolidated, and each summary keeps an explicit link to the raw facts it derives from, so that an audit can always trace back to the origin, even after working memory has forgotten the detail. Finally, any retrieved content enters the context as untrusted data, never as an instruction: memory is also an indirect injection surface (§11).

Linking happens on its own

Rather than asking the user to tag and organize, the system automatically links memories to one another and to entities, by similarity above a threshold. The connections emerge in the graph. The threshold alone is not enough: it produces weak edges and hub nodes; weakly supporting links must be pruned, not only invalidated ones, to preserve the graph's precision. Deep retrieval exploits these links through a traversal of a few hops around the direct results.

The heavy work happens outside the interactive turn

Fact extraction, consolidation of related memories, graph updates, session summaries: these costly operations do not happen while the user is waiting. They are delegated to a background, asynchronous process, triggered after the turn. Memory tidies itself while the system is at rest, not while it is responding.

Validity over time, the heart of the bi-temporal model

Decay handles age, but not validity. A fact true yesterday may be false today. Hence the two times carried by each fact: when it was learned and when it is valid. This is what makes it possible to invalidate without destroying, and to reason about how information evolves, querying what is true now or what was true at a given date. Coupled with the immutable log of Section 07, the bi-temporal model reconciles the forgetting of reasoning with the traceability of audit. When two facts contradict each other, the most recently learned one invalidates the older, unless a more authoritative source says otherwise; both remain in the history.

KEY TAKEAWAYS

- Memory is state, and an architectural pillar in its own right, not a feature of the model.
- Frugality of tiered retrieval: a deterministic decision chooses the depth, the model boundary applied to retrieval.
- Several stores by role, behind ports, with graceful degradation.
- Forgetting is not erasing: a value score (recency, frequency, relevance, importance, centrality) withdraws from view; invalidation and consolidation preserve the history; the heavy work happens in the background.
- Importance is derived from deterministic signals and a label amortized over the asynchronous extraction, with no model call on the turn.
- The bi-temporal model (two times per fact) reconciles the forgetting of reasoning with the traceability of audit.

SECTION 07

State, concurrency, and scaling

Scaling an agentic system depends less on the model than on state management. The rule is simple: the agent is a reducer with no hidden state, and the state lives outside. But "state" is not monolithic: it reads as three layers that must be distinguished, so as not to confuse what must endure with what must be able to be forgotten.

1 The agent as a stateless reducer

An agent turn is treated as a function that takes an input state and an event, and produces an output state, with no hidden internal memory between calls. Each turn acts only on the explicit context it is given. This makes executions reproducible, debuggable, and allows the service to be replicated without surprises.

2 Explicit state, in three layers

All interactions are recorded as an append-only event log (event sourcing), attached to a business entity. The technique is old: it is the heir of double-entry bookkeeping, codified by Luca Pacioli in 1494, the first ledger where nothing is ever erased: a correcting entry is added. But this log is not the agent's working memory: it is its cold source. Three layers stand apart.

- **The interaction log** is immutable: source of truth, the means of auditing and replaying a trajectory from the recorded outputs. Nothing in it is ever erased.
- **Working memory** (Section 06) is a derived view, a projection (CQRS) of this log and of the extractions that follow from it. It is the layer that forgets, densifies, and contextualizes itself, freely, because the log always reconciles it.
- **Prompt provenance** (Section 08) connects the two: at each inference, the fingerprint of what was actually injected is kept, in order to replay exactly what the model saw, even after forgetting.

Confusing these layers, making the log "both the memory and the audit", forces impossible compromises: either the agent is weighed down with multiple contexts, or audit data is destroyed in trying to lighten the memory. Separating them dissolves the dilemma.

Replaying does not mean re-calling the model. A model call is not deterministic: you re-read the recorded output, you do not re-run the model, with the output recorded by provenance. The flow of the trajectory, on the other hand, is deterministic and replays faithfully. Three uses stand apart: audit replay reconstructs a trajectory from outputs already recorded; resumption restarts from a checkpoint without replaying everything; replication, which is precisely what the absence of hidden state allows, has another instance carry the same explicit state. Temporal consistency, knowing which version of prompt and model produced which output, is carried by provenance.

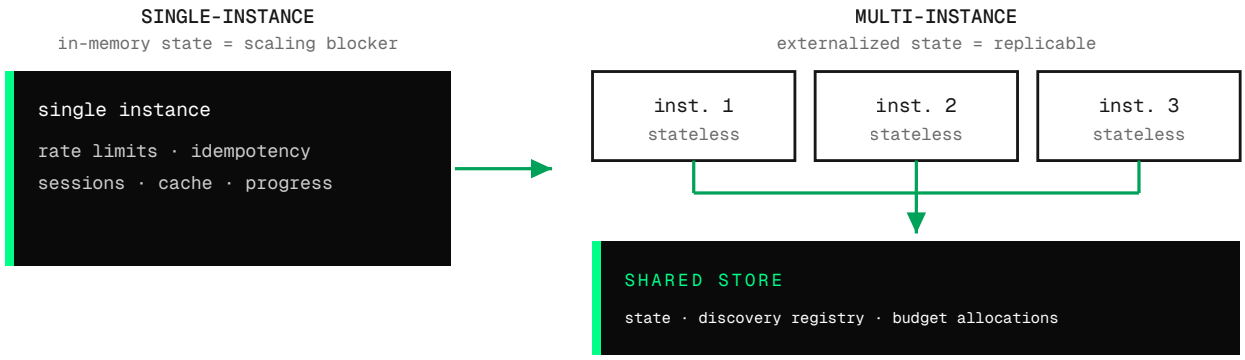
You do not erase the truth, you cool it. The immutable log grows, but its hot portion stays bounded: old segments compacted and archived cold, retained for audit. Physical deletion is the exception, reserved for low-value derived data under a retention rule, and for mandated erasure only, rendering a piece of data unreadable without breaking the audit chain.

3 From single-instance to multi-instance

In a single instance, in-memory structures suffice for rate limits, idempotency, sessions, cache, and progress. These are precisely the structures that block replication. Moving them into a shared store is the move that opens up multi-instance, without rewriting the domain. That same shared store, as soon as the single-process boundary is crossed, hosts three things at once: the externalized state, the capability discovery registry, and the budget allocation registry.

The domain does not change, but the semantics do, and this move is not free. A local, atomic access becomes a network call, with its latency and its partial failures. Strong consistency gives way to eventual consistency at the global level: the state converges without being instantaneously shared. Session consistency, however, often remains required, read-your-writes, following the order of causality, which is not the same thing as pure eventual consistency. An operation that spans several processes is no longer a single transaction; it becomes a sequence of local steps, each equipped with a compensation action that undoes its effect if a later step fails. Causal order is preserved explicitly, an effect never precedes its cause, through per-entity sequence numbers or logical clocks; and idempotency must hold under concurrency, not only on retry, through idempotency keys and a deduplication store. This is the entry price of distribution: it is paid only when an objective trigger demands it, and with full knowledge of the cost.

Diagram, externalizing state to replicate



Concurrency and budgets

Several sub-agents can work in parallel; each has its own iteration budget, capped, and a global cap per root task bounds the whole. As long as everything fits in one process, this cap is held by an aggregate in-memory counter, atomic and free because the event loop serializes accesses, a guarantee that falls away as

soon as you distribute. As soon as the sub-agents spread across processes, the per-token central counter becomes a bottleneck: you move to lease-based block allocation. The orchestrator hands out batches; each sub-agent keeps a local counter, asks again as it nears exhaustion, and returns what is unused, strict control without permanently locking the store. The batch is time-stamped: a sub-agent that goes down sees it expire and be released, rather than tying up the budget. Concurrency is useful for independent subtasks; it must not mask sequential reasoning, which, for its part, benefits from staying in a single thread.

KEY TAKEAWAYS

- The agent is a reducer with no hidden state: each turn acts on an explicit context and produces an explicit state.
- State reads as three layers: immutable interaction log (truth, audit, replay), derived working memory (which forgets), prompt provenance (which reconciles).
- Replaying re-reads the recorded outputs, without re-calling the non-deterministic model; audit replay, resumption, and replication are distinguished.
- Crossing the process boundary changes the semantics: eventual consistency, compensation instead of a single transaction, explicit causal order, idempotency under concurrency. A cost assumed on trigger, not by default.
- You do not erase the truth, you cool it: log compacted and archived cold, physical deletion reserved for derived data and mandated erasure.
- Externalizing state unlocks multi-instance; the shared store then hosts state, the discovery registry, and the budget allocation registry.
- Budgets bounded per sub-agent with a global cap: central counter in a single process, lease-based block allocation as soon as you distribute.

SECTION 08

Asynchronous work and scheduling

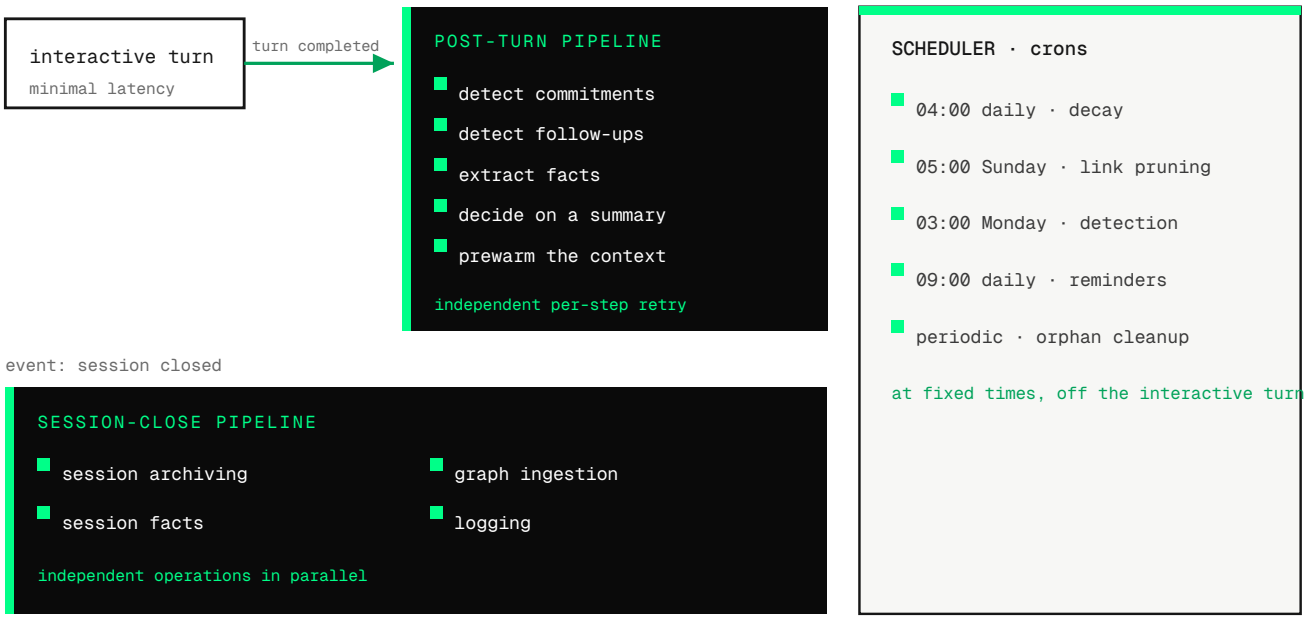
A quality agentic system does little while the user waits, and a great deal while their back is turned. The interactive turn stays thin; heavy, costly, or deferred work is pushed to the background. This section is the runbook for that separation: what runs synchronously, what runs asynchronously, through which triggers, and under which safeguards.

> Guiding rule: anything not needed for the current turn's response leaves the turn. Extraction, consolidation, synchronization, summaries, reminders: in the background. The interactive turn pays only for indispensable latency.

Two regimes: event-driven and scheduled

REGIME	TRIGGER	USE
Event-driven	A business event is emitted (turn end, session close, reminder created)	Immediate reaction but off the turn: processing starts as soon as possible, without blocking the user.
Scheduled	A clock (cron expression) fires at a fixed time	Periodic maintenance: decay, pruning, cleanup, trend detection.

Diagram, the post-turn pipeline and the scheduler



1 The post-turn pipeline

After each turn, an event is emitted and an asynchronous pipeline fires. It chains independent steps: detect the commitments made, detect the follow-ups to schedule, extract the facts to memorize, decide whether a conversation summary is needed, and prewarm the deep context for the next turn. Each step is isolated: it has its own per-step retry, and the failure of one step does not fail the others. This is also where the importance of memories is computed, as a by-product of extraction, with no dedicated model call.

Finally, at emission, the turn deposits its **provenance fingerprint**: a request ID and the signature of what was actually injected into the context. It is what makes it possible to replay exactly what the model saw, even after working memory has forgotten (cf. section 07). And this same trace feeds behavioral evaluation: a single stream emitted by the middleware, multiple consumers, provenance, evaluation (section 12), and observability.

2 The session-close pipeline

At the close of a session, a second pipeline archives the session, extracts and structures its content into the knowledge graph, derives the durable facts from it, and logs the activity. Independent operations are launched in parallel, then recombined.

3 The periodic scheduler

System maintenance is entrusted to tasks scheduled by cron expression. Sample cadences, readable directly in the configuration:

TASK	CADENCE (CRON)	ROLE
Memory decay	`0 4 * * *` (daily, 04:00)	Updates the value score of memories; gates retrieval, does not delete.
Link pruning	`0 5 * * 0` (weekly, Sunday 05:00)	Prunes only invalidated and old derived links, under retention.
Cold archiving	`0 2 * * 0` (weekly, Sunday 02:00)	Compacts and cools old journal segments, kept for audit.
Trend detection	`0 3 * * 1` (weekly, Monday 03:00)	Spots recurring patterns to capitalize on.
Expiration reminders	`0 9 * * *` (daily, 09:00)	Emits the reminders that have come due.
Imposed erasure	On demand	Renders a subject's data unreadable (legal erasure) without breaking the audit chain.

4 Safeguards for asynchronous work

Poorly guarded background work is worse than no background work. Four safeguards are systematic:

- **Per-step retry**: each step replays independently, with a bounded number of attempts, without replaying what has succeeded.
- **Idempotency**: a replayed task produces no duplicate; the effect is the same whether it runs once or three times. It is what protects consistency when a task is delivered or replayed several times, including under concurrency. The question of ordering, a distinct one, belongs to causality (section 07), not to idempotency.
- **Bounded concurrency**: the number of simultaneous tasks is capped, and often partitioned per user, to keep one actor from saturating the queue.
- **Job observability**: every execution is traced; queue lag and failure rate are first-rank metrics.

Runbook, synchronous or asynchronous

OPERATION	WHERE
Produce the turn's response	Synchronous
Memory retrieval at the level required to respond	Synchronous, tiered
Fact extraction, consolidation, graph ingestion	Asynchronous, post-turn

OPERATION	WHERE
Session archiving and summary	Asynchronous, session close
Decay, pruning, cleanup, trends	Asynchronous, scheduled
Prewarming the next turn's context	Asynchronous, post-turn

KEY TAKEAWAYS

- The interactive turn stays thin; heavy work moves to the background, by event or by cron.
- The post-turn pipeline chains steps with independent per-step retry; session close archives and consolidates.
- The turn deposits a provenance fingerprint; this same trace, emitted once by the middleware, serves provenance, evaluation, and observability.
- Periodic maintenance updates scores, invalidates, cools to cold storage, and prunes only derived data; it does not destroy the truth.
- Four non-negotiable safeguards: per-step retries, idempotency, bounded concurrency, job observability.

PART III 4 SECTIONS

RESILIENCE · OBSERVABILITY · SECURITY · EVALUATION

Robustness, security, and evaluation

Holding up over time: runtime resilience, observability, cost control, security, and behavioral evaluation of agents.

IN THIS PART

RESILIENCE · OBSERVABILITY · SECURITY

SECTION 09

Execution resilience

An agentic system composes nondeterministic, fallible dependencies: model, sidecars, external services. Resilience is designed in by default, not after the incident.

Classify the error before reacting

Each failure is first classified, because the right response depends on the type of error. We distinguish transient errors, validation errors, and business errors.

ERROR TYPE	STRATEGY
Transient (network, timeout, overload)	Retry with bounded exponential backoff.
Validation (output not conforming to the contract)	Single retry with the diagnostic reinjected into the context.
Business (missing resource, impossible action)	Delegation to a diagnostic step, or escalation for human review.
Model provider unavailable	Automatic failover to a fallback provider, behind the same port.
Non-critical service unavailable	Degraded mode; the application continues without the affected function.

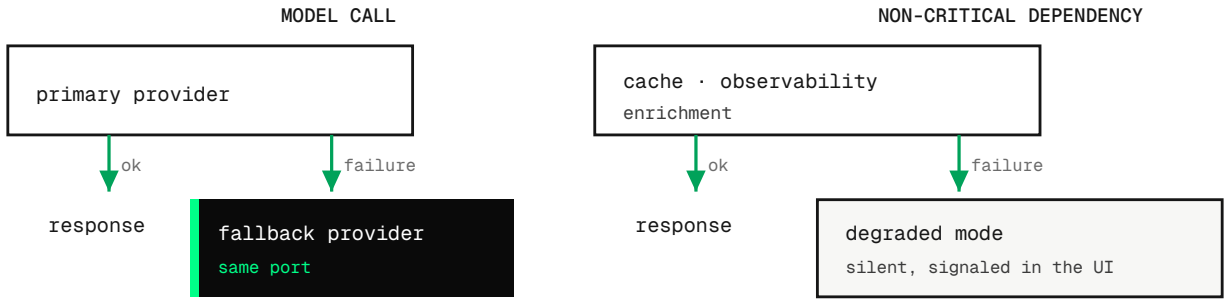
1 Fail-open for the non-critical, fail-closed for the sensitive

A non-critical dependency that goes down must not bring down the system: cache, observability, enrichment switch to silent degraded mode. Conversely, a sensitive action (mutation, write, external push) fails closed and explicitly rather than executing in doubt. The rule: degrade reads, never actions.

2 Feature detection and bimodal deployment

At startup, the system detects the services actually available according to the configuration. A missing dependency cleanly disables the associated function, without crashing. The result is a bimodal deployment: a minimal mode that runs on the bare essentials, and a full mode when all dependencies are present. The interface signals what is active rather than failing silently.

Diagram, failover and degradation




```
SENSITIVE ACTION: no silent fallback. Failure closed, explicit, traced.
```

KEY TAKEAWAYS

- Classify the error first: transient, validation, business, provider, non-critical.
- Fail-open for reads and the non-critical, fail-closed and explicit for sensitive actions.
- Multi-provider failover behind the same port, with no rewrite. Distinct from promoting a new model, which is first validated in shadow deployment (section 12): the availability fallback is immediate, promotion is earned.
- Feature detection at startup and bimodal deployment, minimal or full.

SECTION 10

Observability and cost control

An unobserved agentic system is ungovernable and unpredictable in cost. Observability is not an end-of-project option; it is a design property, carried by the middleware chain. And the trace it produces does not serve the dashboard alone: the same stream feeds continuous behavioral evaluation (section 12) and provenance (section 08).

Three independent levels of observability

LEVEL	WHAT	USE
Structured logs	One line per event, typed context (module, request ID, session).	Standard aggregation, search, alerting.
Lifecycle events	Every orchestrator step and every tool call persisted.	Replay of a trajectory, behavioral audit.
Per-model-call metrics	Input and output tokens, cache, tier used, duration, status, attempt.	Cost and performance tracking per agent.

Propagating a request ID

A unique identifier is propagated from the entry point to every tool and model call, and down to subagents via parent-child lineage. It is what makes it possible to reconstruct a complete trajectory after the fact, a condition of the auditability required in regulated environments.

Reconstructing the context of a past decision

Auditing a decision is not just finding that a call took place; it is reconstructing the context as it was at the moment the agent decided. Yet working memory consolidates and forgets (section 06). Observability must therefore be able to unfold a compressed memory back to the raw facts it came from, at the exact timestamp of the decision, by following the pointers deposited at consolidation. Current execution stays light; explaining a past decision stays possible, even after working memory has summarized it. This is the condition of an audit that holds in a regulated environment.

A consequence often forgotten: these traces contain prompts and outputs, hence data. Exporting them to a third-party observability service is an exfiltration like any other; they follow the same sovereignty regime as the data they carry (section 15).

Cost is steered by architecture, not by luck

Three levers, all structural, keep cost under control:

- **Model boundary**: anything deterministic does not pay the price of a model call.
- **Tier-based routing**: a task's estimated complexity selects the fast, balanced, or deep tier. In doubt, go up one tier rather than risk a bad answer.
- **Caching and prompt stability**: the stable segments of the prompt are positioned for reuse. Content stability takes precedence over token reduction alone.

! Explicit caps in continuous operation

For a system running in the background, an aggregated cost counter per root task and caps per time window are indispensable. On overrun, the orchestrator stops and returns a synthesis, rather than continuing unchecked. Recurring cost thus becomes a steered quantity, not an endured one.

KEY TAKEAWAYS

- Three independent levels: structured logs, lifecycle events, per-model-call metrics.
- A request ID propagated everywhere makes every trajectory replayable and auditable.
- Cost is steered by the model boundary, tier-based routing, and caching, plus explicit caps.
- The trace is not just a dashboard: it is the single stream that also feeds continuous evaluation and provenance.
- Traces contain prompts and outputs, hence data: same sovereignty regime; third-party export is an exfiltration.
- Auditing means reconstructing the context of a decision: unfold a consolidated memory back to its raw facts at the timestamp, via the consolidation pointers.

SECTION 11

Execution security

Guiding principle: security plays out on actions, not on display. Reads can stay open; it is mutations and side effects that are guarded.

Three-level access control

LEVEL	MECHANISM	EFFECT
Tool	Registry filtering by role, before anything is sent to the model.	The model does not know about a tool it is not allowed to call.
Data	A guard that checks ownership or scope before a mutation.	An actor modifies only what concerns it.
Query	Free reads, writes blocked for generic access (for example read-only queries).	Broad analysis possible, mutation impossible through this channel.

Agent identity

Before filtering what an agent may do, you must know who is acting. Each agent is an identified principal, distinct from the user on whose behalf it operates. It carries its own identity, at least privilege; when it acts for a user, the delegation is explicit, bounded in time and scope, never borrowing the user's full rights. The audit thus always distinguishes two things: who acted, the agent, and for whom, the user. A registry corollary: an agent acting while unknown to the registry is a security anomaly, to be treated as such. Where this identity lives, existing access management or a dedicated building block, is an infrastructure decision made on top; the principle itself does not vary.

Human approval anchored in the tool's contract

High-impact tools declare in their contract that they require approval. Before execution, the infrastructure triggers an approval request: accept, refuse, or edit the arguments. If refused, the execution never takes place. The guard does not depend on the model's discipline; it is carried by the infrastructure, and therefore guaranteed.

But an approval must not freeze a living process. If the human is not there, the agent does not stay suspended in memory monopolizing resources until the timeout. It moves into an explicit "suspended" state: its trajectory is serialized into durable state, a projection of working memory; the process frees itself; and the system rehydrates the agent when the decision arrives, minutes or hours later, exactly where it left off. This is checkpoint-based resumption (section 07) applied to waiting on a human: a suspension, not a block.

Low-urgency requests are grouped and prioritized in a queue, presented in summary form, so as not to drown the human or push them to rubber-stamp. That summary is itself an attack surface: it must be deterministic and faithful to the tool's real arguments, never a rephrasing by the model, otherwise an injected action slips into the batch and the injection reenters at the control point. Fail-closed remains intact, without paying the price of a frozen agent.

Diagram, the suspension cycle

> Suspend, do not block: fail-closed without ever freezing the agent.

Dependency isolation and exposed surface

- **Isolated sidecars**: a specialized capability runs in its own process, with restricted access and a circuit breaker on failure.
- **Ephemeral execution**: a risky execution, generated code or processing of untrusted content, runs in a disposable environment, provisioned per task and destroyed after use. What is destroyed cannot be durably compromised: the ephemeral bounds in time what a compromise can reach.
- **Guarded exposed tool surface**: the tool server the system publishes is protected by key and rate limit, and is opened beyond local only in a controlled manner.
- **Secrets and sensitive data**: isolated behind typed configuration, never in the domain. And one step further for runtime secrets: the secret never crosses the model boundary or that of the execution environment. The agent handles only a placeholder; the real key is attached by the infrastructure at the network boundary, toward the only explicitly allowed destinations, and is replaced without redeployment. A secret the model has never seen cannot be disclosed: this guard is structural, not behavioral, and it takes away injection's most coveted prize.

Untrusted inputs and prompt injection

The first-rank threat to an agentic system is prompt injection: hostile instructions slipped into the input. Its direct form comes from the user; its indirect form, more pernicious, hides in retrieved content (page, document, memory) and triggers when the model ingests it. It is therefore **intrinsic to any memory or retrieval** (§06). The worst case arises from a trifecta: access to private data, ingestion of untrusted content, and a means of communicating with the outside, gathered on a single unguarded path; removing any one of the three defuses it. Hence an operational rule, whose window is that of the context: for as long as untrusted content is present in the context, allow only two of these three properties at a time; if all three are genuinely necessary, the action does not execute autonomously, it goes under human supervision. The

window is neither the turn, too narrow since ingested content persists beyond it, nor the entire session, too broad: it opens at ingestion and closes only when the content is purged from the context.

The defense is not detection, unreliable against indirect injection, but architecture: the model is not a trust perimeter, and once untrusted content is ingested, the range of actions narrows. The baseline, non-negotiable as soon as external content is ingested:

- **Untrusted content separated from instructions**: retrieved material is treated as data, never as an order.
- **Least privilege for tools**: the model sees only what it is allowed to call (cf. access control above).
- **Human approval** on consequential actions and external communication.
- **Validation of outputs** from tools and from the model before they act.
- **Immutable journal** (§07): any action stemming from an injection remains traceable and reversible.

For high-privilege deployments, the agent is constrained further by dedicated patterns, proportionate to the risk: allow only a choice among pre-approved actions (action-selector), freeze the plan before exposing the model to tool outputs (plan-then-execute), isolate the processing of untrusted content, purge it from the context after use, or even separate a privileged model that plans from a "quarantined" model that only reads the untrusted (dual LLM). No model being immune, security remains a property of the architecture, whose depth adjusts to the risk.

Compliance by construction

The combination of access control, separation of untrusted content, human approval, complete traceability, and sovereign data handling **equips** the documentation, human-oversight, and transparency requirements expected in regulated environments: it contributes to them without substituting for them. The right to erasure is upheld without breaking the audit, by rendering a piece of data unreadable rather than breaking the immutable chain (§08). Security is not an added layer; it is distributed across the system's boundaries.

KEY TAKEAWAYS

- Security bears on actions, not display: open reads, guarded mutations.
- Three-level access control: tool, data, query.
- The agent is an identified principal, distinct from the user it serves: explicit, bounded delegation, never borrowing the user's full rights; the audit distinguishes who acted and for whom.
- Human approval is anchored in the tool's contract, guaranteed by the infrastructure.
- An approval suspends the agent (durable state) and rehydrates it at the decision, rather than blocking the process; low-urgency requests grouped in a prioritized queue.
- Isolated sidecars, guarded exposed surface, sovereign data; the runtime secret never crosses the model or environment boundary: placeholder on the agent side, the real key attached at the network boundary.
- Prompt injection = first-rank threat, intrinsic to memory: do not detect, constrain through architecture (least privilege, approval, separation of the untrusted, lethal trifecta broken over the window of the context).

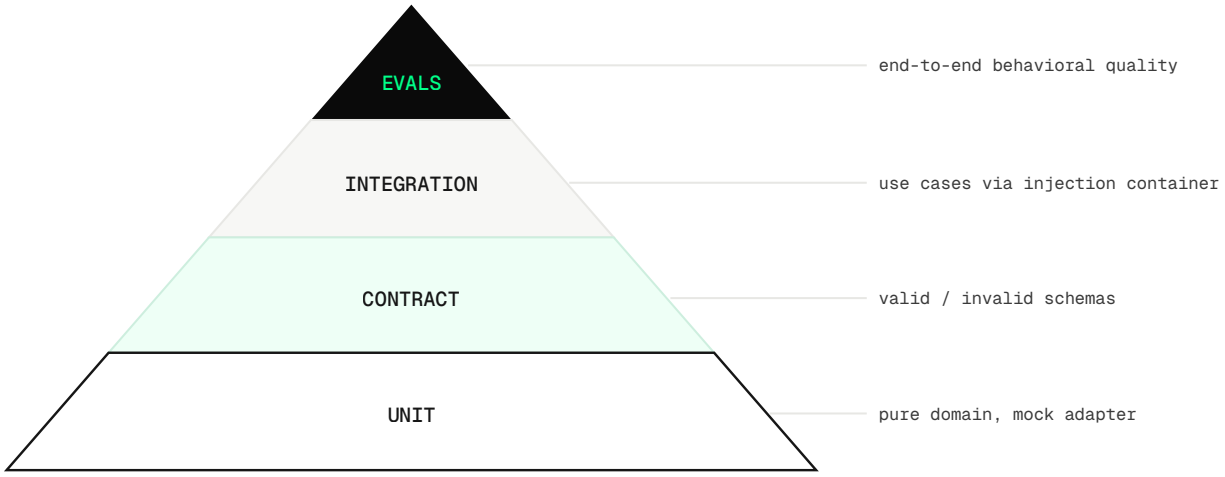
SECTION 12

Agent evaluation and testing

A deterministic system is tested; an agentic system is tested and evaluated. The code around it is verified like any software, but the model's behavior, non-deterministic, calls for a dedicated quality measure. This section is the runbook for both: the test pyramid adapted to agents, and the behavioral evaluation loop.

> Founding distinction: a test answers true or false on deterministic code. An evaluation scores a quality on non-deterministic behavior. Both are necessary; neither replaces the other.

The test pyramid adapted to agents



1 Unit, the domain without the network

Because the domain is pure and the model provider is isolated behind a port, use cases are tested without network calls, by injecting a mock model adapter that returns deterministic fixtures. The tests are fast, reproducible, and depend on no provider. The purity of the domain, established in Section 02, pays off here directly.

2 Contract, the schema as guardrail

Every input and output crossing a boundary is validated by a typed schema. Contract tests verify that a schema accepts valid payloads and rejects invalid ones: a correct payload does not throw, a malformed payload throws. Their real value is detecting the silent gap between the schema and the data actually

produced, the most costly drift in a typed system. Pushed one step further, the contract is consumer-driven: each consumer expresses its expectations as an executable contract, and a producer change that would break them fails at integration, before reaching production. It is the executable counterpart of the contract governance established in section 02.

```
valid payload      → schema.parse(payload)   does not throw   ok, contract honored
missing field      → schema.parse(payload)   throws           ok, expected rejection
wrong type         → schema.parse(payload)   throws           ok, expected rejection
```

3 Integration, orchestration with injected mocks

At the integration level, use cases are assembled via the dependency injection container, with mock adapters placed inside it (persistence, model, execution). Orchestration and boundaries are verified without touching any real infrastructure. The injection container, established in Section 05, makes these tests possible by simple substitution.

4 Evaluations, behavioral quality

At the top, evaluations measure what tests cannot: the quality of the agent's behavior on realistic scenarios. A memory evaluation bench, inspired by public long-term memory protocols, covers six capabilities, over multi-turn conversations, with an expected answer and a scoring rubric.

CAPABILITY EVALUATED	WHAT IT VERIFIES
Direct recall	Retrieving a fact given several turns earlier.
Multi-constraint recall	Combining several scattered pieces of information to answer.
Knowledge update	Returning the most recent value after a correction.
Temporal reasoning	Returning what was true at a given date, and distinguishing when a fact was learned from when it was valid. This is the test bench for the bi-temporal model of section 06.
Abstention	Refusing to invent when the information was never given.
Long session	Remembering after many turns, which triggers summarization.

Scoring follows an explicit rubric: a complete answer that restates all the key elements earns the maximum, a partial answer missing one earns less, a vague but not false answer earns half. Each scenario carries a list of expected terms and a list of forbidden terms, which makes the measurement of abstention as important as that of recall: not hallucinating is a scored criterion.

From the pyramid to the runtime loop

Placed at the end of the cycle, evaluation would amount to a mere preproduction gate. For a non-deterministic system subject to model drift, it must become a **continuous runtime loop**. The stream of traces emitted by the middleware (section 05) is sampled in the background, off the hot path: no latency added to the turn. On this sample, **hybrid** scoring combines deterministic checks with a rubric-driven **judge model** for the behavioral criteria, abstention first among them. The score feeds a **per-agent quality profile**, which extends the notion of trust from the credential alone to observed behavior. A trap to avoid: the judge model is itself a model, and it will be updated. The day it changes, the quality series over time becomes incomparable, silently. So the judge is anchored, just as the serving model is promoted: a pinned version, or an anchor set replayed at every judge change to recalibrate the measure.

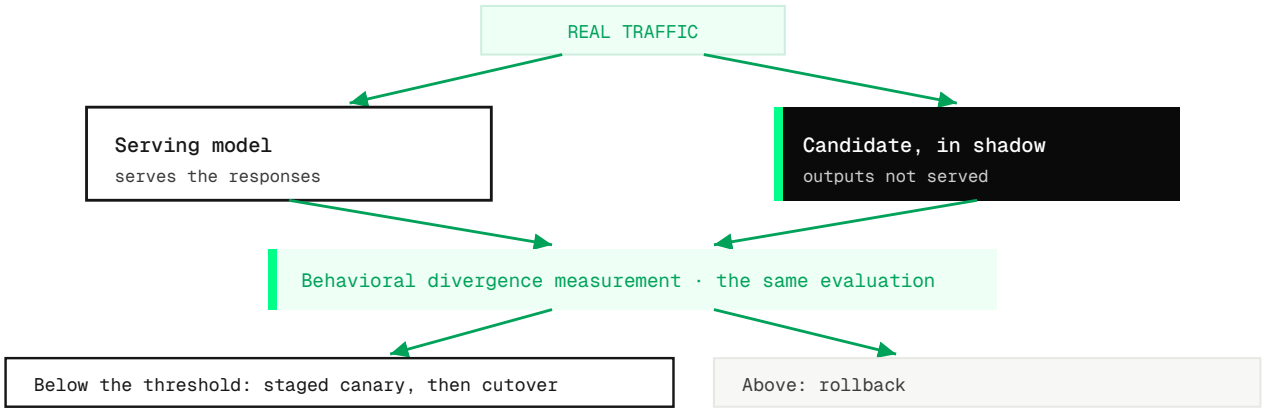
The loop remains **open at the action boundary**. It produces signals and recommendations; acting on them goes either through bounded tuning within a pre-approved, audited envelope, or through human validation, never through autonomous rewriting. A naive closed loop that directly optimized the judge's score would drift toward whatever flatters it without improving real quality; what makes self-tuning hold is a tamper-proof guard, a verifiable hold-out the optimizer never sees and that triggers a rollback. This strong guard assumes verifiable ground truth; for purely subjective behaviors, where it is missing, the safety net is once again the human sample, and the self-tuning envelope stays narrow. The principle "the human decides" applies all the way here.

Why not a chorus of verifiers? The temptation is strong to replace deterministic guardrails with probabilistic agents that review one another, and to let self-correction emerge. It is seductive, but weaker wherever a guarantee is required, for three reasons. First, a hard property, an authorization, idempotence, schema conformance, cannot be manufactured from soft judgments: stacking fallible verifiers stacks uncertainties, it does not cancel them. Second, the argument assumes independent errors, yet models that share data and biases share their blind spots: their errors are correlated, to the point that a majority vote can amplify the common error instead of correcting it, and an adversarial input that fools one fools the chorus. Finally, every verification is one more inference, costly and slow, where a deterministic check is free and safe. This is the category error the model boundary avoids: deterministic for whatever admits a guarantee, judge model for the only irreducible behavioral quality, and always under the tamper-proof hold-out described above. As models grow more reliable, the share entrusted to the judge can grow; but the floor, safety, security, authorization, audit, never leaves the deterministic. And it must be said plainly: the judge this framework retains does not escape these limits, fallible and non-deterministic itself. It holds only under the tamper-proof hold-out and behind the deterministic floor, never as a standalone guarantee. The behavioral assurance layer is useful; it is not as hard as the foundation.

Validating a model change before trusting it

Substituting the model is a right the architecture guarantees; trusting it is earned. A new or updated model can alter behavior in ways no offline evaluation bench captures, and that only shows on real traffic: so the switch is never made in a single move. The candidate first runs in **shadow deployment**, behind the same port, on real traffic but silently: its outputs are not served, and their behavioral divergence from the serving model is measured by the same evaluation that monitors drift. If the divergence stays below a threshold defined in advance, a growing share of traffic is switched over, under the same monitoring, with immediate rollback at the slightest regression. This is the migration discipline of the model port, sister to the expand-then-contract of contracts (section 02): validate in parallel, then switch in stages. One reservation, however:

this automatic gate holds only for criteria with verifiable ground truth. For purely subjective dimensions, where the automatic signal is missing, promotion does not self-trigger; it stays under human validation.



Evaluation runbook

- **Add a scenario**: describe the multi-turn conversation, the final question, the expected and forbidden terms, and the targeted capability category.
- **Run**: replay the turns in order against the current system, ask the final question, compare the answer against the expected and forbidden terms.
- **Score**: apply the rubric, track the score per capability over time to detect behavioral regressions.
- **Trigger the evaluations**: rerun the evaluation bench on any change touching memory, the prompt, or routing, wherever a behavioral regression is possible, and track the score per capability over time.

KEY TAKEAWAYS

- Test the deterministic code, evaluate the non-deterministic behavior: both are necessary.
- Pyramid: unit without the network (mock adapter), schema contract, integration via the injection container, evaluations at the top.
- Contract tests detect schema-versus-data drift, the most costly silent defect; consumer-driven, they prevent a producer change from silently breaking a consumer.
- Evaluations cover six capabilities, including temporal reasoning (which exercises the bi-temporal model) and abstention: not hallucinating is a scored criterion.
- Evaluation is not a final gate but a continuous runtime loop, sampled off the hot path, fed by the trace stream.
- Hybrid scoring (deterministic plus judge model), per-agent quality profile; self-tuning stays bounded, guarded by a tamper-proof hold-out and human override.
- A model change is validated in shadow deployment then in stages, divergence measured by the same evaluation, never by a simple cutover.
- A chorus of probabilistic verifiers does not make a guarantee: their errors are correlated. The deterministic holds the floor (safety, security, audit); the judge model covers only the irreducibly behavioral.

PART IV 3 SECTIONS

MATRIX · SYNTHESIS · EVIDENCE IN CODE

Decision and synthesis

Deciding: the summary decision matrix, the synthesis of recommendations, and evidence in code.

IN THIS PART

SECTION 13

Summary decision matrix

Everything above fits in one decision table. Each row is a trade-off, with its recommended default and the signal that commands a change.

The trade-off matrix

DECISION	RECOMMENDED DEFAULT	EVOLVE WHEN
Core language	TypeScript (orchestration and interface)	Never without a reason; polyglot goes through a sidecar or a service.
Specialized capability	Sidecar in the library's language	As soon as a mature library lives in another ecosystem.
Component on the critical path	Compiled service behind a contract	Latency or throughput proven insufficient.
Decomposition	Hexagonal modular monolith	Load, lifecycle, isolation, or high availability demand it.
Integration with legacy systems	Anticorruption layer behind the port	Progressive replacement, feature by feature, never in one block.
Number of agents	Single orchestrator directing specialists	Parallelizable complexity, never for sequential reasoning.
Model abstraction	Single port, three tiers, direct SDK	Never a heavy chaining framework by default.
State	Stateless agent, immutable journal plus derived working view	Multi-instance: externalize the in-memory structures.
Crossing the process boundary	As late as possible, on an objective trigger	Then accept eventual consistency, compensation, and explicit causal order.
Contract evolution	Versioned, additive, read with a tolerant reader	Breaking change: expand then contract, under consumer-driven verification.

DECISION	RECOMMENDED DEFAULT	EVOLVE WHEN
Memory and forgetting	Value score, invalidation (not deletion), consolidation with pointers to the source facts	Physical deletion reserved for invalidated derivatives under retention.
Evaluation	Tests, plus a continuous, guarded evaluation loop	Bounded self-tuning only, guarded, under human override.
Multi-agent budget and discovery	Central aggregated counter, in-memory registry	Block allocation and shared capability cards as soon as you leave the process.
Model unavailable	Multi-provider failover behind the port	Available from real go-live onward.
Changing the model (new or updated)	Validate in shadow deployment, then in stages	Never in a single move; rollback at the slightest divergence.
Waiting for a human approval	Suspend and release (durable state), rehydrate on decision	Never block the process; low-urgency requests batched in a queue.
Non-critical service	Fail-open, signaled degraded mode	Sensitive action: explicitly fail-closed instead.
Delegation autonomy	Bounded: templates, budgets, approvals	Widen only if the outcome actually improves.
Untrusted inputs	Retrieved content untrusted, least privilege, approval on the action	High privilege: dedicated patterns (frozen plan, quarantine of the untrusted).
Agent identity	Its own principal, distinct from the user served; explicit delegation, bounded in time and scope	Never borrowing the user's full rights; an agent outside the registry = a security anomaly.
Runtime secrets	Placeholder on the agent side, real key attached at the network boundary, allowed destinations	Never a key in the model's context or in the execution environment.
Shared building block (connectors, skills, cross-application memory, delivery)	Consumed via a port; adapter = discovery client via an index (section 15)	The index stays an index; delivery delivers; the orchestrator decides.

Quick decision tree

Does the capability depend on a mature library in another language?

- YES — sidecar in that language
- NO — stay in the TypeScript core

Does a component have its own load, lifecycle, or isolation?

- YES — extract it as a service
- NO — stay in the modular monolith

Is the task parallelizable and highly complex?

- YES — several agents under one orchestrator
- NO — single orchestrator, one single thread

Must the load be carried across several instances?

- YES — externalize the state, then replicate
- NO — single instance, in-memory state ok

KEY TAKEAWAYS

- Every decision has a sober default and an explicit evolution trigger.
- The cross-cutting rule: start simple, isolate behind contracts, add complexity only on evidence.
- None of these decisions is irreversible, because all of them are made behind a boundary.

SECTION 14

Synthesis and recommendations

In the end, everything comes down to one posture and five engineering moves.

1 Set the boundaries before the stack

Define the contracts (ports) and the integration protocol first. Language and topology then become local, reversible choices, made in the right place, not initial bets.

2 Start simple, isolate behind contracts, add complexity on evidence

Modular monolith, single orchestrator, single model port. A service is externalized, an agent is added, or a model tier is raised only when an objective criterion or a measured gain justifies it.

3 Keep the deterministic out of the model

Classification, routing, validation, idempotence, and guardrails stay in testable, cacheable code. The model keeps its budget for what it alone does better.

4 Make state explicit, and memory governed

An agent with no hidden state. The event journal is immutable, for audit and replay; working memory is a derived view of it, which forgets through a value score, invalidates rather than deletes, and cools off into archive. In-memory structures are externalized in order to replicate: scaling becomes a state migration, not a rewrite.

5 Govern, trace, and evaluate from day one

A single middleware chain for logging, access, cost, and approval; a request identifier propagated everywhere; multi-provider failover and degraded mode for resilience. The same trace feeds a continuous, guarded evaluation, which monitors behavioral drift without ever letting the system rewrite itself. Trust is built through instrumentation.

Recommendations for technical teams

ACTION	HORIZON
Set the ports and the integration protocol before any stack choice	Day zero

ACTION	HORIZON
Start as a modular monolith with a single orchestrator	Day zero
Wire up logging, metrics, request identifier, and approvals	Before any go-live
Externalize state as soon as multi-instance is in sight	On a load trigger
Wire a continuous, guarded behavioral evaluation into the trace stream	Before any go-live
Introduce sidecars, services, or additional agents on objective criteria	On proven need

Overall position

Running a reliable agentic system is not a framework problem or a language problem; it is a problem of **architecture around the model**. With the boundaries set first, the full-TypeScript-or-polyglot choice and the monolith-or-microservices choice cease to be structuring dilemmas: they are local, reversible trade-offs, made behind stable contracts. And since the model itself lives behind a port, it remains a component to be replaced, up to its very paradigm: what compounds as capital is not the model, it is the architecture that frames it.

What this framework establishes, it demonstrates: the principles are not speculative, they live in three independently built systems that converge without sharing code (appendix). For validating an architecture, this is the strongest evidence one can provide. What belongs to another discipline, organizational adoption (who owns what, lifecycles, decision bodies) and enterprise data strategy, is not a blind spot of the architecture but a distinct layer, which conditions every project whatever its architecture. Conflating them would amount to blaming a structural drawing for not being a financing plan. The architecture itself holds, and it is proven.

The posture fits in one sentence: **start simple, isolate behind contracts, add complexity only on evidence**, keeping state explicit, governance by design, and sovereignty by construction. It is the operational translation, for teams, of a simple conviction: architecture first, model second.

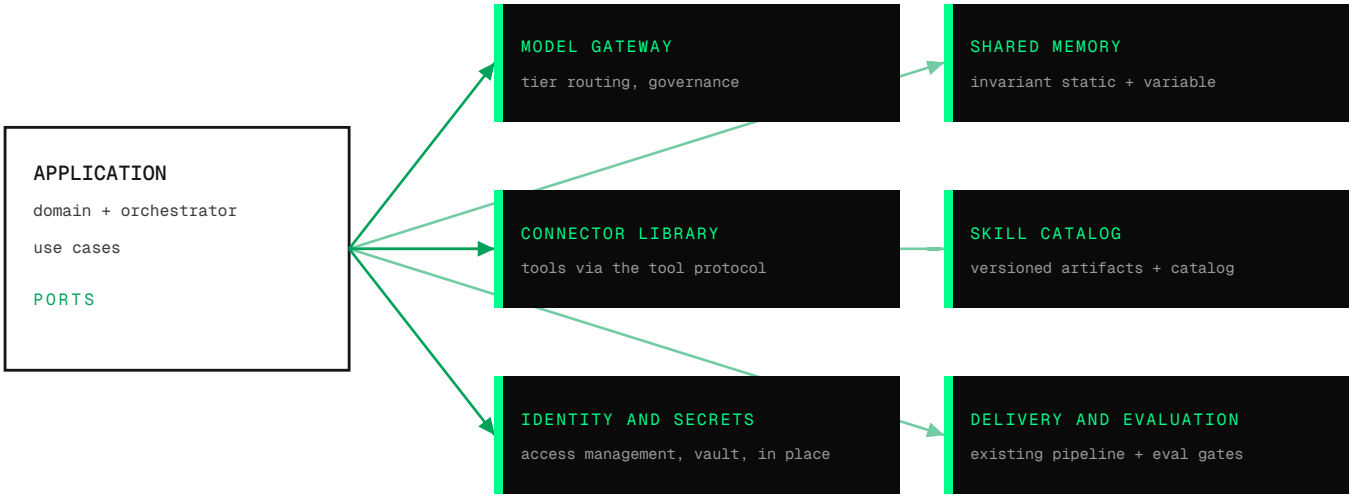
SECTION 15

Beyond the application, the shared building blocks

Everything above addresses one application: a domain, its ports, its adapters. But some capabilities are the adapter of no application in particular: they are shared by several, by nature. A connector library, a skill catalog, a cross-application memory, a delivery and evaluation chain, an agent identity: these are enterprise building blocks. This section fixes their principles; the landscape of locations and offerings, shifting by nature, is deliberately left outside these pages, which retain only what holds whatever the vendors.

The shift, and the rule that does not move

Housing a shared building block inside an application condemns it to duplication; housing it elsewhere does not leave the model. The block leaves the application, but the application still consumes it through a port, a stable contract; what changes is that the adapter behind that port points to an enterprise service instead of a local structure. The question is therefore never port or no port: it is which location behind the port, existing infrastructure, dedicated platform, or managed service. This location trade-off is reversible precisely because the contract itself does not move.



The block leaves the application; the application still consumes it via a port, a stable contract.

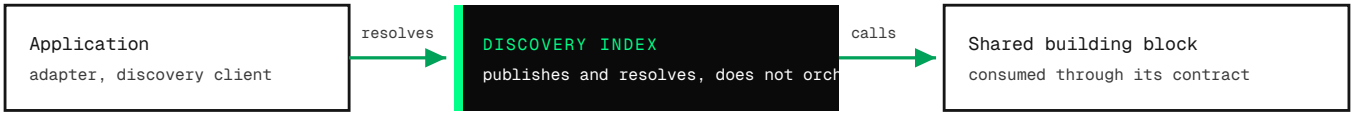
The locations, and the criteria that decide

Behind each port, five location families. In the application itself: a local adapter, simple but duplicated as soon as a second application appears. Existing infrastructure, API gateway, artifact repository, integration pipeline, access management: sovereign and amortized, but not designed for the agentic. The dedicated

internal platform: control and reuse, at the price of building and operating it. The managed service from an infrastructure provider: mature and fast, less sovereign, dependent. The managed runtime from a model provider: maximum velocity and the strongest dependency, the one on the supplier of the central component, with partial sovereignty when tool execution stays in-house. Six criteria decide, and they alone are stable: sovereignty and control, reuse across applications, implementation velocity, cost and dependency, reuse of the existing estate, compliance. The grid is durable; filling it in depends on a dated landscape and context, deliberately outside these pages.

The recursive motif: contract, index, implementation

At the scale of one application, the adapter is local. At the scale of an infrastructure, since the block is shared, the adapter becomes a thin client that resolves the capability through a common index, then calls it by its contract. It is the direct extension of the two-adapter registry (section 05): the motif is recursive. Three things remain distinct and must never merge: the contract, which states what the block promises; the index, which publishes and makes discoverable; the shared implementation, which honors the contract. It is not the adapters that become the index: it is their discovery that goes through an index.



Contract, index, implementation: three distinct things. The index is not a brain.

The guards, amplified at scale

Two guards of this doctrine apply as they stand and become more critical at platform scale. The first: the index, or the catalog, stays an index, not a brain; it publishes and makes discoverable, it does not orchestrate, on pain of recreating the bloated integration bus (section 05). The trap takes a precise form here: letting the delivery chain or the catalog decide at runtime which skill to call. Delivery delivers; the application's orchestrator decides. The second: a third-party skill or connector is an untrusted input; least privilege, approval on the action, and the triad breaks on the context window (section 11). A shared catalog amplifies this surface: curation and security review are not optional there.

The durable execution engine, one adapter among others

When turns grow long, waits, recoveries, compensations, a durable execution engine can carry these properties as a shared building block: step journal, retry, suspension and wake-up. It remains an adapter behind the ports, and the guard is the same as for the index: it carries control flow and durability, never reasoning or business logic. The model call remains an activity whose output is recorded: it is reread, not replayed (section 07). Letting the orchestration engine decide business logic would, here again, amount to rebuilding the bloated bus.

The building-block matrix

As with the decision matrix of section 13, each block has a sober default and an explicit trigger; the default applies as long as no trigger commands a change.

BLOCK	RECOMMENDED DEFAULT	SWITCH WHEN
Model gateway	The existing API gateway, seen as the model port	Agentic functions (tier routing, cost governance) outgrow it.
Connector library	Existing infrastructure carries transport and security	It caps out on tool semantics and discovery: dedicated or managed block.
Skill catalog	Existing artifact repository for the versioned package	Curation and security review demand it: dedicated catalog, never without curation.
Shared memory	In the application for working memory	A real cross-application need, not a distant target.
Discovery registry	In-memory table (section 05)	Crossing a process boundary: shared registry (section 07).
Durable execution	Simple loop in the application	Long turns, approval waits, recovery after failure: durable engine, which never carries business logic.
Delivery and evaluation	Existing integration pipeline plus evaluation gates	Guarded continuous evaluation (section 12) outgrows the pipeline: dedicated evaluation block.
Observability	Existing stack for trace transport	Regulated data: self-hosted, because traces are data.
Identity and secrets	Existing access management and vault	Agent identity and bounded delegation required (section 11): dedicated block.

Sovereignty is graduated by class of data

Sovereignty is not a switch, it is a gradient: it is decided per class of data, not wholesale. The rule that follows: sovereign by default wherever sovereignty costs only operations, gateway, memory, registry, execution, access control; the only tier where it can cost quality is the model, and the model port makes precisely its sensitivity-based routing local and reversible, best available engine for ordinary data, sovereign engine for regulated data. A consequence often forgotten: execution traces contain the prompts and the outputs, hence data; they follow the same gradient, and exporting them to a third-party service is an exfiltration like any other (section 10).

Compliance consumes the architecture

Regulatory compliance is not a separate workstream: it is a layer that consumes the primitives already in place, immutable logging, human oversight, least privilege, memory governance and erasure, provenance. One extension is enough to equip it: the discovery registry extends into a usage registry, which carries, per deployment, the risk class, the data classes, and the owner, under an identified responsible party and a periodic review. Because risk is classified per deployment, not per platform: the same foundation, reused for a sensitive use, tips that particular deployment into a more demanding regime, without the architecture changing.

The lesson of the layers

A history lesson, to close. Infrastructures sediment: each generation deposits a layer of abstraction on top of the previous one, and every layer ends up commoditized while a new one stacks above it. At each transition, the systems that consumed the layer below through a contract crossed it; those that had welded themselves to it paid dearly. The building blocks of this section will not escape the rule: they will be commoditized in their turn. That is precisely why one consumes them behind ports and does not weld to them: the discipline of contracts does not only protect against a change of vendor, it carries you across the layers themselves.

KEY TAKEAWAYS

- Some building blocks are shared by nature; consumption stays through a port, the question is the location behind the port, and that trade-off is reversible.
- The motif is recursive: the adapter becomes a thin client that resolves via an index; contract, index, and implementation remain three distinct things.
- The index is not a brain; delivery delivers; the orchestrator decides; a third party is an untrusted input, curation is not optional.
- The durable engine carries control flow, never reasoning; the model call remains a reread activity.
- Five location families behind each port; six stable criteria decide, filling them in is a dated exercise.
- Each block has a sober default and a switch trigger: the building-block matrix extends the decision matrix.
- Sovereignty is graduated by class of data; only the model tier can cost quality; traces are data.
- Compliance consumes the architecture; the usage registry is its centerpiece, and risk is classified per deployment.
- Every layer ends up commoditized: consume it through a contract, do not weld yourself to it; the discipline of ports carries you across the transitions.

APPENDIX

Evidence in code

This appendix grounds the principles set out here in real artifacts. They live in three systems built separately, with no code sharing: a chain of test agents born of several experiments (system 1), a project-support platform built on a knowledge graph (system 2), and a personal assistant with long-term memory (system 3). The first two cover the full set of pillars and are set side by side below; the third, specialized, exercises the memory and evaluation pillars. That these three systems converge on the same patterns, without sharing a single line of code, is a serious clue. Let us stay precise about its scope: it is validation by use, not proof of independence, since the same hand designed them. Absence of shared code does not equal independence of design.

> The artifacts below are described by their function, not by their location. What matters is not the name of a file but the fact that the block exists and that the systems converge on it, verifiable by a team with access to the repositories. No path, file name, or commercial name is cited.

Foundation, boundaries, and providers

PRINCIPLE	SYSTEM 1, TEST AGENTS	SYSTEM 2, PROJECT PLATFORM
Pure domain, ports and adapters	Pure domain and use cases, secondary adapters separated from the core	Domain ports isolated, dedicated persistence adapters
Explicit dependency injection	Dependencies passed by constructor, centralized factories	Dependency injection container with a single entry point
Model boundary, tiered port	Model provider port, three tiers fast, balanced, deep	Model adapters behind a complexity router
Routing by model tier	Tier declared per agent, escalation in case of doubt	Task classified as simple, moderate, complex
Feature detection, bimodal	Detection at startup, minimal, full, integrated modes	Per-service detection, graceful degradation if absent

Orchestration, tools, and governance

PRINCIPLE	SYSTEM 1, TEST AGENTS	SYSTEM 2, PROJECT PLATFORM
Orchestrator without business logic	Orchestrator use case that composes, with no business logic of its own	Thin orchestration entry point, delegating to the registry
Tool registry by assembly	Tools assembled and exposed by a dedicated builder	Single tool registry, centralized registration and resolution
Cross-cutting middleware chain	Tracing and approval envelope around every tool	Logging and metrics middleware plus a data access guard
Tool filtering by access	Contextual exposure of tools to the model	Tool selection by access level, upstream
Human approval on the action	Approval request declared in the tool's contract	Data access guard before any mutation
Guardrails and budgets	Safety net around the tools, counters, rate limit	Rate limits, read-only access for generic queries

Execution, state, and observability

PRINCIPLE	SYSTEM 1, TEST AGENTS	SYSTEM 2, PROJECT PLATFORM
Specialized capability as a sidecar	Execution engine driven via the tool protocol, dedicated port	External tool servers via a dedicated client
Exposed and guarded tool surface	Published tool server, protected by a dedicated guard	Tool server integration, guard and limits
Observability through events	Lifecycle events traced, trace export	Metrics middleware, structured logs
External state and topology	Persistence behind its port, in-memory structures identified	Dedicated real-time state and migrations, orchestrated deployment

! Dynamic delegation is already under way

A notable point for the trajectory toward adaptive orchestration: in system 1, the orchestrator already has dynamic delegation building blocks. There is a sub-agent invoker, sub-agent templates, parallel and template-based delegation, a procedural memory that records and replays routines, and a behavior

evolution subject to human validation, still held in to-review status. Tier 3 of maturity is therefore not a distant target: its foundation already exists.

! Memory, evaluation, budgets: where the evidence lives

The three reinforcements of this edition also rest on existing code. For memory (section 06), the third system, dedicated to long-term memory, already applies non-destructive forgetting: a periodic task updates the retrieval score without deleting memories, only derived links are pruned below a threshold, and importance as well as salience are produced as a by-product of asynchronous extraction, with no dedicated model call; validity over time, two timestamps per fact, is its natural extension. For evaluation (section 12), system 1 already has a memory evaluation bench, a trace export, a reliability feedback loop whose action remains deliberately manual, and a per-agent confidence score: the guarded runtime loop is their culmination. For distributed delegation (sections 04, 05, 07), system 1 already signs agent cards and caps the budget with an aggregated ceiling in autonomous mode; discovery by cards and block-based budget allocation are the distributed adapters of those same ports.

KEY TAKEAWAYS

- Each principle set out here corresponds to a real artifact, in three systems built separately.
- Their convergence without code sharing is strong evidence by use; not proof of independence, the same hand designed them.
- The dynamic delegation of tier 3 is already partially implemented, not merely projected.

Afterword

This document began with an engineer's worry: almost everything that calls itself agentic does not hold up in production. At the end, what remains comes down to an inversion. The most impressive component in our systems is also the least reliable; the whole edifice consists of treating this marvel the way one treats a fallible component, behind contracts, under guards, with evidence. That is not skepticism, it is the opposite: we only discipline with such care what we intend to use for a long time.

Two lessons follow, one of humility and one of confidence. Humility: almost nothing here is invented. The plumbing comes from decades of distributed systems, and the temptation to overcome was not technical; it was the urge to reinvent everything because the component is new. Confidence: the four genuine novelties have architectural answers, not incantations. A reliable agentic system is not a feat, it is a discipline.

These pages do not run. A doctrine ships nothing: the logical next step, for whoever closes them, is not to debate them but to lay down an executable floor, wired to a real flow, and let it prove itself. Everything that precedes is ordered toward that: what you lay down on day zero amounts to very little, and the rest waits for its trigger. Nor is governance a chapter added at the end: it is already in the boundaries, and it begins on day one or never.

This document also applies its own rule to itself. It was written the way one maintains a system: every position verified against what exists, tested against the state of the art, locked in by a traced decision before being written. It will be revised the same way, on evidence, never by fashion. It is a living document; if it ever stops being true, it must be corrected, not defended.

The challenges remain, and it would be dishonest to close without naming them. The engines will grow more reliable, and the boundary of what we delegate to them will widen; it will widen through guarded steps, or it will be paid for. The floor, for its part, will not move: safety, authorization, and audit will never leave the deterministic. And the hardest work is not in these pages: it is adoption, who owns what, who answers for what, who learns what. It is human before it is technical, and no architecture spares you from it; it can only make it possible.

If only one sentence were to be kept, it would be the one that opened this document and can close it: architecture first, model second. Start simple, isolate behind contracts, add complexity only on evidence. The rest is execution, and execution is the only judge.

THIBAULT SOURIS · 2026

Architecture first, model second.

Nearly every application calls itself agentic; almost none holds up in production. The gap is not a model problem, it is an architecture problem.

These pages lay out the decision framework that makes the difference: where to draw the boundaries, how to make language and topology reversible, how to keep state explicit, govern memory, secure untrusted inputs, and evaluate nondeterministic behavior. From the first port laid down on day zero to the loop that watches for drift, each section turns an initial bet into a reversible decision, made behind stable contracts. The model remains what it must be: a component you replace, down to its paradigm, while the architecture compounds.

For anyone who designs, runs, or defends such systems, and wants to do so on foundations that last.

Thibault Souris.