Konstera

GUIDE

AI Agent *Architecture Guide.*

Ten architecture decisions that separate a production agent runtime from a weekend prototype. Written from the choices we made in Konstera, with the trade-offs and alternatives we considered. If you're building your own, use this as a design-review checklist.

20-minute read · 10 sections · Konstera · 2026 · For CTOs & platform engineers

GUIDE

Contents

01 Who this is for
02 MCP as the agent protocol
03 Containerised runtime
04 Multi-tenancy done at the data layer
05 Autonomy as policy, not code
06 State management
07 Observability and audit
08 Cost attribution per invocation
09 Security and credential handling
10 Multi-agent coordination patterns

01 Who this is for

This guide is written for CTOs and platform engineers who need to decide — within the next quarter — whether to adopt an AI agent platform or build a thin layer internally. The practical answer depends on ten architecture decisions that compound. Get them wrong early and you'll rewrite in year two; get them right and you'll ship your fifth agent as cheaply as your first.

Each section covers one decision: the trade-off, our choice in Konstera, the alternatives we considered, and when each one is correct. If you're building, treat this as a design-review checklist. If you're buying, treat it as the architecture contract you're signing up to.

---

02 MCP as the agent protocol

Pick one wire protocol for how agents talk to the platform and commit to it. Our choice: the Model Context Protocol (MCP). The agent is an MCP server; the platform kernel is an MCP client for agent-facing services (llm, vector, state, credentials, cost, log, tool, human, prompt).

Why MCP matters architecturally: it decouples agent authoring from kernel implementation. Any language that can speak MCP over stdio or HTTP becomes a valid agent language. We ship first-class SDKs for .NET and Python (TypeScript follows), but nothing in the kernel assumes a specific SDK.

Alternatives we considered: a bespoke JSON-RPC schema (would have shipped faster but forked the ecosystem); gRPC with protobuf definitions (type-safe but adds a code-gen burden to every agent language). MCP wins because it's becoming the emerging standard for agent↔tool wire protocols — Anthropic introduced it, the ecosystem is converging, and we don't have to be the ones standardising it.

---

03 Containerised runtime

Agents ship as OCI container images. The kernel spawns one container instance per invocation; the container exits when the invocation completes or times out. No long-running agent processes.

Three reasons this is the right default:

- **Blast radius.** An agent cannot leak state to the next invocation because the next invocation is a fresh container.
- **Resource bounding.** Memory, CPU, timeout, network scope — all enforced at the container level.
- **Language freedom.** If it builds to a container, it runs. We don't care if your agent is compiled, interpreted, or shelled-out.

Alternatives: in-process agents (shared Python/Node runtime — faster cold-start but blast radius becomes a nightmare); dedicated long-running agent processes (lower per-invocation cost but complicates tenant isolation). Container-per-invocation wins on operational predictability.

04 Multi-tenancy done at the data layer

Multi-tenancy is a database-layer concern, not an application concern. The temptation to filter by `tenant_id` in your ORM queries is overwhelming and wrong — one missed filter leaks across tenants.

Our choice: PostgreSQL Row-Level Security (RLS) on every tenant-scoped table. The platform's database role runs as `NOBYPASSRLS`. Every connection sets a session GUC (`app.tenant_id`) via `SET LOCAL` before the first query. RLS policies filter `tenant_id = current_setting('app.tenant_id')::uuid`. Cross-tenant admin operations use a separate connection pool with a role that has `BYPASSRLS`.

Trade-off: slightly more complex ORM configuration (every query needs the GUC set in advance). Benefit: a missed filter becomes impossible, not a code review question.

05 Autonomy as policy, not code

Autonomy — what runs without a human, what requires reviewer sign-off, what requires named-approver approval — is policy. It cannot be code, because the business will want to change it every month, and redeploying every month is a tax.

Our model: a scope-stacked policy configuration (binding → agent → tenant → platform). Each layer can override the one below. The platform ships a sensible default; the tenant admin can tighten; per-binding overrides handle the edge case where one business unit needs different autonomy than another.

The hard part is making this editable without a redeploy. The agent reads its current policy from the `human` MCP service at invocation time, not from an environment variable baked into the container.

06 State management

Three kinds of state live inside a running agent system, and each needs a different store.

- **Workflow state** — "where are we in this invocation." This is Temporal's job. Every agent invocation is a Temporal workflow; retries, timeouts, cron, webhooks, HITL signals, and sagas are first-class rather than hand-rolled per team.
- **Durable per-agent state** — "what have we learned." Lives in a per-agent, per-tenant key-value store exposed as the `state` MCP service. Not visible to other agents.
- **Context / knowledge** — "what do we know about the business." Lives in the context graph + vector store, surfaced as the `vector` MCP service. Shared across agents by design — this is where cross-agent compounding happens.

Keeping these three categories separate is one of the quiet architectural wins. Confusing them produces agents that either leak context (shared state as kv store) or can't build on each other (per-agent knowledge siloed).

07 Observability and audit

Three signals, three different consumers:

- **Platform observability** — OpenTelemetry traces, metrics, logs exported via OTLP to the customer's existing APM stack. Standard SRE kit.
- **Agent audit trail** — every invocation reconstructable: inputs, model, prompt, every tool call, reviewer decisions. Stored in platform tables with 90-day default retention, exportable to SIEM. This is what regulators and incident responders need.
- **Business metrics** — reviewer approval rate, time-to-decision, cost per outcome, deflection or capacity shift. Surfaced in the portal and exportable to the customer's BI stack.

Replay is the subtle part. Any past invocation can be re-run against a new agent version — critical for "would the fix have worked on last Tuesday's failed case" before deploying.

08 Cost attribution per invocation

Every LLM call flows through the `llm` MCP service, which populates a `usage_events` row with: tenant, agent, invocation, user (where applicable), model, prompt tokens, completion tokens, and provider price-at-call-time. Never let an agent call a provider SDK directly.

Why not: direct provider calls skip metering, skip tenant scoping (keys could cross boundaries), and lose the audit trail. The one centralising layer is table stakes for cost control and governance.

Attribution matters more than totals. "Our monthly LLM spend grew 40%" is useless at board. "Our customer-service agent on Claude Sonnet grew 60% as volume doubled, while our forecast agent on GPT-5 stayed flat" is a decision input.

09 Security and credential handling

Credentials — API keys, OAuth tokens, DB connection strings — live in the credentials MCP service, encrypted at rest, scoped per tenant. The agent asks for a credential by name; the service resolves it from the current tenant's store.

BYOK for LLM keys is non-negotiable. The customer contracts with Anthropic / OpenAI / Google directly. The platform meters usage; the LLM bill goes to the customer's account at the LLM vendor. This matters for data flow (the customer's inference traffic goes to their own vendor account) and for commercial

clarity (no vendor markup).

Container-level security: no inbound network except from the kernel; no outbound network except to allowlisted hosts (LLM providers, declared connectors); no shared filesystems; no shared memory. Blast radius bounded even if an agent is actively malicious.

---

10 Multi-agent coordination patterns

Multi-agent is where most agent platforms either over-promise or under-deliver. The honest reality:

- **Explicit coordination works.** Agent A calls Agent B via `ctx.agents.invoke("B", ...)`. The call becomes a child workflow with its own audit trail. Latency and cost are predictable.
- **Implicit swarms are a research project.** "Autonomous agents that figure out how to work together" is a 2027+ problem for production deployments. We've built explicit; implicit is a research track that occasionally produces papers.
- **Shared context, not shared state.** Agents compound by reading the same context graph (customer identity, product catalogue, policy knowledge). They do not coordinate by writing to the same KV store. Confusing the two causes nondeterminism.

If you're evaluating a platform that pitches "agent swarms" as a current feature, ask for the audit trail format for a 10-step chain. The answer will tell you how serious the platform is about production operations vs. conference demos.

Konstera

One platform for every AI agent in your business.

Governed centrally, costed in one ledger, built on shared context — with people in control of every decision that matters.

konstera.ai · konstera.ai/book-a-call · konstera.ai/resources