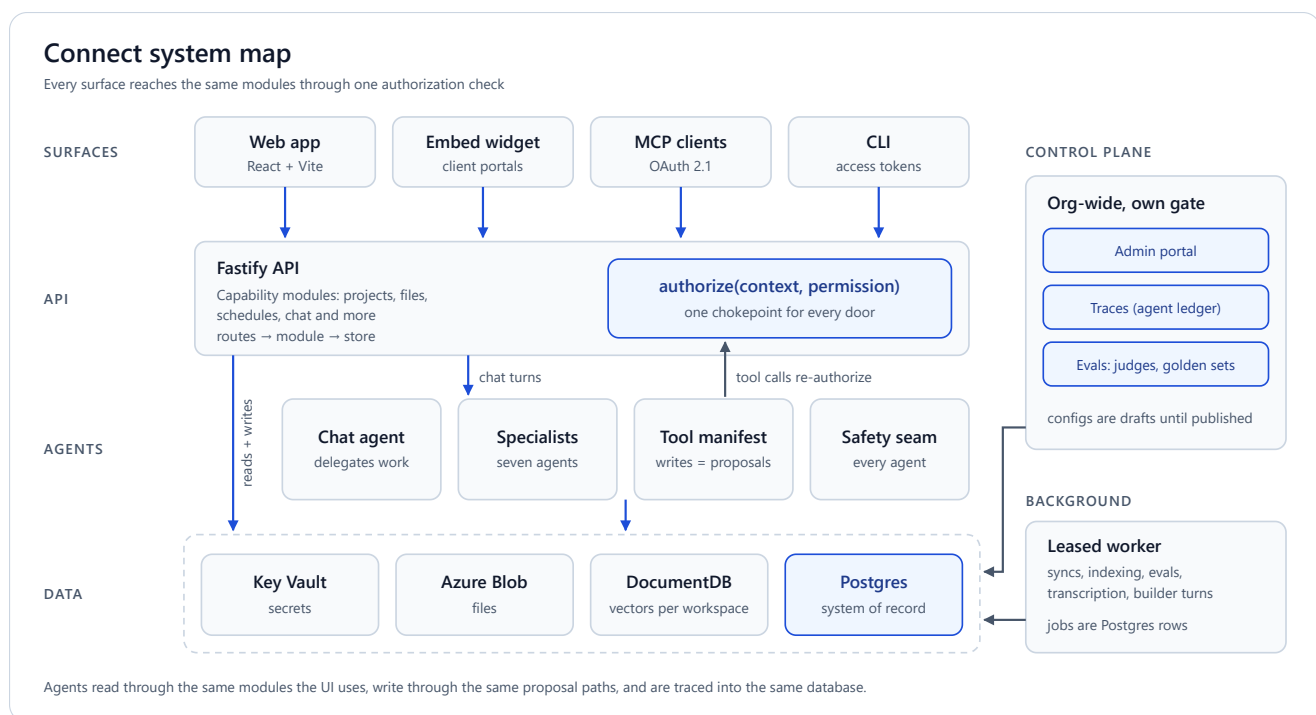


The Enterprise AI Platform Blueprint: A Checklist from Connect

A practical, reusable checklist for CTOs and operations leaders planning their own agentic platform, drawn from building Connect, with every item linked to the article that explains it and a recommended sequence for time to value.



Most teams planning an agentic platform start with the agent. They pick a model, wire up a chat, and get a convincing demo in a week. Then the real questions arrive: who is allowed to see which project, what happens when the agent writes something wrong, how anyone finds out what it did last Tuesday, and how to ship a prompt change without breaking the thing that worked yesterday. Those questions decide whether the platform survives contact with a real company.

This checklist comes from building Connect, the agentic platform we built on top of Syncify for CPM scheduling. Every item links to the article in our [architecture series](#) that explains how we did it and what it cost. Use it as a planning tool: copy it into your own tracker, cross out what does not apply, and argue with the rest. It is written for CTOs, engineering leads and operations leaders at contractors and construction-tech companies, but very little of it is construction-specific.

Questions to answer before you build

Write down answers to these before anyone opens an editor. Vague answers here become expensive rewrites later.

1. **Who is the tenant?** A company, a business unit, a project? Your answer becomes the key on every table and every vector index.
2. **What is the unit of access?** In Connect a person is added to a project with a role. Decide whether access is granted on the project, the program, the document, or something else, and make it one thing.
3. **What are all the doors?** Web app, mobile, an embedded widget in a client portal, an API or CLI, MCP for external agent clients, and every agent tool. Each is a way into the same data.
4. **What may an agent change on its own?** Ideally almost nothing. List the systems of record and decide which writes need a person to approve them.
5. **Where does the evidence live?** If an agent states a fact, can a reviewer click through to the source document? Decide now whether citations are required.
6. **Who reviews agent quality, and with what?** Someone needs to read traces, grade answers and approve changes. Decide who, and what they need to see.
7. **What should fail open, and what should fail closed?** Moderation outages, trace writes, retrieval failures, permission errors. Pick a direction for each.
8. **How will you ship changes to prompts and knowledge?** Code has CI. Prompts and knowledge need a draft, a test and a publish step too.
9. **What does your team already run well?** A team that knows Postgres and VMs should probably not start with a new database and an orchestrator.

Phase 1: Foundations (tenancy, access, credentials)

Everything else inherits these decisions. Get them right while the codebase is small.

- ☐ Model workspaces as tenants and put the tenant key on every row, ideally as part of composite foreign keys so cross-tenant references are a database error. See [Postgres schema design](#).
- ☐ Route every permission check through one `authorize(context, permission)` function with named permissions. See [the single authorization chokepoint](#).
- ☐ Separate entitlement (what a workspace has bought) from capability (what a role can do), and keep the feature catalog in code so a typo fails typecheck. See [multi-tenant RBAC](#).
- ☐ Scope access by resolving the resource's owner, never by trusting an id in the URL. See [multi-tenant RBAC](#).
- ☐ Keep a manifest of every route and its auth level, and test that the registered routes equal it. See [architecture tests](#).
- ☐ Give each credential type (session, personal access token, embed token, OAuth token) exactly one route scope, and re-check the user's live access on every request. See [headless auth](#).
- ☐ Store only hashes of sessions and tokens, show plaintext once, and let tokens expire by default. See [headless auth](#).
- ☐ Keep secrets in a vault, load them at boot, and validate config before the process accepts traffic. See [deploying the monolith](#).
- ☐ Seal third-party connector credentials in their own table, bound to the workspace and connection they belong to. See [the connector framework](#).

Phase 2: Data (schema, files, ingestion, retrieval)

Agents are only as good as the context they can reach, and only as safe as the scope filter in front of it.

- ☐ Choose a system of record for anything that needs transactions, keys or audit, and a separate store only where it earns its place (vectors, agent working state). See [Postgres schema design](#).
- ☐ Make approved artifacts immutable at the database level, with triggers if you have to. See [Postgres schema design](#).
- ☐ Track migrations by name, apply them only from the deploy, and refuse to boot on a mismatched schema. See [Postgres schema design](#).
- ☐ Upload large files straight to object storage on short-lived signed URLs, and mark a file ready only after the server verifies it. See [resumable uploads](#).
- ☐ Support resumable uploads for multi-gigabyte drawing sets and record where each file came from. See [resumable uploads](#).
- ☐ Run sync, indexing and transcription in a separate worker with leased jobs, so the API never waits on them. See [leased background jobs](#).
- ☐ Chunk deterministically so long indexing jobs can resume from a checkpoint, and hash chunks so re-ingest only embeds what changed. See [the RAG pipeline](#).
- ☐ Apply the retrieval scope filter in one place, before ranking, and test it against your vector engine's real operator support. See [the RAG pipeline](#).
- ☐ Build connectors from one registry used by both the API and the worker, with one active sync per connection. See [the connector framework](#).

Phase 3: Agents (tools, safety, delegation, approval)

This is the phase everyone wants to start with. It goes much faster once Phases 1 and 2 exist.

- ☐ Declare every agent tool in one registry, with an owning capability mapped to an RBAC feature, checked when tools are offered and again on every call. See [the agent tool manifest](#).
- ☐ Pin the short list of tools that mutate data in a test, so adding one requires review. See [the agent tool manifest](#).
- ☐ Make new agent writes stage proposals that a person accepts. See [the agent tool manifest](#).
- ☐ Order each chat turn by cost: cheap checks first, network moderation in parallel with retrieval, the model call last, with hard caps on tool turns and output. See [anatomy of an AI chat turn](#).
- ☐ Put safety behind one seam that every agent calls, run checks in cost order, and decide a failure direction for each. See [layered safety guardrails](#).
- ☐ Treat structure as the real safety: read-only tool filters, server-pinned scope and role-filtered tool lists. See [layered safety guardrails](#).
- ☐ When chat delegates to a sub-agent, let the model supply only the target and the instruction; pin workspace, user and conversation server-side. See [multi-agent delegation](#).
- ☐ Store "the next message belongs to the paused sub-agent" as a database row, and re-check permissions on every resume. See [multi-agent delegation](#).
- ☐ Split exploration from commitment: a research phase drafts with citations, a person approves through a server route, and a constrained step writes the structured record. See [two-phase agents with human approval](#).
- ☐ For generative outputs like schedules, let the model edit a declarative spec and let deterministic code expand, check and gate it. See [the AI schedule builder](#).
- ☐ If you expose tools over MCP, act as your own OAuth 2.1 authorization server, gate the advertised surface in one place, and let per-workspace overrides only restrict. See [building an enterprise MCP server](#).
- ☐ If users or clients will embed the assistant, use an isolated iframe and short-lived, scoped, revocable tokens. See [the embeddable chat widget](#).
- ☐ If the AI generates apps or code, run the output in a sandbox with no network and no credentials, reaching data only through approved queries. See [the sandboxed app builder](#).

Phase 4: Operations (tracing, evals, admin, learning)

Without this phase you cannot answer "what did the agent do, was it good, and who changed it".

- ☐ Write one trace row per agent turn into the same database as your tenant data, with tokens, cost, tools and config version. See [tracing LLM agents in Postgres](#).
- ☐ Make trace persistence best-effort, so a failing write never fails a user's turn, and redact secrets at write time. See [tracing LLM agents in Postgres](#).
- ☐ Match the grader to the output: an LLM judge for prose, deterministic checks for structured work. See [LLM evals in production](#).

- ☐ Promote golden examples from real traces and gate every config publish on a draft-versus-live comparison from the same run. See [LLM evals in production](#).
- ☐ Treat prompts, model settings, tool allow-lists and knowledge as versioned drafts that are validated at publish and stamped on every trace. See [the executive admin portal](#).
- ☐ Build an org-wide control plane with its own narrow permission, so operators can read traces, pause an agent for one customer and approve changes without database access. See [the executive admin portal](#).
- ☐ Log human corrections with a classified reason, and let only reviewer-approved, versioned, reversible promotions change future behavior. See [the AI learning loop](#).
- ☐ If you automate improvement, keep it off by default, limit what it can change, and route it through the regression gate. See [LLM evals in production](#).

Phase 5: Delivery (CI, architecture tests, deploy, rollback)

Start this in week one, not at the end. It is cheap early and expensive late.

- ☐ Wire the server, the worker and the tests through one composition root so they cannot drift apart. See [deploying the monolith](#).
- ☐ Split processes by blocking behavior: API, background worker, and anything with long-lived sessions. See [deploying the monolith](#).
- ☐ Run CI against real service containers and finish with a browser smoke test. See [deploying the monolith](#).
- ☐ Write architectural rules as tests: import layering, route manifest, catalog parity, pinned exceptions. See [architecture tests](#).
- ☐ Refuse to deploy commits that are not on the main branch, migrate between install and restart, and gate on a readiness check. See [deploying the monolith](#).
- ☐ Make rollback check that the database still matches the previous build before starting old code. See [deploying the monolith](#).
- ☐ Choose the simplest hosting that fits your scale today, and keep the process layout portable so containers later are a packaging change. See [deploying the monolith](#).

Sequencing and time to value

The phases overlap, but the order matters. This is the sequence we would recommend to a team starting today. Effort is relative, not a quote: it depends heavily on how many systems you integrate and how much of Phase 1 you already have.

Order	Work	Relative effort	What you can put in front of users at the end
1	Foundations and delivery skeleton: tenancy, authorize , route manifest, CI, deploy script	Medium	Nothing visible yet, but every later feature is cheaper and safer
2	Files, one connector and ingestion into a scoped index	Medium	A searchable project document store people use without any AI
3	A read-only chat over that index, with safety and tracing from the first turn	Small to medium	Cited answers about a project's documents, with every turn inspectable
4	One specialized agent that proposes, plus the approval route	Medium	The first workflow where AI drafts and a person commits
5	Evals, golden sets and a basic admin view	Medium	Prompt and knowledge changes you can ship with evidence
6	More agents, delegation from chat, MCP and embed surfaces	Large, incremental	The platform: many workflows, many surfaces, one set of rules
7	The learning loop and controlled automation	Medium	Estimates and answers that improve from reviewed corrections

Two observations from doing this. Steps 2 and 3 deliver value long before any agent writes anything, which buys patience for the rest. And tracing at step 3, rather than later, is the single cheapest decision on the list: every question you will ask in step 5 needs data you can only collect from the start.

Build, buy or partner

Not every box above needs custom code. Identity, object storage, vector search, content moderation and model hosting are all things to buy. What is hard to buy is the layer that encodes how your company works: your access model, your approval rules, the steps of your procedure and the knowledge your best people carry. That layer is where a platform earns its keep, and where off-the-shelf tools tend to stop.

A useful rule: buy the commodity infrastructure, build the parts that make your process yours, and partner where you need the experience to avoid the first round of mistakes. Our [build vs buy guide for construction software](#) goes through that decision in more detail, and [governing AI agents in construction](#) covers the policy questions to settle alongside it.

Why this matters if you're building something similar

- **Foundations first is faster, not slower.** Authorization, schema discipline and delivery tests make every agent you add afterwards cheaper.

- **Ship value before autonomy.** Search and cited answers earn trust that makes proposal-and-approve workflows easy to adopt.
- **Instrument from the first turn.** Traces are what evals, admin views and the learning loop are built from.
- **Keep people at the commit points.** Agents draft, propose and explain. People approve, through code the agent cannot call.

Where to go next

- [How we built Connect: architecture of an enterprise AI platform](#)
- [Multi-tenant RBAC with a single authorization chokepoint](#)
- [The agent tool manifest](#)
- [LLM evals in production](#)
- [Build vs buy for construction software](#)

Ready to turn this checklist into a plan? Start with [Plan your build](#), or [talk to us about an engagement](#).