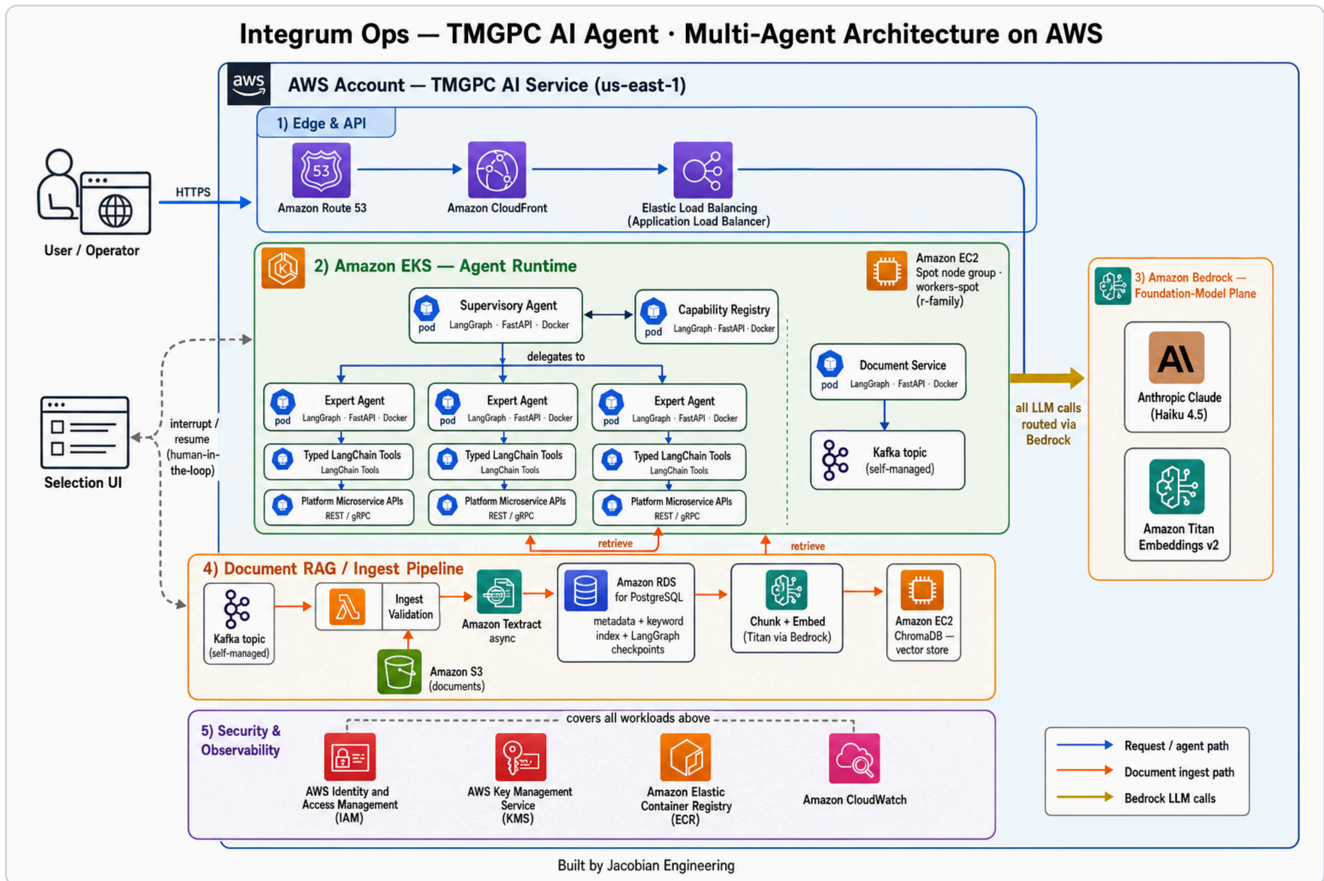
Integrum Ops: A Production Multi-Agent AI Platform for Construction Project Management



Architecture at a Glance

1. Executive Summary

TMG Project Center (TMGPC) is a new-to-the-market commercial construction project management platform. It gives general contractors and major-trade subcontractors the tools, workflow, and reporting they need to run modern commercial construction projects — submittals and RFIs, drawing and document control, schedule and field reporting — across a fabric of purpose-built microservices.

The **TMGPC AI Agent** is the intelligence layer of that platform. It delivers three capabilities to the people running construction projects: a **smart assistant** that can actually act on the platform on the operator's behalf, **RAG-based document search** grounded in the project's own drawings and documents, and **workflow automation** that drives multi-step tasks through the platform's APIs. **Jacobian Engineering designed and built the TMGPC AI Agent for Integrum Ops** — owning the architecture, the agent runtime, the document ingestion and retrieval pipeline, and the production deployment posture.

The defining engineering decision is architectural: the AI Agent is a **multi-agent system**, not a single monolithic prompt. A **supervisory agent** is the entity users talk to. Behind it sits a roster of **domain-expert agents**, each scoped to a bounded slice of the platform, each backed by a set of **typed LangChain tools** that expose real microservice API endpoints. The supervisor uses a **capability registry** to decide which expert should handle a given request and delegates accordingly. Every interaction with a foundation model — whether a Claude generation call or a Titan embedding call — is routed through **Amazon Bedrock**, which makes Bedrock the single, governed LLM plane for the entire system.

The stack is Python. **LangGraph** provides agent orchestration; **FastAPI** provides platform integration. The service is packaged as a **Docker** container and deployed onto the platform's **Amazon EKS** cluster — `tmgpc-dev-v29` — beside the rest of the microservices, targeting a **spot-instance EC2 node group** of memory-optimized r-family instances. The document RAG plane runs an ingest pipeline that consumes upload events, extracts text with **asynchronous AWS Textract**, embeds chunks through **Amazon Titan Embeddings v2 on Bedrock**, and stores vectors in **ChromaDB on EC2** alongside metadata and keyword indexes in **Amazon RDS for PostgreSQL 16.8**. A human-in-the-loop UI adapter, built on LangGraph's interrupt/resume construct with checkpoint state persisted in that same Postgres database, keeps the operator in control whenever an action is ambiguous.

> *"The TMGPC AI Agent is the intelligence layer of our platform — a smart assistant, document search you can trust, and workflow automation, all driven by a multi-agent system rather than one overgrown prompt." — Chris Worth, Co-Founder & Chief Architect, Integrum Ops This case study is the engineering story behind that sentence.*

---

2. The Customer & The Problem

Integrum Ops builds **TMG Project Center**, a commercial construction project management platform aimed squarely at the two parties who carry the operational weight of a modern build: the **general contractor** coordinating the whole job, and the **major-trade subcontractors** — mechanical, electrical, structural — who run large scopes within it. Commercial construction is a coordination problem before it is anything else. A single project generates submittals, requests for information, change orders, drawing revisions, daily field reports, inspection records, and schedule updates, and every one of those artifacts has to be findable, current, and tied to the right people and the right phase of the job.

TMGPC models that world as a platform of cooperating microservices — a document service, a drawings service, project and workflow services, and more — each owning its slice of the domain and exposing it through APIs. That is the right architecture for the platform. It is also exactly what makes a naive AI layer fail.

The temptation, when adding "AI" to a platform this broad, is to wire a single large language model to a single enormous prompt: stuff every capability description, every API, and every business rule into one context window and hope the model routes itself. That approach collapses under the breadth of construction project management. A prompt that has to simultaneously understand RFI workflow, drawing markup conventions, submittal review states, schedule logic, and document retrieval is a prompt that does none of them reliably. The model's attention is spread too thin; tool selection becomes guesswork; and there is no clean place to scope

permissions, validate inputs, or debug a wrong answer. Worse, a monolithic prompt that can *read* the platform is only half a product. The bar Integrum Ops set was an assistant that can **act** — open an RFI, locate the controlling drawing, advance a workflow — by calling the platform's own APIs the same way a human user's actions would.

There is a second dimension that compounds the first. The three capabilities Integrum Ops wanted — a smart assistant, document search, and workflow automation — are not the same *kind* of problem, and a single prompt would have to be good at all three at once. The smart assistant is an action problem: translate intent into safe, validated API calls. Document search is a retrieval-and-grounding problem: find the right passage in the project's own drawings and documents and answer from it rather than from the model's training data. Workflow automation is an orchestration problem: carry a multi-step task through several services in the right order, pausing when a human decision is required. Cramming three problem shapes into one prompt does not make the model versatile; it makes it mediocre at each, because the design has no seams along which to specialize, scope, or debug.

The "act" requirement is the sharpest constraint of all. Reading the platform is forgiving — a wrong answer to a question is a wrong answer, and the operator can see it. *Acting* on the platform is unforgiving — an action that opens the wrong RFI, advances the wrong workflow, or attaches the wrong drawing is a mutation of real project state on a real job. An AI layer that can act therefore needs an enforcement boundary between what the model *intends* and what the platform *does*: a place to validate every argument, scope every operation, and record every action. A monolithic prompt offers no such boundary. It hands the model a description of every API and trusts it to behave.

That raised the real engineering question. Not "can a model answer questions about construction," but "can we build an AI system that reasons over a wide, multi-service platform, takes bounded and safe actions through real APIs, and stays debuggable and governable as the platform grows." That is an agentic-systems problem, and it is the problem Jacobian Engineering was engaged to solve.

> *"Construction project management is a wide surface — submittals, RFIs, drawings, schedules, dozens of microservices. No single prompt is going to reason competently across all of that, and we knew it from day one." — Chris Worth, Co-Founder & Chief Architect, Integrum Ops That was the bar Jacobian Engineering was hired to clear.*

---

3. Jacobian's Approach

Integrum Ops engaged Jacobian Engineering as the **AI/ML solution partner** for the TMGPC AI Agent — owning the architecture, building the agent runtime and the document pipeline end-to-end, and shaping the production deployment posture on the platform's existing EKS footprint. The work was scoped to fit *inside* the platform Integrum Ops already runs, not beside it: the AI service is another microservice in the cluster, deployed and operated like the rest, and it consumes the platform's APIs the way any other client would.

Three engineering bets shaped the work.

1. A multi-agent supervisor over a monolith. Jacobian's foundational bet was to reject the single-prompt design entirely in favor of a **supervisory agent that delegates to domain-expert agents through a capability**

registry. Each expert owns a bounded responsibility and a bounded set of tools; the supervisor's job is routing, not domain reasoning. This is the decision that makes everything downstream tractable — scoped permissions, debuggable delegation, and the ability to add a new capability by adding an expert rather than expanding one fragile prompt. Chapter 5 covers the agent system in depth, because it is the differentiator of the whole engagement.

2. Amazon Bedrock as the single governed LLM plane. Every foundation-model call in the system — the supervisor's routing reasoning, each expert's generation, and the RAG pipeline's embeddings — is routed through **Amazon Bedrock**. The alternative, self-hosting models or scattering calls across multiple external providers, would have fragmented governance, multiplied credential surfaces, and made model-version control a per-component negotiation. Routing everything through Bedrock gives Integrum Ops one place to govern model access, one egress point to audit, one set of IAM controls, and the freedom to choose the right model for each job — Claude Haiku for cheap, high-frequency routing work, larger Claude models for harder reasoning, Titan for embeddings — without model sprawl. Bedrock is not an implementation detail here; it is the control plane for the system's AI behavior.

3. Document grounding through a dedicated RAG ingest pipeline. A construction assistant that cannot answer from the project's actual drawings and documents is a toy. Jacobian built a dedicated **ingest-and-retrieval pipeline** — event-driven from the document service, OCR'd by Textract, embedded by Titan, and split across a vector store and a relational store for hybrid retrieval — so that the document-search capability is grounded in the customer's real corpus rather than the model's training data. Construction documents are a hard retrieval target: large multi-page submittal packages, scanned field reports, and engineering drawings whose content is as much layout and table as it is prose. Grounding on that corpus required real extraction and a retrieval substrate that handles both semantic and exact-match recall, not a single off-the-shelf vector index. Chapter 6 describes that pipeline step by step.

Around those bets the engagement ran as a tight **discovery-and-build loop**. Discovery characterized the platform's microservice surface and the real action patterns operators needed — which capabilities had to be expert-scoped, which API endpoints each expert's tools would wrap, and where ambiguity in a tool result demanded a human decision rather than a model guess. The build then implemented the supervisor, the capability registry, the expert agents and their typed tools, and the ingest pipeline incrementally, deploying each into the EKS cluster as another microservice so the AI layer was exercised against the live platform from early on rather than in isolation.

---

4. Architecture Overview

The TMGPC AI Agent is built as **two cooperating planes**, both AWS-native and both anchored on Amazon Bedrock, running in **`us-east-1`** and co-located inside the platform's **Amazon EKS** cluster (`tmgpc-dev-v29`) beside the other microservices.

The agent runtime plane. This is the online, synchronous plane that users interact with. A request reaches the **supervisory agent**, which consults the **capability registry** and delegates to the appropriate **domain-expert agent**. Each expert is backed by **typed LangChain tools** that wrap the platform's microservice API endpoints, so an expert "acts" by calling the same APIs a human user's session would. Orchestration is handled by

LangGraph as a state graph; **FastAPI** exposes the service to the rest of the platform. The whole runtime is packaged as a **Docker** image and scheduled onto the EKS **spot node group** (`workers-spot`) of memory-optimized r-family instances (r6a/r6i/r7a/r7i/r8a/r8i large), with the on-demand `workers-default` group carrying baseline cluster workloads. Every model call the runtime makes — supervisor routing, expert generation — goes through **Amazon Bedrock**.

The ingest / RAG plane. This is the offline, event-driven plane that turns uploaded documents into retrievable knowledge. The document microservice emits an upload event onto a **Kafka topic** — a self-managed event backbone running inside the EKS cluster alongside the services it serves. The ingest pipeline consumes that event, validates the document for ingestion, and invokes **asynchronous AWS Textract** against the **S3 object** to extract text from construction documents and drawings. On the Textract completion notification, the pipeline indexes the document and its per-page metadata into **Amazon RDS for PostgreSQL**, then chunks the extracted text and embeds it through **Amazon Titan Embeddings v2 via Bedrock**, storing the resulting vectors in **ChromaDB running on EC2**. The result is a hybrid retrieval substrate: dense vectors in ChromaDB, metadata and keyword indexes in Postgres.

The human-in-the-loop seam. Where the two planes meet the operator, a UI adapter implements human-in-the-loop control. When an expert agent's tool call returns **multiple candidate results**, the runtime does not guess — it uses LangGraph's **interrupt/resume** construct to present a selection UI and waits for the operator's choice. The checkpoint state that makes pause-and-resume possible is persisted in the same **Amazon RDS for PostgreSQL** database that holds the RAG metadata, so an interrupted flow survives and resumes cleanly. Chapter 7 covers this seam in detail.

---

5. The Multi-Agent System

This is the chapter that distinguishes the TMGPC AI Agent from a chatbot wired to an API. The multi-agent design is the single highest-leverage piece of engineering Jacobian delivered, and it is what makes the system safe to put in front of operators running real construction projects.

The supervisory agent is the front door. Users do not talk to a dozen specialists; they talk to one supervisor. The supervisor's job is deliberately narrow: understand the request well enough to **route** it, not to answer it. It owns conversation context and delegation, and it leans on a **capability registry** — a declarative catalog of what each domain-expert agent can do — to decide who should act. When a user asks to find the controlling drawing for an RFI, the supervisor does not reason about drawings; it recognizes the request as belonging to the drawings/document expert and hands off. This separation is the core of the design: routing logic lives in one place, domain logic lives in the experts, and neither contaminates the other.

The capability registry is the routing contract. The registry is what makes the supervisor's job small and the system extensible. Rather than hard-coding a branching ladder of "if the request looks like X, call expert Y," the supervisor consults a registry that describes each expert's domain and the capabilities it exposes. Delegation becomes a lookup against that catalog: the supervisor matches the request to a registered capability and routes to its owner. The payoff is that the system grows by **registration, not surgery**. Onboarding a new domain expert — say, a scheduling expert — is a matter of building the expert and its typed tools and adding its entry to

the registry; the supervisor's routing logic does not change, and no existing expert is touched. That is the structural property that lets the AI layer keep pace with a platform that is itself still adding microservices.

Domain-expert agents own bounded slices. Each expert is responsible for one coherent area of the platform and nothing more. That bounded scope is what keeps each agent reliable: a smaller responsibility means a tighter prompt, a smaller tool set, a clearer notion of what "done" looks like, and a far simpler debugging story when something goes wrong. Adding a new platform capability to the AI layer means adding (or extending) an expert and registering it in the capability registry — not surgically editing one monolithic prompt and re-validating every behavior it ever had. The supervisor-plus-experts pattern turns "make the AI smarter" into an additive, composable operation.

Typed LangChain tools are how experts act. Each expert agent is backed by a set of **typed LangChain tools**, and each tool wraps a real **platform microservice API endpoint**. "Typed" is the load-bearing word. Every tool has a schema-validated input and output contract, which means the model cannot call a platform API with malformed or hallucinated arguments and have it silently slip through — the type boundary rejects it. Typed tools give Jacobian three properties that matter in production: **safety**, because inputs are validated at the tool boundary before any API is touched; **debuggability**, because every action an agent takes is a discrete, typed, logged tool invocation rather than free-form text the platform has to parse; and **bounded blast radius**, because an expert can only do what its registered tools allow — there is no path from the model to an arbitrary platform operation. This is the discipline that lets an AI system *act* on a production platform without becoming a liability.

Why typed tools over a single mega-prompt. A monolithic prompt that "knows" every API has no enforcement layer between the model's intent and the platform's mutation. It is a confident-text generator pointed at production. Typed tools invert that: the model proposes, the tool's schema disposes, and only well-formed, in-scope actions reach a microservice. The cost is that someone has to design the tool contracts; the payoff is an action surface that is auditable line by line and safe by construction.

Why the supervisor pattern beats one big prompt. The architectural payoff of supervisor-plus-experts is concentrated in three places. **Scoping:** each expert's permissions and tools are bounded, so the question "what can this agent do" has a small, enumerable answer instead of "whatever the one big prompt decides." **Routing:** delegation is an explicit, observable decision the supervisor makes against the registry, not an emergent property buried inside a single model's reasoning — which means a misroute is a diagnosable event, not a mystery. **Bounded responsibilities:** when an answer is wrong, the failure is localized to one expert and one tool call, so the fix is local too. A monolithic prompt has none of these seams; every change risks every behavior, and every failure implicates the whole. The multi-agent design trades a small amount of up-front structure for a system that stays comprehensible as it grows — which, for a platform still expanding its microservice surface, is the decisive property.

State flows through the graph. Because delegation is explicit, so is state. The supervisor carries conversation context; an expert receives a scoped working state for the task it was handed; results flow back up to the supervisor. Keeping that state explicit — rather than implicit in a single ever-growing prompt context — is what makes a multi-turn, multi-delegation interaction tractable to reason about and, critically, to **checkpoint** for the human-in-the-loop and resilience behaviors described later.

LangGraph orchestrates the control flow. The supervisor-and-experts topology is naturally a **graph** — nodes for the supervisor and each expert, edges for delegation and return, and state that flows along those edges. **LangGraph** models exactly this. Jacobian uses it to express the multi-agent control flow as an explicit **state**

graph rather than an implicit tangle of nested calls, which is what makes the system's behavior inspectable and its delegation deterministic enough to reason about. A state graph is the right abstraction here precisely because multi-agent control flow is not a linear pipeline: it branches (the supervisor chooses an expert), it loops (an expert may call several tools before returning), and it pauses (an ambiguous result waits on a human). Encoding those as graph nodes and edges with typed state makes each of those behaviors a first-class, testable construct rather than control-flow improvised in application code. Crucially, LangGraph's first-class support for **interruptible, resumable** execution is what the human-in-the-loop seam in Chapter 7 is built on — the state graph can pause at a node, surface a decision to the operator, and resume exactly where it left off. A bespoke orchestration layer would have had to reinvent that, and reinvent it correctly under failure.

FastAPI integrates the runtime with the platform. The agent runtime is exposed to the rest of TMGPC through **FastAPI**, the same way the platform's other Python services present themselves. That keeps the AI service a well-behaved citizen of the microservice fabric — consistent request/response semantics, consistent health and readiness surfaces, and a clean integration boundary — rather than a special-case appendage.

Packaged as Docker, scheduled on EKS spot. The runtime ships as a **Docker** container and runs on the EKS cluster's **spot node group** of memory-optimized r-family instances. Memory-optimized instances fit agent workloads well: holding graph state, conversation context, and the working set for multiple in-flight delegations is memory-bound more than CPU-bound. The cluster's node groups split the work deliberately — the on-demand `workers-default` group provides stable baseline capacity, while the `workers-spot` group of r-family instances (r6a/r6i/r7a/r7i/r8a/r8i large, scaling from zero) carries the agent containers at spot economics. Running that on **spot** capacity is an intentional cost decision, and it is only safe because the agent containers are designed to tolerate it — a subject Chapter 9 returns to, since spot interruption and the checkpoint-persistence design are two halves of the same resilience story.

The whole thing is a microservice. None of this runs in a special environment. The agent runtime is built, containerized, and deployed onto the same EKS cluster as the platform's other services, fronted by the cluster's ALB, and it integrates with the platform through the same kind of API boundary every other service uses. That is a deliberate operational choice: the AI layer inherits the platform's deployment pipeline, networking, scaling, and observability rather than standing up a parallel stack that has to be operated separately. An AI system that is a first-class member of the microservice fabric is an AI system the existing platform team can actually run.

> *"The supervisor-and-experts pattern is what made this tractable. Each expert owns a bounded slice of the platform through typed tools, and the supervisor just decides who should act — that is the difference between a demo and something we can run." — Chris Worth, Co-Founder & Chief Architect, Integrum Ops*

---

6. The Document RAG & Ingest Pipeline

The document-search capability is only as good as the pipeline that feeds it, and construction documents are an unforgiving input: large multi-page submittal packages, scanned reports, and engineering drawings dense with annotations. Jacobian built an event-driven ingest pipeline that turns those documents into a hybrid retrieval substrate, step by step.

1. The upload event. When a document lands in the platform — uploaded by a user into the document service — that service emits an **upload event onto a Kafka topic**. The Kafka topic is the document service's event backbone, **self-managed inside the EKS cluster** beside the services that use it. The ingest pipeline is a consumer of that topic, which decouples ingestion from upload: the document service's job ends when it has stored the file and published the event, and the pipeline picks up the work asynchronously. That decoupling matters operationally — a slow or temporarily unavailable ingest pipeline never blocks a user's upload, and a backlog of documents to process is just a backlog of events on the topic, drained at the pipeline's pace. Running Kafka in-cluster keeps the event backbone co-located with the producers and consumers that use it, consistent with the platform's pattern of operating its own infrastructure beside its microservices.

2. Validation. The pipeline first **validates the document for ingestion** — confirming it is a type and shape the pipeline can process — before committing any of the expensive downstream work. This gate keeps malformed or out-of-scope uploads from consuming Textract, embedding, and storage budget.

3. Asynchronous text extraction with Textract. For a validated document, the pipeline invokes the **asynchronous AWS Textract API**, passing the **S3 object** that holds the document rather than shipping bytes around. Asynchronous Textract is the correct primitive for this workload for two reasons. First, construction documents are **large** — multi-page packages and high-resolution drawings that can take meaningful time to process, well past the bounds of a synchronous call. Second, async Textract is a **durable job model**: the pipeline submits a job, Textract works it, and the pipeline is **notified on completion**, so a long extraction does not tie up a worker holding a connection open and does not fail because a request timed out. The presence of the `pc-ai-textract-temp` S3 bucket in the account corroborates this Textract-on-S3 ingest path.

4. Document and per-page metadata indexing. Once notified that extraction is complete, the pipeline **indexes the document and its per-page metadata** into **Amazon RDS for PostgreSQL 16.8**. This relational store holds the metadata and **keyword indexes** — the structured, exact-match side of retrieval. Per-page granularity matters for construction: an answer that can point to the specific page of a submittal or the specific sheet of a drawing set is far more useful than one that can only name the document.

5. Chunking and embedding through Bedrock. The final ingestion step is vector embedding. An embedding component **chunks** the extracted text into retrieval-sized units and invokes **Amazon Titan Embeddings v2 via Amazon Bedrock** to turn each chunk into a dense vector — the same Bedrock plane that serves the agent runtime, so embeddings are governed and audited through the same egress point as every other model call. The resulting vectors are stored in **ChromaDB running on EC2** (`tmgpc-dev_chromadb`), the system's vector store.

Hybrid retrieval is the point. The pipeline deliberately populates **two** stores, because document search over construction content needs both modes of recall. **ChromaDB** serves **dense semantic retrieval** — finding the chunk that is *about* what the operator asked, even when the wording differs. **Postgres** serves **keyword and metadata retrieval** — finding the exact RFI number, the named specification section, the specific drawing sheet, where semantic similarity is not the right tool and exactness is. Construction language makes this split essential. A query like "what does the spec say about concrete cover" is semantic — the relevant passage may never use the word "cover." A query like "show me drawing A-301" or "RFI 142" is exact — vector similarity would happily return near-neighbors that are precisely *not* what the operator asked for, while a keyword/metadata lookup returns the one correct record. A system that only does vector search is confidently fuzzy on identifiers; a system that only does keyword search is blind to paraphrase. Hybrid retrieval over both

stores gives the document RAG agent the recall of vector search and the precision of keyword and structured lookup, which is what a grounded, trustworthy answer in a construction context requires.

Why two purpose-fit stores instead of one. Jacobian deliberately did not force both retrieval modes into a single engine. ChromaDB is a dedicated vector store, sized and operated for embedding search on its own EC2 host; Postgres is a mature relational engine with strong keyword, metadata, and indexing capabilities — and it is already in the architecture for the LangGraph checkpoints described in Chapter 7. Letting each store do the job it is good at keeps each one simple and observable, and it means the relational store earns its keep twice: once for metadata and keyword retrieval, once for durable agent state.

The document RAG agent itself is one of the domain experts from Chapter 5: it is backed by typed tools over this retrieval substrate, the supervisor delegates document-search requests to it, and its answers are grounded in the project's own corpus rather than the model's training data.

---

7. Human-in-the-Loop

A construction assistant that *acts* on the platform has to know when not to act on its own. The most common case is ambiguity: an operator asks the agent to operate on "the structural RFI" and the tool call returns five RFIs that could match. A system that silently picks one is a system that will eventually pick wrong on a job where wrong is expensive. Jacobian built the human-in-the-loop seam precisely for this moment.

The UI adapter and the interrupt/resume construct. The user-facing interface implements a **human-in-the-loop UI adapter**. When an expert agent detects **multiple results from a tool call**, the adapter does not let the agent guess. It uses LangGraph's **interrupt / resume** construct to **pause the running graph**, surface a **selection UI** to the operator listing the candidate results, and wait. The operator makes the call; the adapter **resumes** the graph from exactly the point it paused, now carrying the operator's choice forward as if the agent had selected it all along. The interruption is not an error path bolted on afterward — it is a first-class state in the agent's control flow.

Checkpoint persistence makes pause-and-resume durable. Pausing a stateful agent graph and resuming it later — possibly after the operator steps away, possibly after the underlying container is rescheduled — requires the graph's state to live somewhere durable, not in process memory. Jacobian persists LangGraph's **checkpoint state in Amazon RDS for PostgreSQL** (the same database that holds the RAG metadata and keyword indexes). Because the checkpoint is durable, an interrupted flow is not fragile: it can wait on the operator for as long as it needs, and it can survive a container restart or a spot-instance interruption without losing the in-flight work. The human-in-the-loop seam and the spot-resilience story share this same foundation.

Why ambiguity is the right place to stop. Not every agent action needs a human in the loop — that would defeat the point of automation. The design choice Jacobian made is to interrupt specifically when a tool call returns **multiple candidate results**, because that is the precise moment where the model's confidence and the operator's knowledge diverge. The agent can reliably *find* the five RFIs that match; only the operator knows which one they meant. Drawing the human-in-the-loop boundary at multi-result ambiguity puts the human exactly where their judgment is irreplaceable and nowhere it is mere friction. The agent stays fast on the unambiguous path and yields control on the ambiguous one.

Why this builds trust in construction ops. Construction is a domain of accountability — every action ties back to a person, a phase, and a record. An AI layer earns its place there not by being autonomous but by being **controllable**. The interrupt/resume seam keeps the operator in command of every ambiguous action: the agent does the fast, broad work of finding candidates, and the human makes the consequential choice. That division is also an **auditability** win — the operator's selection is an explicit, recorded decision, not a model's silent inference — and it is exactly the property that lets an operations team trust the assistant with real work rather than relegating it to read-only Q&A. In a setting where a wrong mutation propagates into schedules, submittals, and downstream trades, "the human chose, and we recorded that they chose" is not a nicety; it is the basis on which the tool is allowed to act at all.

> *"When a tool comes back with five possible matches, the agent doesn't guess — it stops and asks the operator. In construction, the person stays in control of the ambiguous call, and that is what earns trust."*
> *— Chris Worth, Co-Founder & Chief Architect, Integrum Ops*

---

8. Complete AWS Service Inventory

The complete service inventory for the TMGPC AI Agent, grounded in the deployed dev-account resources and ordered from the generative-AI plane through the supporting compute, data, eventing, security, and observability fabric.

Category	Service	Role
AI/ML — LLM plane	Amazon Bedrock	Single governed plane for every foundation-model call — supervisor routing, expert generation, and RAG embeddings.
	Amazon Titan Embeddings v2	Chunk embedding for the document RAG ingest pipeline (`amazon.titan-embed-text-v2:0`).
	Claude on Bedrock	Generation/reasoning models available to the agents (Haiku 4.5 for cheap routing; larger Claude models for harder reasoning) — all via Bedrock.
AI/ML — document AI	AWS Textract	Asynchronous OCR/text extraction over S3 document objects; durable job model for large construction docs and drawings.
Compute	Amazon EKS	Cluster `tmgpc-dev-v29` hosts the AI service beside the platform's other microservices.
	EC2 spot node group	`workers-spot` — memory-optimized r-family instances (r6a/r6i/r7a/r7i/r8a/r8i large) run the agent containers.
	EC2 on-demand node group	`workers-default` — t3a.medium baseline capacity for the cluster.
	Amazon EC2	Standalone host `tmgpc-dev_chromadb` (t3a.medium) runs the ChromaDB vector store.
Vector store	ChromaDB on EC2	Dense semantic vectors for hybrid document retrieval.

Category	Service	Role
Relational store	Amazon RDS for PostgreSQL 16.8	Document/per-page metadata + keyword indexes **and** LangGraph checkpoint persistence (db.t4g.small).
Storage	Amazon S3	Document buckets across the ingest path — `tmgpc-dev-inbox`, `pc-ai-docs`, `tmgpc-dev-ai`, `tmgpc-dev-drawings`, `pc-lang-docs-test`, and the `pc-ai-textract-temp` working bucket.
Eventing	Self-managed Kafka on EKS	The document service's in-cluster event backbone; upload events are the ingest pipeline's trigger.
Container registry	Amazon ECR	Holds container images for the platform's services in the account.
Load balancing	Elastic Load Balancing (ALB)	`k8s-tmgpcdevalb-*` provisioned by the EKS AWS Load Balancer Controller fronts cluster services.
Identity & access	AWS IAM	Scoped, least-privilege roles (IRSA-style) granting pods access only to Bedrock and the microservice APIs they need.
Encryption	AWS KMS	Encryption keys for data at rest across the storage and database tier.
Observability	Amazon CloudWatch	Metrics, logs, and alarms for the AI service and its dependencies.

Bedrock is intentionally the **only** path to a foundation model in this inventory: there is no second model provider and no self-hosted model endpoint, which is what makes the LLM plane governable as a single surface (Chapter 9).

---

9. Security & Operational Maturity

Jacobian Engineering designed the TMGPC AI Agent to live inside a real microservice platform, which means its security and operational posture is inherited from — and additive to — the platform's existing EKS footprint. The system is in active development in the `tmgpc-dev-v29` cluster; where the dev environment differs from the production-intended posture, this chapter describes the intended target honestly rather than overstating what the dev account currently enforces.

Bedrock as the single auditable LLM egress point. The most consequential security property of the architecture is that **every** foundation-model interaction goes through **Amazon Bedrock** — supervisor routing, expert generation, and Titan embeddings alike. That single egress point is a governance, cost, and safety win at once. Governance: model access is controlled through one IAM surface, and there is exactly one place to allow, deny, or audit model usage rather than a per-component sprawl of external API credentials. Model-version control: because Bedrock brokers the models, Integrum Ops chooses and pins model versions centrally — routing cheap, high-frequency work (such as supervisor routing) to a small Claude model and reserving larger models for harder reasoning — without changing where calls go. Cost: one egress point is one place to observe and attribute spend. There is no model sprawl to chase.

EKS and IAM posture. The agent service runs as pods in the EKS cluster, and the production-intended access model is **IRSA-style scoped roles** — each workload assumes an IAM role granting **least-privilege** access to exactly the resources it needs and nothing more: Bedrock for model calls, the specific S3 buckets in the ingest path, the Postgres and ChromaDB endpoints, and the platform microservice APIs the typed tools wrap. The typed-tool design from Chapter 5 reinforces this at the application layer: even within its granted permissions, an expert agent can only invoke the bounded set of platform operations its registered tools expose.

Spot-instance resilience. Running the agent containers on the **spot** node group is a deliberate cost decision, and it is safe only because the system is built to absorb interruption. The agent containers are **stateless** with respect to durable work — the state that matters does not live in the container. When AWS reclaims a spot instance, EKS reschedules the agent pods onto healthy capacity, and any **in-flight, interrupted human-in-the-loop flow resumes from its checkpoint** because that checkpoint is persisted in **Amazon RDS for PostgreSQL**, not in the lost container's memory. Graceful handling of spot interruption and the checkpoint-persistence design are two halves of the same resilience story: durable state on Postgres is what lets ephemeral compute be cheap without being fragile.

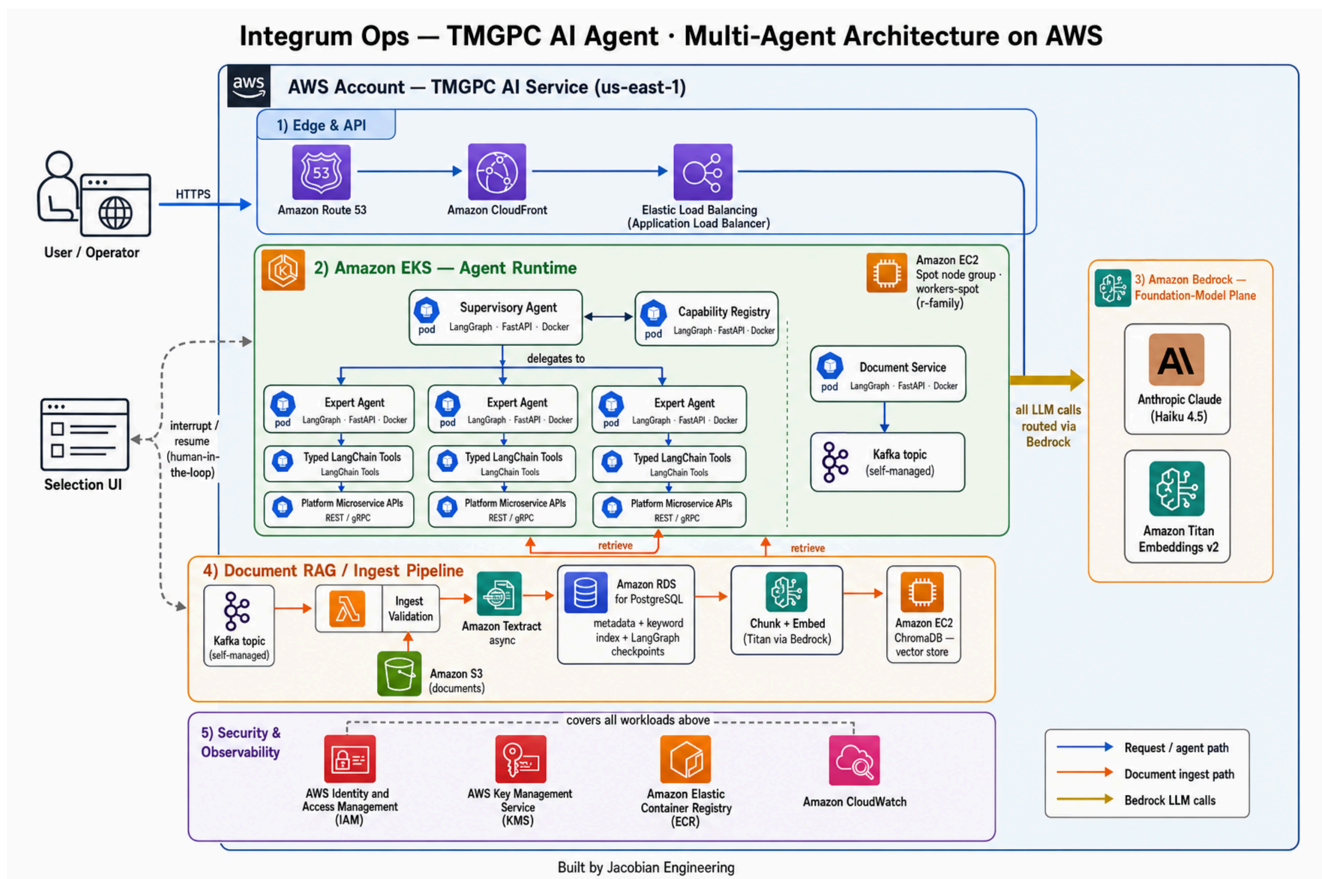
Secrets management. Database credentials, the ChromaDB endpoint, and the platform API credentials the typed tools use are managed as secrets rather than baked into images or manifests, consistent with running secret material through a managed secret surface in the cluster rather than in source. Routing all model access through Bedrock's IAM model also removes a whole class of secret — there are no long-lived third-party LLM API keys to store, rotate, or leak.

Observability. The AI service is instrumented like the rest of the platform — metrics, logs, and alarms via **Amazon CloudWatch**, alongside the cluster's own logging stack (the account runs a Loki-based log store, evidenced by the `tmgpc-dev-loki-*` buckets). Because the multi-agent design makes every action a discrete, typed tool invocation and the orchestration a explicit LangGraph state graph, the system is observable at the granularity that matters: which expert handled a request, which tool it called, and where a flow paused for a human decision.

Honest dev-stage framing. This is a system under active development, and Jacobian is explicit about that. The architecture above describes the production-intended posture; the dev cluster is where it is being proven. The value of stating it plainly is that the design decisions — Bedrock as single egress, least-privilege scoped roles, durable checkpoints under stateless spot compute — are *architectural*, in place from the start, and carry forward to production rather than being retrofitted after the fact.

---

10. Architecture at a Glance



The agent plane. Users talk to a **supervisory agent** that consults a **capability registry** and delegates to **domain-expert agents**, each backed by **typed LangChain tools** that wrap the platform's microservice API endpoints. **LangGraph** orchestrates the multi-agent control flow as an explicit state graph; **FastAPI** integrates the runtime with the rest of TMGPC; and the service runs as a **Docker** container on the EKS (`tmgpc-dev-v29`) **spot node group** of memory-optimized r-family instances, beside the platform's other microservices. Every model call — supervisor routing and expert generation — is routed through **Amazon Bedrock** in `us-east-1` .

The ingest / RAG and human-in-the-loop plane. Document uploads flow from the document service onto a **self-managed Kafka topic** in the cluster, where the ingest pipeline validates them, runs **asynchronous AWS Textract** over the **S3 object**, indexes document and per-page metadata into **Amazon RDS for PostgreSQL**, and chunks-and-embeds the text through **Amazon Titan Embeddings v2 on Bedrock** into **ChromaDB on EC2** — a hybrid vector-plus-keyword retrieval substrate. Where the agents meet operators, a **human-in-the-loop** UI adapter uses LangGraph's **interrupt/resume** construct to pause on ambiguous, multi-result tool calls and let the operator choose, with checkpoint state persisted durably in that same Postgres database so interrupted flows survive and resume.

---

11. Why Jacobian

Production multi-agent AI systems are hard in ways that do not show up in a demo. Anyone can wire a model to an API and screenshot a good answer. The difficulty lives in everything that has to be true for that system to run on a real platform, in front of real operators, day after day: orchestrating a supervisor and a roster of experts so

delegation is deterministic instead of lucky; designing **typed tools** so an agent can act on production microservices without acting *wrongly*; building a **human-in-the-loop** seam so ambiguous actions stay under human control; routing every model call through **Bedrock** so the AI plane is governable and auditable; and engineering the runtime to ride **spot** capacity by making its durable state survive interruption. The TMGPC AI Agent does all of that, and it does it as a well-behaved microservice inside Integrum Ops' existing EKS platform rather than as a bolt-on.

That is the differentiator Jacobian Engineering brings: **agentic-AI delivery** as a production engineering discipline, not a prototype-and-handoff. The multi-agent architecture, the capability registry, the typed-tool action surface, the interrupt/resume human-in-the-loop flow, and the Bedrock-governed LLM plane are not features added at the end — they are the shape of the system from the first commit, because that is what it takes to put an AI agent into the operational path of a construction platform and trust it there.

This is also the case for an **AWS AI/ML Competency** partner. The hard, valuable work in applied AI right now is exactly this kind of build: foundation models on Bedrock, wrapped in agentic orchestration, grounded in a customer's own documents through a real RAG pipeline, and deployed with the resilience and governance posture of a production system. Jacobian builds those systems and operates them where they live — inside the customer's platform, on AWS-native primitives, with the production seams in place before launch.

> *"Jacobian built this with us as a real engineering partner — multi-agent orchestration, Bedrock governance, the human-in-the-loop flow, spot-instance resilience. They treated it as a production system from the first commit, not a proof of concept." — Chris Worth, Co-Founder & Chief Architect, Integrum Ops For AWS reviewers, prospective customers, and partners considering Jacobian for their own AI/ML build: **jacobianengineering.com**.*

© 2026 Jacobian Engineering · jacobianengineering.com · Full study: trustedge.ai/resources/case-studies