Muhammad Jehanzaib

AI Case Studies — Agents, Automation & Voice

In-depth engineering case studies of products I designed and built, grouped by domain. Each one covers the problem, the requirements, the architecture, the key decisions, and measured results.

mjehanzaib99@gmail.com · linkedin.com/in/muhammad-jehanzaib-4036b5195 · github.com/mjehanzaib999 · Resume: mjehanzaib-ai-resume.up.railway.app · Portfolio: mjehanzaib-ai-portfolio.up.railway.app

LIVE deployed & clickable · DEMO video walkthrough · FLAGSHIP lead production work

Case studies

14 case studies in 11 domains. Open a domain, then pick a case study.

01 FinTech / TaxTech — 1 case study
- 1.1 Ventive — AI R&D Tax Credit Platform

02 PropTech / Real Estate — 3 case studies
- 2.1 Real-Estate CRM Intelligence Layer
- 2.2 PropertyAI CRM
- 2.3 Deal Analyzer

03 E-Commerce & Retail Intelligence — 1 case study
- 3.1 Commerce IQ

04 Logistics & Operations — 1 case study
- 4.1 MoveOps

05 HealthTech & Life Sciences — 2 case studies
- 5.1 Biologics Intelligence
- 5.2 Cure Health

06 AI Sales & Outreach — 1 case study
- 6.1 Speed-to-Lead

07 Conversational AI / Telecom — 1 case study
- 7.1 Omvi William

08 EdTech / LMS — 1 case study
- 8.1 HSC Chemistry E-Learning Platform

09 Content / SEO — 1 case study
- 9.1 AI SEO Blog Generator

10 AI Infrastructure — 1 case study
- 10.1 OptiML — LLM Compression & Fine-Tuning Framework

11 HR Operations / Agent Tooling (MCP) — 1 case study
- 11.1 AI Onboarding Ops

01 · FinTech / TaxTech

1 CASE STUDY

1.1 · Ventive — AI R&D Tax Credit Platform

FLAGSHIP DEMO FINTECH / TAXTECH

Client: a US R&D tax-credit advisory firm, delivered through Ventive · **Role:** lead AI engineer across six services, from architecture and PRDs to code, review and rollout · **Timeline:** 2026, multi-release and ongoing · **Demo:** Contract analysis walkthrough ↗

A platform that takes a tax firm's engagement from raw client documents to a signed, client-ready credit letter. AI agents read the contracts, extract the tax documents, compute the credit and fill the deliverables. Analysts approve at each gate, and the system learns from their corrections.

THE PROBLEM

- Every client contract read by hand to decide if the research was "funded".
- Payroll, ledgers and returns keyed into Excel workpapers manually.
- Credit computed, forms filled and letters written per client, hours each.

WHAT I BUILT

- A 13-node agent pipeline that decides each contract with evidence.
- Form-aware extraction for W-2, 1099, K-1, ledgers and census files.
- A rules-driven credit engine that fills workpapers, forms and the letter.

THE OUTCOME

- Documents in Drive flow to a filled workpaper with no one touching them.
- Contract verdicts agree with the firm's own answers ~86% of the time.
- Cost per analysis cut by a third; a 51-minute extraction now takes 7.

1 By the numbers

6 services behind one admin app and one shared schema	**13** nodes in the contract-analysis agent graph	**−35%** cost per analysis from prompt caching, verified like-for-like	**51 → 7.4 min** 906-employee census extraction, full coverage (was 676 rows)
~86% agreement with the firm's ground truth on a 38-contract replay	**to the cent** workpaper fills tie out to the source ledger	**26** lifecycle events on the automation bus	**9,000+** automated tests across the repos

2 The problem

An R&D tax credit study is mostly reading and arithmetic done by expensive people.

- **Contracts:** each one must be read to decide whether the customer "funded" the research, which removes it from the credit.
- **Documents:** payroll, ledgers and prior returns arrive as scans, spreadsheets and PDFs and are keyed into Excel by hand.
- **Calculation and delivery:** the credit is computed under several methods, filled into Form 6765 and state forms, and written up as a letter.

- **Consistency:** each step took hours per client and the answer depended on which analyst did it.

The firm wanted the whole chain automated, with a human approving at the points where judgement or liability sits.

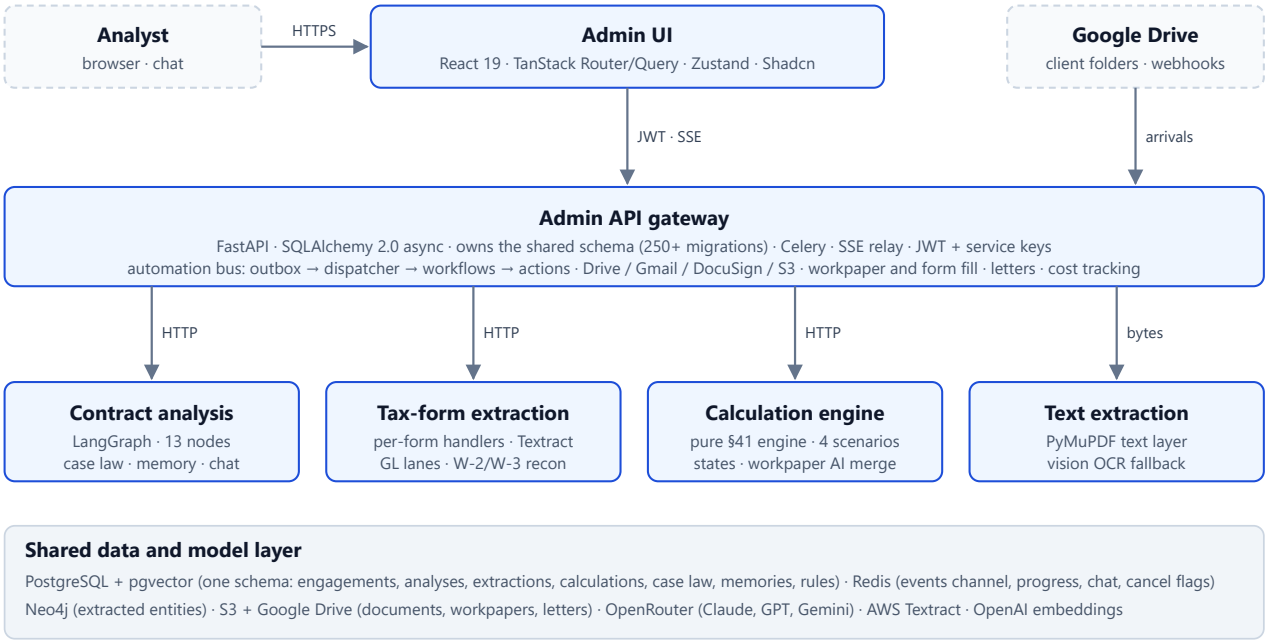
3 Requirements I derived and built against

Requirement	How the system meets it
Decide "funded research" per contract, with evidence	A 13-node LangGraph pipeline: document check, triage, web search, a rights test and a payment-risk test in parallel, adversarial self-review, an IRS counter-argument step, and a merge that derives verdict, confidence and tax position.
Never guess when the contract is silent	Triage questions pause the run; the analyst answers in chat and the run resumes from a cached first pass instead of re-paying the whole pipeline.
Ground decisions in law	A case-law and state-conformity knowledge base in Postgres with pgvector. Mandatory cases are always injected; the rest are retrieved by similarity per node.
Handle hundreds of contracts at once	Batch upload, deterministic bundling of related files (MSA + change orders, master PDF + SOW) with an LLM fallback, analyst bundle review, paced fan-out, and cancel that propagates to every worker.
Read every tax document the firm receives	A per-form handler registry (W-2/W-3, 1099, K-1, 2553, 6765, 1120/1065, census, general ledger), AWS Textract for scans, an LLM path for text, dual-model consensus, and W-2 ↔ W-3 reconciliation per EIN and year.
Read any ERP's general ledger	An LLM head-verdict detects a ledger, a layout-inference step maps columns, and seeded dialects (QuickBooks, SAP B1) parse postings with a running-balance check.
Compute the credit correctly and explainably	A pure engine: 4-scenario matrix (ASC / regular × with and without §280C), controlled-group allocation, Decimal money, full audit trail, rule-driven recommendation. Rules are versioned data per jurisdiction and year.
Fill the firm's own Excel workpapers	Header-aware sheet mapping plus an AI merge agent. Every written value must trace to the source extraction; one invented value fails the whole fill. Gaps are completed from the ledger itself.
Produce the deliverable	Government forms filled from the same data, a branded Credit Release Letter with draft/final lifecycle and Drive archival, email delivery, and DocuSign behind a kill-switch.
Run itself as documents arrive	A transactional outbox turns 26 engagement, study and interview transitions into events. Admin-authored workflows bind events to actions with dedupe, cascade guards and a run audit dashboard.
Learn from analysts	An accepted correction is distilled into a lesson; the analyst approves or edits it; later runs retrieve approved lessons per node by similarity and show which ones they used.
Show cost and progress	Token and dollar cost per node, live per-stage progress, re-run records stamped at dispatch, and a stall watchdog that fails wedged runs closed.

4 Architecture

One admin API owns the shared Postgres schema and every integration. The three AI engines are separate services that never write each other's tables. Requests go over HTTP; completions come back on one Redis

event channel, so the gateway can fan out, wait, reconcile and re-drive without any engine knowing about the others.



Blue boxes are services I built or led; dashed boxes are external. Completions flow back on one Redis channel; the gateway correlates them to workflow runs, reconciles misses on a timer, and relays them to the browser as live progress.

THE SIX SERVICES

- **Admin UI** (React 19, TypeScript): engagement timeline with live progress, batch upload and bundle review, analysis reports, an agentic chat for correction re-runs, editors for rules, prompts and memories, workflow authoring, run audit, letter and form cards.
- **Admin API gateway** (FastAPI): the only writer of the shared schema. Orchestrates batches, hosts the automation bus, fills workpapers and forms, renders letters, fronts Drive, Gmail, DocuSign and S3.
- **Contract analysis engine** (FastAPI, LangGraph, Celery): the agent graph, per-node case-law and memory retrieval, pause and resume, streaming chat with tools, correction distiller, stall watchdog, per-node cost events.
- **Tax-form extraction** (Python, Textract, LLM): classifier, per-form handlers, ledger and census lanes, reconciliation, Neo4j structuring.
- **Calculation engine** (Python, FastAPI, Celery): pure compute with a thin IO shell, readiness checks and claims so a fill runs once per input, AI merge agents with source guards.
- **Text extraction** (FastAPI): classifies each PDF as text, image or hybrid and routes to native text or vision OCR, with warnings that gate low-quality input.

5 How it works

A CONTRACT, START TO FINISH

1. **Upload and bundle.** Files are grouped into contracts by filename pattern, then by an LLM for the ambiguous ones. The analyst confirms the bundles and picks the taxpayer.
2. **Document check.** The engine extracts eight verbatim clause families (IP, payment, termination, governing law, incorporated clauses and more), folding in clauses incorporated by reference.
3. **Triage.** If the contract is silent on something that matters, the run pauses with questions. The analyst answers in chat and the run resumes.
4. **Two prongs in parallel.** A rights test (who owns the research) and a payment-risk test (who bears the cost of failure), each grounded in retrieved case law.

5. **Challenge.** An adversarial review argues the other side; an IRS counter-argument step writes what an examiner would say.
6. **Verdict.** One derived-verdict step merges the prongs into qualifies / does not qualify, a confidence score and a tax position. It is the only node allowed to write the verdict.
7. **Review and learn.** Low-confidence results wait for review. A correction re-runs only the affected nodes, and an accepted correction becomes a lesson for future runs.

A DOCUMENT, START TO FINISH

1. **Arrival.** A file landing in the client's Drive folder emits an event. Content-hash dedupe stops re-uploads from running twice.
2. **Plan.** The gateway works out what the engagement still needs (which forms, prior years, states) from the rule bundle and submits only those.
3. **Extract.** Each chunk is classified and routed to its form handler. Multi-page and multi-copy forms are merged. Ledgers go through layout inference and a balance walk.
4. **Reconcile.** W-2 totals are checked against the W-3 per EIN and year with cents-level tolerance.
5. **Fill.** When the inputs are complete, the calculation service claims the fill and the AI merge agent writes the workpaper rows, every value checked against its source.
6. **Calculate and deliver.** After workpaper approval, the engine computes the scenarios, the analyst accepts or overrides, and forms and the letter are generated and sent.

6 Agent engineering

- **Schema-constrained outputs:** every node returns strict JSON against a wire schema with a parse backstop. A malformed answer is retried once per failure kind, never silently accepted.
- **Selective re-runs:** node outputs are captured at completion. A correction marks the nodes it touches dirty and re-runs only those and their consumers, derived from what each node reads.
- **Prompt caching:** the contract is sent once as a cached block and reused by the six nodes that read it. Moving response schemas off the wire kept the cache key stable. Same contract: $3.31 → $2.16 per run.
- **Reasoning tiers:** mechanical nodes run at low effort, analytical nodes at medium. Output tokens fell 22% and wall-clock 21% with an identical verdict.
- **Safety rails:** abort probes with heartbeats between nodes, a 10-minute stall sweep that fails closed on Redis outage, publish-after-commit so no event precedes its data, and scrubbing of model text that could forge a prompt frame.
- **Correction memory:** lessons are embedded before they are saved, retrieved per node by cosine similarity (pinned lessons always), and stamped on each run so the report shows which memories were considered and applied.

7 Human in the loop

The system pauses where judgement sits, and every gate leaves an audit row:

- Triage questions before analysis, and bundle review before dispatch.
- A pending-review state for low-confidence verdicts.
- Workpaper approval before calculation; accept or override the recommended scenario with a required reason.
- A five-minute chat card to approve, edit or reject each distilled lesson.
- A reviewed-narrative gate before a letter can be finalized.

8 Key engineering decisions

Problem	Decision
Three AI services sharing one database	The gateway owns all migrations; engines declare foreign-owned tables and only write their own. Fresh-DB replay of the full chain is a release gate.
Verdicts written by four different nodes	Remove the fork so both prongs always run; one derived-verdict step is the only author of determination and confidence.
Completion events lost or duplicated	Redis pub/sub for speed plus a timer reconcile and a dispatch marker; one unified writer delivers the result whether it arrived live, on reconnect or on sweep.
A crashed re-run that looked like a 10-hour "Analyzing"	Re-run records stamped at dispatch, heartbeats per node, a stall sweep with compare-and-set, progress stored as a Redis hash so parallel branches cannot clobber each other.
LLM filling a spreadsheet invents or collapses rows	Every written value must trace to the source; recurring identical postings are kept distinct by count; any deficit is completed from the ledger deterministically.
Retries multiplying across layers	One retry ticket per failure kind threaded through every layer; worst case went from 18 billable generations to 4.
Tax rules changing every year	Rules, prompts and templates are versioned data in a knowledge base with an editor and a sign-off gate, not code.
Learning without retraining	Distill corrections into short lessons, gate them on human approval, retrieve by similarity per node, stamp each run with the lessons it used.

9 Results

Measure	Result
Sandbox replay of a real client year (38 contracts)	**31 completed, 7 for review, 0 failed**; 38 of 38 workpaper rows bound; ~86% agreement with the firm's ground truth
Cost per contract analysis (same contract, same model)	**$3.31 → $2.16** with prompt caching (−35%); output tokens −22% with reasoning tiers
906-employee census extraction	**51 min → 7.4 min**, 906 of 906 rows (was 676)
Expense workpaper from a general ledger	**105 rows**, contract-research total tied to the cent
Arrival-driven pipeline, end to end	file arrival → extraction → readiness → calculation → workpaper in Drive, **unattended**

- The extraction-to-workpaper chain ran unattended on a second client's 24 documents, including 9 stranded runs auto-redriven by the sweep.
- The correction-memory loop was closed live: an approved lesson was applied by three nodes on the next similar contract and flipped the verdict correctly.
- Delivered as stacked, reviewed pull requests across six repositories, each with CI: lint, type-check, migration replay and full test suites.

STACK Python · FastAPI · LangGraph · Celery · Redis · PostgreSQL + pgvector · SQLAlchemy 2.0 async · Alembic · Neo4j · AWS Textract · PyMuPDF · pypdf · openpyxl · OpenRouter (Claude Opus / Sonnet, GPT, Gemini) · OpenAI embeddings · React 19 · TypeScript · Vite · TanStack Router/Query · Zustand · Shadcn UI · Google Drive & Gmail APIs · DocuSign · S3 · Docker · GitHub Actions

LINKS Watch demos: Contract analysis · Tax-form extraction · 1 | 2 | 3 | 4

02 · PropTech / Real Estate

3 CASE STUDIES

2.1 · Real-Estate CRM Intelligence Layer LIVE

PROPTECH / REAL ESTATE

Client: a high-volume residential real-estate agent and their four-person team · **Role:** sole engineer, design through production operation · **Timeline:** Jul 2026 – present, live since Oct 2026 · **Code:** private repo, 246 commits

An AI layer between four systems that do not talk to each other: Gmail, Follow Up Boss (people), Monday.com (deals) and Slack. It reads every email as it arrives, works out who each person is and what role they play, extracts the terms of attached contracts, and keeps tags, deal stages and notes in sync. It owns no data of its own and never writes without a confidence floor or a human rule behind it.

THE PROBLEM

- 300–400 emails a day and 40–50 closings a year, triaged by hand.
- The old CRM had no API, so nothing could be automated.
- Contacts, deals and conversations lived in four tools with no link between them.

WHAT I BUILT

- A Gmail push pipeline with Claude classification, identity and contract extraction.
- A memory of every person's role per property, ranked by evidence source.
- An SLA state machine that escalates missed VIP messages to Slack.

THE OUTCOME

- Validated by month-by-month dry-run replays of 500–800 real emails before any live write.
- In production: 1,039 emails in the first three days with zero errors.
- Next agent is a config change, not a fork.

1 By the numbers

1,168
tests across 45 files; the suite is larger than the app

7
versioned prompts; classification is on v32

0.80 / 0.85
confidence floors for property and identity matches

20 + 9
Celery tasks and scheduled Beat jobs

3
Claude roles (fast / contracts / smart) with OpenAI failover

554
lines of policy YAML, the only client-specific layer

~84%
of input tokens served from prompt cache

150
messages per catch-up chunk so a backlog drains safely

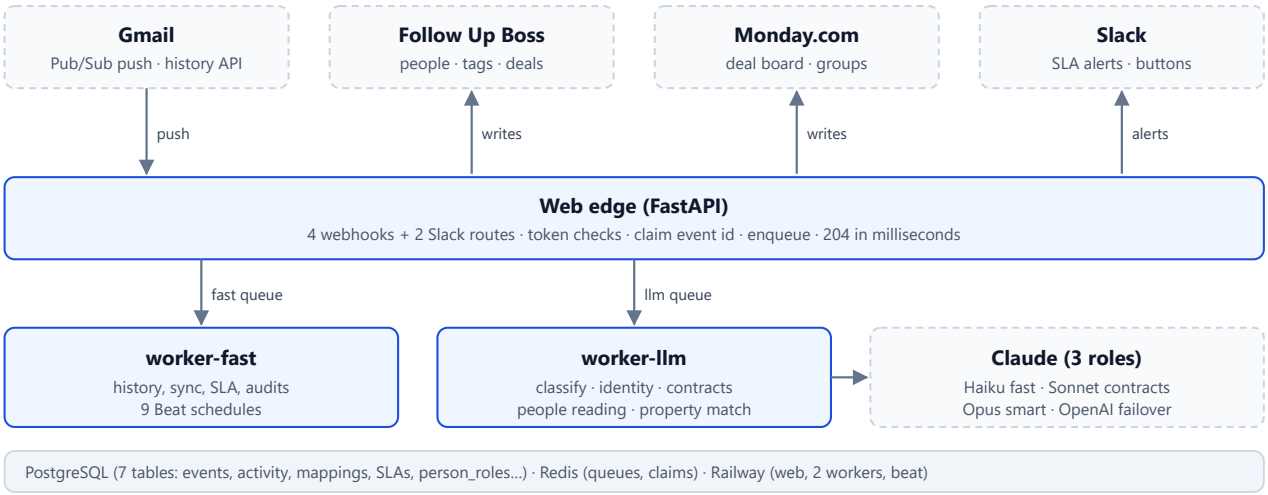
2 The problem

- **Volume:** hundreds of emails a day, many with contracts, disclosures and closing statements attached.
- **Identity:** the same person appears as buyer on one property and agent on another; names are misspelt; trusts and LLCs hide the people behind them.
- **Sync:** tags in Follow Up Boss and stages on the Monday deal board drifted apart, and nobody knew which was right.
- **Response time:** VIP and in-escrow messages were missed because nothing tracked them.

3 Requirements I derived and built against

Requirement	How the system meets it
Read mail the moment it arrives	Gmail Pub/Sub push to a thin webhook that checks a token, claims the message id, enqueues a Celery task and answers in milliseconds. Workers read history from a stored checkpoint in bounded chunks.
Classify every email reliably	Claude with a forced tool call and Pydantic validation; categories vip, active_deal, vendor, fyi, spam. The address is rebuilt from structured parts, never from free text.
Know who each person is	Exact email match first; otherwise up to 8 candidates go to an identity prompt that answers same_as, new_person, not_a_client or unsure, with invented ids rejected.
Understand the contract, not the forwarder	PDFs go to Claude as native document blocks (25 MB cap). Parties come from the document; non-transaction documents are refused.
Link to the right property	An LLM match against live board items with a 0.80 floor. A clean no-match may create a listing; an ambiguous match is logged and refused.
Escalate what is missed	SLA tiers: active escrow 2 h, VIP 4 h, sphere 24 h, paused outside business hours. States pending → escalated → covered / dismissed / resolved with Slack claim, snooze and override buttons.
Never write twice, never write blind	Unique (provider, event id) dedupe, claim rows per step, dead-letter rows after retries, a write log per CRM write with its cause, and dry-run-first scripts that need both --apply and --approved-by.
Redeploy for the next agent	Every client-specific value, from board ids to SLA hours, lives in one policy file. The README's setup is "clone and reconfigure".

4 Architecture



Thin webhook edges answer inside each provider's acknowledgement window (Slack 3 s, Follow Up Boss 10 s). Every model call and every platform write runs as a task on one of two queues.

5 How it works

ONE EMAIL, START TO FINISH

1. **Arrive.** Gmail pushes a history id. The edge claims it and enqueues; the worker reads new messages under a single-flight lock.

2. **Classify.** Claude returns the category, the property address in structured parts, and whether the mail moves a deal. Outbound mail is only classified if it carries a contract.
3. **Resolve people.** Exact email match, else the identity prompt over candidates. New people are created only with an email address, per the client's rule.
4. **Read documents.** Contracts, addenda and closing statements are extracted (prices, dates, parties, counters, with the latest counter winning). Closing statements can be pulled from Drive.
5. **Match the property.** The LLM matches against live deals; a confident match links, an ambiguous one is refused.
6. **Update memory.** Each person gets a live role row per property (seller, buyer, counterparty, professional…) with a stage and evidence quotes. Sources are ranked human > contract > board > thread > classifier > tags, and a human row is never overridden.
7. **Sync.** Tags, deal stage, a dated note line and contact price are written to Follow Up Boss, each write logged with its cause. Deals move forward only.
8. **Watch the clock.** A pending SLA opens for VIP and in-escrow senders and escalates to Slack if unanswered.

6 Agent engineering

- **Models by job:** Haiku for property matching and closing-statement picks, Sonnet 5.5 with medium adaptive thinking for classification, contracts, roles and identity, Opus for client synthesis. Chosen on a 40-email comparison.
- **Prompt caching:** the system prompt and policy context are cached; about 11.2k of 13.4k input tokens per call are cacheable.
- **Replay before write:** each month's real emails were replayed in dry-run against a scratch database, every verdict reviewed, and defects fixed as versioned prompt changes before the next month went live. Prompts advanced from v24 to v32 this way.
- **Facts beat tags:** a contract or a human row always outranks what a classifier or an old tag says.
- **Deterministic edges:** the LLM judges; code builds the canonical address string and enforces stage order, thresholds and the "no contact without email" rule.

7 Key engineering decisions

Problem	Decision
Providers time out on slow webhooks	Thin fast-ack edges, fat Celery workers; every LLM call and platform write is a task with retries, claims and dead letters.
The model invented contact ids and addresses	Invented-id guards on every pick; addresses rebuilt from structured parts (the model had produced 437 strings for 229 properties).
Wrong buyers and sellers from forwarded mail	Attribution from the document and the thread, never the sender; parties behind trusts resolved by surname.
A person is a client on one deal, an agent on another	Per-property role rows with source ranking and forward-only stages, instead of one global tag.
One bad write costs trust	Confidence floors, refuse-on-ambiguity, a write log with causes, Monday writes off by default, and apply scripts that need a named approver.
Client-specific rules everywhere	One policy file for ids, mappings, SLA tiers, team names and thresholds; a sandbox variant for testing.

8 Results

Measure	Result
Dry-run replays before go-live	**Aug, Sep and Oct 2026**, 500–800 real emails each, every verdict reviewed; fixes shipped as prompt versions
First three days in production	**1,039 emails, 0 errors**; 3 contacts created, 6 flagged for review
Steady state	**p50 2.5 s** per inbound email through classification
Outage recovery	1,050 emails dead-lettered during an API-key outage were replayed in dry run for about $18, then 20 approved writes applied
Replay fidelity	**97.4% agreement** between the live API and an offline replay of 537 emails

STACK Python 3.12 · FastAPI · Celery + Beat · Redis · PostgreSQL · SQLAlchemy 2.0 · Alembic (11 migrations) · Anthropic SDK (Claude Haiku 4.5, Sonnet 5.5, Opus) · Gmail API + Pub/Sub · Follow Up Boss REST · Monday.com GraphQL · Slack Bolt · Google Drive · Railway · GitHub Actions

2.2 · PropertyAI CRM — AI Property Management on Buildium

LIVE DEMO

PROPTECH / REAL ESTATE

Client: Kognio AI product, scoped against a 75+ unit Buildium property manager · **Role:** architect and sole engineer · **Timeline:** Apr – Aug 2026, 125 commits · **Live:** propertyai.kognio.ai ↗ · **Demo:** Watch ↗

A tenant-communication and operations layer on top of Buildium. Inbound tenant messages on four channels go through one AI pipeline that answers, detects intent and opens the right record: a maintenance request and vendor dispatch, an escalation, a rent-hardship task or a renewal. Fifteen scheduled jobs run the rest of the lifecycle, and a Portfolio Copilot answers questions over a live snapshot of the portfolio.

THE PROBLEM

- Managers of 50–100+ units spend their day on rent reminders, maintenance intake and renewals.
- Buildium holds the data but does none of the communication.
- Every vendor (SMS, e-sign, screening, payments) is a separate contract.

WHAT I BUILT

- A single-service FastAPI + React app with 248 endpoints and 56 tables.
- One AI pipeline for tenant messaging with per-tenant memory and intent actions.
- Eleven provider seams that default to mock, so every flow runs with no vendor accounts.

THE OUTCOME

- Rent follow-ups, renewals, escalation and dispatch run unattended on a schedule.
- Live two-way Buildium sync for tenants, leases, balances and work orders.
- Deployed and demo-ready on Railway with seeded data for 27 modules.

1 By the numbers

15
scheduled jobs across the tenant lifecycle

248
API endpoints across 55 routers

56 / 40
tables and Alembic migrations

58
React pages, 10 of them public magic-link flows

4
inbound channels (SMS, WhatsApp, email, voice) into one pipeline

11
pluggable provider seams, mock by default

10
turns of memory per tenant conversation

5
AI surfaces: messaging, rent follow-up, Copilot, Retention Radar, KB Q&A

2 The problem

- **Repetitive communication:** the same reminders, intake questions and renewal offers, written by hand, many times a day.
- **Slow intake:** a maintenance text sits until someone reads it, then gets re-typed into Buildium.
- **No memory:** each tenant conversation starts from zero.
- **Vendor sprawl:** SMS, e-signature, screening, payments and valuation each need a provider before anything can be demonstrated.

3 Requirements I derived and built against

Requirement	How the system meets it
Answer tenants on any channel	Inbound webhooks for SMS, WhatsApp, email and voice feed one pipeline; the reply goes back on the same channel through a messaging seam.
Act on what the tenant means	The model returns an intent in a metadata block. Maintenance creates a request, a task and a vendor dispatch; escalation creates an urgent task; rent hardship and lease questions open their own tasks.
Remember the conversation	One active conversation per tenant with a JSON transcript; the last 10 turns and the latest intent are replayed into every prompt, with tenant facts and up to 3 knowledge articles.
Run the lifecycle without a person	APScheduler jobs: rent follow-up daily, renewals weekly, escalation hourly, Buildium sync every 6 h, dispatch every 2 h, late fees, payment plans, insurance offers, turnover, compliance, credit reporting, drips, showing feedback.
Renew leases end to end	Offers built from policy (+3%, 12 months, 60-day window) with a ReportLab PDF and a public token page; status offered → viewed → accepted → signed → completed, each step audited.
Mirror Buildium	REST sync pulls tenants, leases and balances every 6 hours; form-created maintenance requests push work orders back.
Answer portfolio questions	Portfolio Copilot builds a live snapshot (occupancy, top delinquencies, leases ending in 60 days, urgent maintenance, turnovers) and streams the answer over SSE.
Demonstrate without vendor accounts	Provider seams for SMS, email, WhatsApp, voice, payments, screening, e-sign, abstraction, listing content, AVM and MLS, each defaulting to a logged mock.

4 Architecture

- **One process:** FastAPI serves the API, runs the scheduler in its startup hook, and serves the built React app from a catch-all route. Two-stage Docker build; Alembic upgrade on boot; same origin so no CORS.
- **Data:** PostgreSQL via async SQLAlchemy, 56 tables in one models file, 40 migrations with a create-all fallback.
- **Integrations:** Buildium REST over httpx is the one real external system; OpenRouter for every model call through a single shared client.
- **Client:** React 18 + TypeScript + Vite, 49 private routes and 10 public token routes (tenant portal, pay, sign, renew, apply, move, owner).

5 How it works

A TENANT TEXT MESSAGE

1. **Receive.** The channel webhook finds the tenant by phone and loads their active conversation.
2. **Compose.** The prompt carries the tenant's facts (unit, rent, balance, lease end), the last 10 turns and matching knowledge articles.
3. **Reply and classify.** Claude answers in plain language and appends a metadata block with the intent and urgency; regex parses it, and a canned apology covers an empty answer.
4. **Act.** The intent handler creates the maintenance request, task, dispatch, escalation or renewal task, and the transcript is saved.
5. **Follow through.** The hourly escalation job promotes stale high-priority tasks; open emergency maintenance older than four hours raises an urgent task.

6 Agent engineering

- **Grounded snapshots, not tool calls:** the Copilot receives the portfolio as data and is told never to invent tenants; the client holds the last 12 turns.
- **Retention Radar:** up to 60 tenants' signals in one JSON call; scores are clamped, bands re-derived, and a transparent heuristic fills gaps so the page always renders and reports its source.
- **Knowledge base Q&A:** keyword-overlap retrieval and an answer-only-from-articles prompt; deflected questions are tagged and article use counts rise.
- **Humans keep the pen:** screening runs in assist mode and lease abstraction is confirmed by a person before terms are written.
- **Deterministic where it should be:** listing copy, maintenance triage and lease abstraction are templates and rules with a documented swap to a model provider.

7 Key engineering decisions

Problem	Decision
Demo needs every vendor live	Mock-first provider seams with a config switch per provider; flows stay repeatable.
Scheduled jobs re-sending messages	Idempotency checks per day, per active renewal and per month inside every job.
Streaming closes the request session	SSE generators open their own database session.
Many model clients drifting	One shared OpenRouter client with a comment forbidding a second.

Fair-housing risk in pricing	Renewal pricing takes no protected-attribute inputs.

8 Results

- Rent follow-ups, renewals, escalation, dispatch and Buildium sync run on schedule with no operator.
- A tenant maintenance text becomes a request, a task and a vendor dispatch in one turn; a form-created request becomes a Buildium work order.
- Deployed to Railway with a demo guide, dark mode, 27 seeded modules and a 21-connector integrations catalogue.

STACK React 18 · TypeScript · Vite · FastAPI · PostgreSQL · SQLAlchemy (async) · Alembic · APScheduler · OpenRouter (Claude Sonnet 4.5) · Buildium REST · ReportLab · JWT · Docker · Railway

LINKS Open the live app ↗ · Watch demo ↗ · Case study on GitHub ↗

2.3 · Deal Analyzer — Investment Underwriting LIVE PROPTECH / REAL ESTATE

Client: Kognio AI product for wholesalers and investors · **Role:** sole engineer · **Timeline:** Aug 2026, built in 3 days · **Live:** analyzer.kognio.ai ↗

Enter a property and its comparable sales; get the after-repair value, five exit strategies, a 0–100 score, a stress test and an AI underwriting read. Every number comes from a deterministic engine. The model only interprets what the engine computed.

THE PROBLEM

- Underwriting a deal means the same spreadsheet math, repeated per property.
- Investors want a read on strategy fit and an offer plan, not only numbers.
- An AI that invents a cap rate is worse than none.

WHAT I BUILT

- Pure-function engines for comps, ARV, five strategies, scoring and sensitivity.
- An Underwriting Agent that streams fit scores and rationale over SSE.
- A Negotiation Strategist that returns an offer ladder and scripts.

THE OUTCOME

- Numbers appear instantly; AI commentary streams after.
- Works with no model key at all: deterministic fallbacks everywhere.
- Kanban pipeline, dashboard, compare and CSV import on top.

1 By the numbers

5
exit strategies (wholesale, flip, buy-and-hold, subject-to, seller finance)

25
scenarios in the sensitivity grid plus break-even ARV per strategy

6
weighted scoring criteria → 0–100 score and A–F grade

13 × 3
rehab line items × severity bands

21
API endpoints across 11 routers

0
financial figures written by the model, by design

2 How it works

1. **Comps and ARV.** Each comp starts at 100 and loses capped points for size, beds and baths, distance, condition and age; ARV is the similarity-weighted price per square foot times the subject's size.
2. **Strategies.** Wholesale (70% rule less repairs and fee), flip (6 months holding at 1%/month, 8% selling costs), buy-and-hold (rent, taxes, insurance, vacancy and maintenance → NOI, cap rate, cash-on-cash), subject-to when mortgage data exists, and seller finance at 85% of ARV.
3. **Score and stress.** Six weighted criteria give the score and grade; a 5×5 grid moves ARV −10% to +10% and repairs +0% to +50%, and a binary search finds each strategy's break-even ARV.
4. **Underwrite.** The agent streams deterministic strategy cards first, then AI fit scores and risk flags for buy-and-hold, BRRRR, flip and wholesale, then a streamed rationale, over one SSE connection.
5. **Negotiate.** The strategist returns opening, target and walk-away prices, a concession ladder, talking points and two message scripts, with the recommended strategy's offer price as the ceiling.

3 Key engineering decisions

Problem	Decision
Trust in the numbers	Thin routes, pure service functions; the engine runs first and the model receives its output as JSON with a "never invent numbers" rule.
Waiting on the model	Deterministic cards are sent before any model token, so the page is useful immediately.
No API key in some environments	Heuristic judgement and a fallback plan replace the model, so the product still works.
Two services, one origin	A runtime Next.js route handler proxies to the private FastAPI backend and passes streams through unbuffered.
Repeatable demos	Mock comps are seeded from a hash of the address, and an idempotent seed creates 10 deals and 30 comps on an empty database.

STACK Next.js 14 (App Router) · React · TypeScript · shadcn/ui · FastAPI · SQLAlchemy (async) · OpenRouter (Claude Sonnet 4.5) · SSE · Docker · Railway

LINKS Open the live app ↗ · Case study on GitHub ↗

03 · E-Commerce & Retail Intelligence

1 CASE STUDY

3.1 · Commerce IQ — Commerce Intelligence for Amazon-Led Brands LIVE DEMO

E-COMMERCE & RETAIL

Client: Kognio AI product for founders of Amazon-led D2C brands · **Role:** sole engineer · **Timeline:** Apr – Aug 2026 · **Live:** commerce.kognio.ai ↗ · **Demo:** Watch ↗

A decision-support tool that tells a founder what changed yesterday, what to worry about and what to do. Commerce data lands in a canonical warehouse, six deterministic rules score it every morning, and a tool-calling chat agent, a root-cause analyst and a Telegram digest explain the results without ever inventing a figure.

THE PROBLEM

- Sales, ads, inventory, Buy Box and returns live in separate reports.
- Founders find out about a stock-out or a conversion drop days late.
- Generic AI summaries make up numbers.

WHAT I BUILT

- A 15-table warehouse fed by a pluggable ingestion seam.
- Six scoring rules, a lead-time-aware reorder engine and a true-profit calculator.
- A chat agent with 10 typed, account-scoped tools, plus streaming analysis.

THE OUTCOME

- A nightly pipeline: ingest 02:00, score 06:00, digest 07:00 UTC.
- Alerts, in-app notifications, email by severity and a Telegram digest.
- One engine feeds the alert, the page and the AI planner, so they never disagree.

1 By the numbers

6
deterministic alert rules with severity thresholds

10
typed agent tools, account id injected server-side

44
API endpoints across 14 routers

15
warehouse and app tables

3
nightly stages: ingest → score → digest

5
selectable models behind one OpenRouter seam

45 × 7 × 25
days × marketplaces × SKUs of seeded data with planted anomalies

~17k
lines of code across the stack

2 Requirements I derived and built against

Requirement	How the system meets it
Catch what changed yesterday	Rules on daily facts: revenue ≤ −15% vs 7-day average, conversion ≤ −20% day over day, ACOS ≥ +25%, Buy Box share drop ≥ 15 points from a ≥ 85% baseline, return rate ≥ 2× baseline, and reorder-now from the reorder engine. Each has a high or critical band.

Know when to reorder	Velocity over 7 days, days of cover, reorder point = lead time + safety stock, quantity = (lead + safety + cycle) × velocity − stock, with per-ASIN overrides and a supplier.
Show true profit	(Gross − refunds) − Amazon fee % − other fees − ad spend − units × COGS, with COGS uploaded by CSV and SKUs missing COGS counted.
Ask questions in plain English	A tool loop (max 6 iterations) over summary, marketplaces, products, advertising, alerts, inventory, change, reorder, profit and returns tools; streamed over SSE; history stored.
Explain a bad day	Root-Cause Analyst gathers evidence server-side, streams status and evidence chips, then a narrative, then a strict-JSON list of ranked causes; the narrative stands if the JSON fails.
Plan the next order	Reorder Planner wraps the engine output (15 items) in one JSON call; broken JSON is salvaged per ASIN and every missing field falls back to the engine value.
Reach the founder where they are	In-app notifications for the tenant, email above each user's severity threshold, and a daily Telegram digest from a fixed template.
Swap the data source later	An ingestion Protocol with four fetch methods; mock source implemented, Amazon SP-API documented as a stub, delete-then-insert per window so re-runs are safe.

3 Architecture

- **Single same-origin service:** a two-stage Dockerfile builds the Vite app; FastAPI serves it and the API, and runs APScheduler in its lifespan hook.
- **Warehouse:** fact tables for account, marketplace, product, advertising and returns by day, plus alerts, notifications, preferences, chat sessions and messages, product settings, COGS and fee settings.
- **Models:** every call through OpenRouter's OpenAI-compatible client; the chat UI offers Sonnet, Opus, Haiku, GPT-5 and Gemini 2.5 Pro.
- **Frontend:** React 18 + TypeScript + Vite + TanStack Query; 12 app pages and 7 settings sub-pages.

4 Key engineering decisions

Problem	Decision
Models inventing figures	Rules produce the facts; prompts say "never invent figures"; the AI only explains and ranks.
Three screens showing different reorder advice	One reorder engine shared by the alert rule, the inventory page and the AI planner.
Truncated JSON on large plans	Cap to 15 items, raise the token budget, salvage complete objects, fall back per field, never raise.
Re-running ingestion	Delete the date window and re-insert, then rebuild the daily rollup.
Notifications breaking the scoring job	Notification and AI steps fail soft so an alert is always written.

STACK React 18 · TypeScript · Vite · Tailwind · TanStack Query · FastAPI · SQLAlchemy 2.0 (async) · PostgreSQL 16 · APScheduler · OpenRouter (multi-model) · python-telegram-bot · Docker · Railway

LINKS Open the live app ↗ · Watch demo ↗ · Case study on GitHub ↗

04 · Logistics & Operations

1 CASE STUDY

4.1 · MoveOps — AI Operations Platform for Movers

LOGISTICS & OPERATIONS

LIVE

Client: Kognio AI product for removals companies · **Role:** sole engineer · **Timeline:** 2026 · **Live:** moveops.kognio.ai ↗

One hub for a moving company's operations: leads from marketplaces, website and email land in a pipeline, a rules engine automates the follow-ups, an AI estimator turns a description of a home into inventory, volume, crew and price, and field staff close the job to an invoice on their phone.

THE PROBLEM

- Leads arrive from Oneflare, Find a Mover, the website and email and are handled by hand.
- Slow replies lose bookings; nothing tracks the pipeline.
- Zapier-style automation adds per-run cost and lock-in.

WHAT I BUILT

- A lead → job → invoice pipeline with 52 endpoints and 16 tables.
- An automation engine with data-defined rules, timed sweeps and a run audit.
- Four AI features, each with a deterministic fallback.

THE OUTCOME

- New leads get an automatic reply, assignment and reminder with no one watching.
- The estimator streams its reasoning, then a structured estimate.
- The whole demo works with or without a model key.

1 By the numbers

52
API endpoints

8 / 4 / 5
built-in automations, action types, condition operators

3
messaging channels (SMS, email, WhatsApp) behind one adapter Protocol

4
AI features, each with a non-LLM fallback

25
leads scored per batched model call

16
tables; 21 pages

2 How it works

A RULE, END TO END

1. **Event.** A domain action fires a trigger: lead created, stage changed, job booked, job completed, invoice generated, or a timed sweep (uncontacted for 30 minutes, job in 3 days, job tomorrow).
2. **Match.** The engine loads the tenant's enabled rules for that trigger and builds a context of lead, job and extras.
3. **Evaluate.** One condition per rule, a dotted path and an operator (eq, ne, in, not_in, contains); an empty condition always passes.

4. **Act.** Send a templated message on a channel, create a notification with interpolated fields, update the lead stage, or round-robin assign.
5. **Record.** Every rule writes a run row as success, skipped or failed with a detail string; exceptions are caught and logged.

AN AI ESTIMATE

1. The prompt sets a surveyor persona with per-item volumes, crew and truck bands and hourly pricing.
2. The model streams prose reasoning; the server withholds the last few characters until it sees the JSON marker, so a half-sent marker never reaches the client.
3. After the marker, the JSON is coerced: missing totals are summed from items, cubic feet converted to m^3.
4. No key, an error or bad JSON falls back to a heuristic estimate from bedroom count with a price band.

3 Key engineering decisions

Problem	Decision
Automation vendors charge per run	Rules are JSON data evaluated by plain Python inside the hub; no external workflow tool.
Demo without SMS or email accounts	Each channel adapter logs to the console and returns a fallback id when unconfigured; every send still writes a Message row.
Preview and invoice disagreeing	One pricing engine (quarter-hour rounding, minimum hours, capped discount and deposit, tax after discount) serves both the field preview and the saved invoice.
Chatbot making up prices	A 4-iteration tool loop whose only tools are FAQs, the same rule-based quote service and create-lead with consent; FAQ keyword matching when the model is off.
Email leads in free text	Regex first, then a temperature-0 JSON call; model values win where present.

STACK React · TypeScript · Vite · Tailwind · FastAPI · SQLAlchemy (async) · PostgreSQL · APScheduler · OpenRouter (Claude Sonnet 4.5) · aiosmtplib · Twilio · WhatsApp Graph API · Docker · Railway

LINKS Open the live app ↗ · Case study on GitHub ↗

05 · HealthTech & Life Sciences

2 CASE STUDIES

5.1 · Biologics Intelligence — Citation-Verified Research Assistant

FLAGSHIP LIVE DEMO

LIFE SCIENCES / COMPETITIVE INTELLIGENCE

Client: a consultancy in the spinal bone-graft biologics market · **Role:** sole engineer from first commit to production handover · **Timeline:** Aug – Oct 2026, 517 commits · **Demo:** Watch the walkthrough ↗

A research assistant that answers questions like "which of these 49 products are cleared as 510(k), PMA or 361 HCT/P, and since when?" from a library it keeps current itself. Every sentence is tied to a passage the reviewer can open, every citation is checked before the reader sees it, and when the corpus does not hold the answer it says so.

THE PROBLEM

- Analysts read FDA records, trial registries, journals and SEC filings by hand.
- Days per question, and no way to show where each fact came from.
- Product names collide with common words and company aliases.

WHAT I BUILT

- An ingestion pipeline over 8 public sources plus uploads, with entity resolution.
- Three-lane retrieval fused by rank over a filtered DiskANN index.
- A plan → search → judge → write → check workflow with server-verified quotes.

THE OUTCOME

- A hard question went from 6 minutes and $4 to 2 minutes and $1.
- Every regulatory number on the acceptance set verified correct against the FDA record.
- Handed over with admin console, runbook, guides and a cost sheet.

1 By the numbers

33,800
documents · 668,000 passages · 711 tracked products

6 min $4 → 2 min $1
a hard question, before and after retrieval and judge work

$1.36 → $0.07
judge cost per broad question

99.9%
agreement between the filtered index and the exact scan

230
questions answered in production in 60 days, median $0.12

~$70
to embed and OCR the first 34,000-document library

~3,000
API tests · 773 web tests · 47 migrations

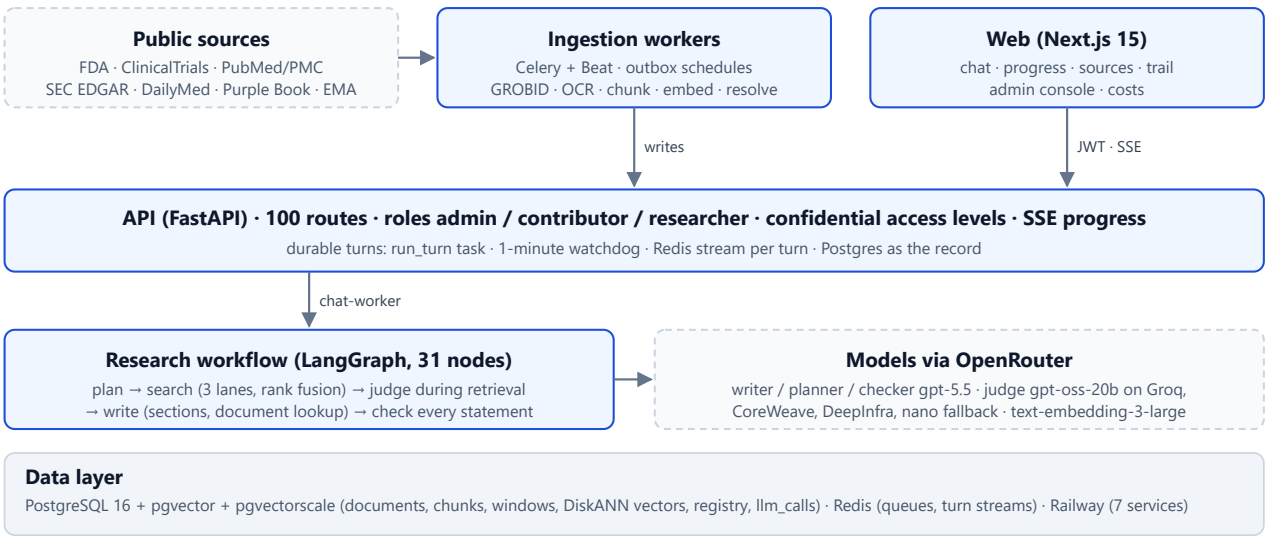
31
LangGraph nodes across research, turn and writer graphs

2 Requirements I derived and built against

Requirement	How the system meets it

Keep the library current without a person	8 connectors (ClinicalTrials.gov, openFDA, FDA 510(k)/PMA, DailyMed, Purple Book, PubMed/PMC, SEC EDGAR, EMA) on per-source cadences stored in Postgres; a 15-minute Beat tick and an outbox table give one dispatch per slot and one catch-up after an outage. EDGAR runs on its own queue at concurrency 1.
Read PDFs properly	GROBID for scientific papers, PyMuPDF in a resource-limited subprocess for regulatory filings, model OCR for scanned pages; section-aware chunks of ~900 tokens with tables kept whole.
Know which product a passage is about	A seeded registry (Purple Book, DailyMed, client list) resolved in three tiers: structured fields, word-boundary alias scan, then an LLM pass only when nothing matched. Short or common-word aliases need two capitalised mentions or the marketer nearby.
Find the one decisive sentence	Three lanes: chunk cosine, keyword rank, and 3-sentence sliding windows; fused by reciprocal rank (k = 60), top 10 per product per search, near-duplicates folded at 0.99.
Stay fast at 668k passages	pgvectorscale StreamingDiskANN with product labels on a narrow vectors table kept by triggers; shortlists rescored exactly; anything not fully indexed falls back to the scan.
Keep only evidence that answers the question	A judge model reads every candidate during retrieval in batches of 5 at concurrency 24, with a fixed schema so the prompt prefix caches, routed to a small open model across three hosts with a fallback.
Never cite what the passage does not say	A checker judges every statement against its cited passages; missing citations publish with visible warnings, invalid or unsupported ones block. The writer can ask a reader for notes whose quotes are verified verbatim by code.
Show the work and the cost	Durable turns on Celery with a watchdog, a Redis stream for live progress, a five-stage progress view, a source panel per citation, an audit trail, and a cost row per model call.

3 Architecture



Workers are isolated by queue so a slow SEC crawl never delays a question. Vectors live in Postgres so entity filters stay in one join; there is no separate vector database and no LangChain.

4 How it works

A QUESTION, START TO FINISH

1. **Plan.** The planner turns the question into up to 10 searches, each one fact with a source kind, and may exclude document types with a stated reason. Low reasoning effort cut planning from 49 s to about 15 s.
2. **Search.** Each search runs the three lanes per product in scope; results are rank-fused and deduplicated. Sentence-window lookups start beside the scans.
3. **Judge.** As passages arrive, the judge reads each one against the question itself and keeps only what supports an answer, writing a short finding.
4. **Write.** The writer composes per-product sections in parallel above a size threshold, citing passages by number; it can ask a reader model for notes on a document, and only verbatim-verified quotes survive.
5. **Check.** Every statement is checked against its citations in parallel batches, at most two attempts; only changed statements are re-checked. Unsupported claims are stripped or the answer is refused with a fixed sentence.
6. **Show.** The reader sees stage progress live, then the answer with a source panel per citation, the trail of what was searched and kept, and the cost.

5 Agent engineering

- **Judge economics:** judging during retrieval, a fixed passage-slot schema for prefix caching, batch 5, and a 20B open model on fast hosts took the judge from $1.36 to $0.07 per broad question.
- **Index with a safety net:** the exact scan is the default and the reference; the DiskANN path is switched per environment and verified at 99.9% agreement before production used it.
- **Memory like a chat:** the model sees the thread verbatim; compaction into a rolling summary happens only at half the window and records the access level it was built under.
- **Scope guard:** product pins are enforced once for every tool; out-of-scope products get a typed refusal that offers to widen the scope rather than a silent wider search.
- **Cost per call:** every model call stores tokens, cached tokens, latency and cost (provider-reported or estimated and flagged), so cost per question is a query.
- **Reasoning effort per stage:** set explicitly for planner, judge, writer and checker after measuring that defaults differed by model.

6 Key engineering decisions

Problem	Decision
A product name is also a common word	Stoplist and short-alias rules needing two capitalised mentions or the marketer nearby; search terms stored as provenance, never as tags. One rule removed 1,951 false tags on one product.
Blended scores from unlike lanes	Reciprocal-rank fusion with per-lane floors instead of a weighted sum of incomparable numbers.
Label updates too slow through HNSW	Labels live in a narrow vectors table maintained by triggers; rewriting chunk rows cost ~140 ms each.
Hallucinated quotes in notes	Code checks every quote verbatim against the text the reader saw; unverifiable quotes are withheld.
Long turns dying silently	Celery turns with no auto-retry, a one-minute sweep that judges silence, and a Redis stream so a reconnecting browser catches up.

Regulator rate limits	A dedicated single-concurrency queue for SEC, cadence per source, and one catch-up after downtime.

7 Results

Measure	Result
Hard regulatory question (49 products)	**348 s / $3.74 → 110 s / $1.07**, answer accurate
Broad clinical question in production	**273 s / $0.90 → 98 s / $0.13** with the filtered index, zero fallbacks
Judge cost per broad question	**$1.36 → $0.07**
Deduplication stage	**5–21 s → 1–6 s**
Client acceptance set	**every regulatory number verified correct** against the FDA record
Production, first 60 days	**230 questions**, median **$0.12** each

STACK Python · FastAPI · LangGraph · Celery + Beat · Redis · PostgreSQL 16 + pgvector + pgvectorscale · SQLAlchemy 2.0 (async) · Alembic · GROBID · PyMuPDF · OpenRouter (GPT-5.5, gpt-5.4-mini, gpt-oss-20b on Groq / CoreWeave / DeepInfra) · text-embedding-3-large · Next.js 15 · TypeScript · Tailwind v4 · Docker · Railway · GitHub Actions

LINKS Watch demo ↗

5.2 · Cure Health — Unified Digital Health Platform LIVE HEALTHTECH

Client: Kognio AI product, designed for a European, GDPR-first market · **Role:** sole engineer · **Timeline:** Jul – Aug 2026, 30 commits · **Live:** curehealth.kognio.ai ↗

Five role-scoped portals connected by one care loop: book → visit → e-prescription → pharmacy dispensing → insurance adjudication → patient financing. Sixteen AI features sit inside the loop, each limited to its role and grounded in the same records and rules the portals use, plus in-browser WebRTC telehealth.

THE PROBLEM

- Patients, providers, pharmacies and insurers each have their own system; the hand-offs are paper and phone.
- AI "assistants" bolted on later skip the business rules.
- Health data needs audit and least privilege from day one.

WHAT I BUILT

- A FastAPI + Next.js 16 platform with 46 routes, 26 screens and 11 tables.
- Four enforced state machines with coverage and instalment math in code.
- A per-role AI tool registry where the booking assistant calls the real booking function.

THE OUTCOME

- One coherent demo chain from booking to a 70%-covered claim and a 6-month plan.
- 62 backend tests with every external call mocked; CI on each push.
- Every AI call audited without storing message content.

1 By the numbers

5
roles, each with its own portal

16
AI features: 14 in the per-role registry plus Care Navigator and Care Plan

46
API routes (45 HTTP + 1 WebSocket)

4
enforced state machines: prescription, pharmacy order, claim, financing

62
backend tests, no network needed

26
screens

2 How it works

THE CARE LOOP

1. **Book.** A patient books a slot (409 if taken); fees are set per specialty; video visits get a room id. The AI booking assistant runs a tool loop over list providers, get slots and book, capped at 6 rounds and one booking per turn, calling the same function as the form.
2. **Visit.** The provider marks the visit and drafts a S/O/A/P note from bullets or a transcript; interaction checks are refused unless a treatment relationship exists.
3. **Prescribe and dispense.** Prescription active → sent → dispensed; the pharmacy order moves received → dispensing → ready → completed, with illegal moves returning 409. Pharmacy tools check dispensing, suggest at most three substitutions with a prescriber note, print labels in English or Albanian, and forecast demand from 30-day aggregates.
4. **Claim.** Submitted → in review → approved or denied → paid. Four checks run in code (policy active, source valid, no duplicate, amount within source); the claims copilot only recommends, and bad output becomes needs-info. Covered amount = amount × coverage %, patient share = the rest, shown on an EOB.
5. **Finance.** The patient's share becomes a 3, 6 or 12-month plan with ceiling-divided instalments; the plan completes when the last one is paid.

3 Key engineering decisions

Problem	Decision
AI skipping the rules	The booking assistant calls the same perform_booking as the form; claim checks, coverage math and forecasts are computed in code and the model only narrates.
Role leakage	A tool registry keyed by role, 403 on any cross-role call, pinned by a test that tries every tool with every wrong role.
Health data in logs	Append-only audit rows committed in the same transaction, with AI calls logged by key and no message content.
Records revealing their existence	Ownership checks return 404, not 403.
Tests needing a model key	In-memory SQLite and a monkeypatched model client; bad model output is replaced with the safest valid value.

STACK Next.js 16 · React 19 · TypeScript · shadcn/ui · TanStack Query · FastAPI · async SQLAlchemy · Alembic · PostgreSQL 16 · WebRTC + WebSocket signaling · OpenRouter (Claude) · GitHub Actions · Docker · Railway

LINKS Open the live app ↗ · Case study on GitHub ↗

06 · AI Sales & Outreach

1 CASE STUDY

6.1 · Speed-to-Lead — Multi-Channel AI Intake & Booking Engine DEMO

AI SALES & OUTREACH

Client: Kognio AI portfolio build, demo client Kognio AI itself; positioned for marketing agencies running many clients on GoHighLevel · **Role:** sole engineer · **Timeline:** Sep 2026, 49 commits; parked at $0/month after recording · **Demo:** Watch ↗

One lead-qualification engine behind three doorways: website chat, a voice agent and web forms. Every lead is scored against rules the business owns in Airtable, enriched from its own website, booked onto the live calendar mid-conversation, and written to whichever CRM the client already uses. Change a cell in Airtable and the behaviour changes with no deploy.

THE PROBLEM

- Leads that wait an hour are lost; most businesses cannot answer at 2 AM.
- Each client's services, area, hours and scoring are different.
- Agencies will not maintain code per client.

WHAT I BUILT

- Seven n8n workflows (98 nodes): intake, two CRM adapters, voice, prospects, metrics, dead letter.
- An Airtable control plane of 7 tables and 73 fields that a non-developer edits.
- A Vapi voice assistant that books into GoHighLevel through plain HTTP tools.

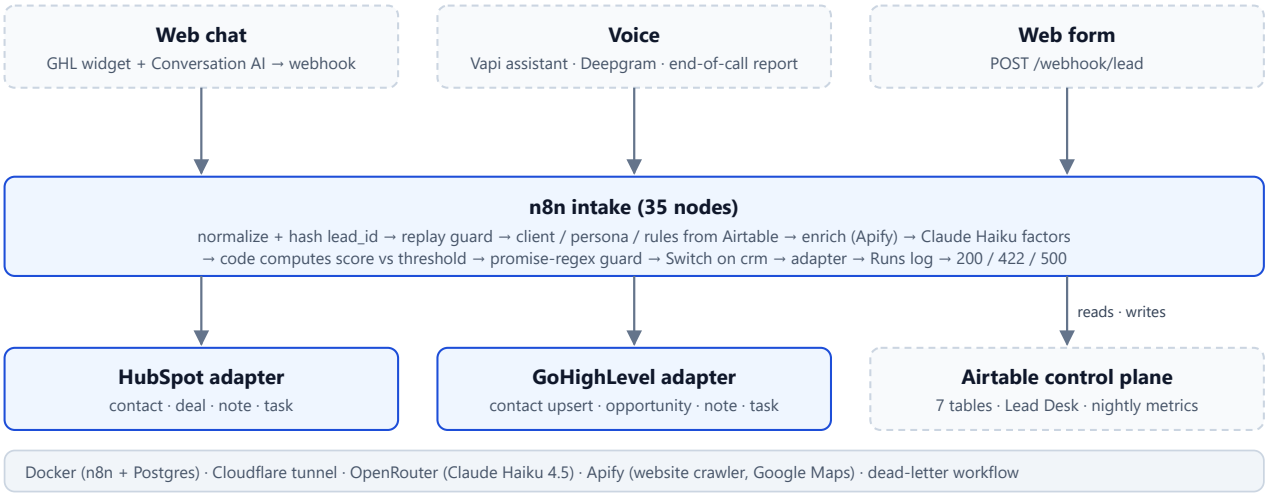
THE OUTCOME

- Form to scored CRM record in about 20 s; chat to booked call in about a minute.
- Re-pointed from roofer to agency to AI studio with three Airtable rows and no code.
- 10 of 10 exit tests passed; every failure answers 500, never a silent 200.

1 By the numbers

~20 s form to scored CRM record (21.3 s in the demo)	**~1 min** chat to booked discovery call, no human	**7 / 98** workflows and nodes	**2** CRMs (GoHighLevel, HubSpot) switched by one Airtable field
3 rows to re-point the engine to a new industry	**7 / 73** Airtable tables and fields in the control plane	**10 / 10** exit tests passed	**$0** per month when parked; about a day to bring back

2 Architecture



Three doorways post the same lead shape to one intake. The model returns factors between 0 and 1; code multiplies them by the weights in Airtable, so the business controls the score without touching the prompt.

3 How it works

1. **Normalize.** Validate, build a stable lead id from channel, phone or email and message, and normalize the phone to E.164. Bad payloads get 422.
2. **Replay guard.** A lead already in Airtable returns 200 "already recorded" and creates nothing.
3. **Load the client.** Client, persona and qualification rules come from Airtable rows; an unknown client is rejected.
4. **Enrich.** The lead's website (explicit, or from a business email domain) is crawled by Apify with a 15 s cap; failure is recorded, never fatal.
5. **Score.** Claude Haiku returns four factors (service match, in area, urgency, budget signal), a disqualifier, an escalate flag and a draft reply. Code computes the weighted score, applies the threshold, and a regex catches replies that promise a time.
6. **Write.** A Switch on the client's CRM field calls the HubSpot or GoHighLevel adapter, which upserts the contact, creates or updates one opportunity, adds a note and a task, and returns the id.
7. **Log.** A Runs row records stage, outcome and latency; any failure goes to Shape Run (failed) → Log Failed → Respond 500.

4 Key engineering decisions

Problem	Decision
The model would not do weighted arithmetic	Model returns factors 0..1; code computes the score and writes the sum into the reason field.
The model echoed the lead's deadline as a promise	Prompt rules plus a regex guard that sets needs_review; the rule is enforced twice.
Failures answering 200	Twelve I/O nodes route errors to a failed-run shaper and a 500 response; an error workflow catches the rest.
Two opportunities per contact	The GHL adapter searches by contact and updates the existing opportunity, keeping a booked stage.
Voice tools needing OAuth that never worked	Three plain HTTP tools and a slots endpoint that does the date maths, so the model never computes timestamps.

Per-client code	Everything client-specific is an Airtable row; the HubSpot adapter was built first on a free tier to fix the contract before the GHL trial clock started.

STACK n8n · Claude Haiku 4.5 (OpenRouter) · Vapi (Deepgram nova-3, Clara voice) · GoHighLevel API · HubSpot API · Airtable + Interface Designer · Apify · Cloudflare Tunnel · Docker · PostgreSQL

LINKS Watch demo ↗

07 · Conversational AI / Telecom

1 CASE STUDY

7.1 · Omvi William — Multi-Tenant Agentic AI SaaS with Real-Time Voice

FLAGSHIP DEMO

CONVERSATIONAL AI / TELECOM

Client: World Mobile (telecom operator), delivered through Omvi AI · **Role:** architect and engineer, designed and built end to end · **Timeline:** Oct 2025 – Mar 2026, demo platform Oct 2026 · **Demo:** Watch the walkthrough ↗

A white-labelled AI assistant, "William," that people can talk to or type to. It answers from the operator's own knowledge base, acts in email and calendar, remembers earlier conversations, and turns a voice call into a project that a team of five AI agents completes while the user watches.

THE PROBLEM

- One assistant for customers and staff, by voice and by text.
- Answers from the company's own material, not the open web.
- Fast enough on a live call that it never feels like a bot.

WHAT I BUILT

- One LangGraph brain behind both chat and voice.
- Six services: gateway, graph server, voice agent, RAG, orchestrator, Studio.
- Multi-tenant from day one: 6 roles, scoped state, cross-org guard.

THE OUTCOME

- Agent starts speaking 2.0 s after the question ends (median).
- A call becomes up to 20 tasks that five agents finish in 15–20 s each.
- Sellable to the next client without a rebuild.

1 By the numbers

2.0 s end of question → agent starts speaking (median, 9 live calls)	**3.9 s** end of question → grounded knowledge answer starts	**35 → <10 s** first reply after a cold start	**15–20 s** per task through all five agents
6 services around one agent brain	**15 + 6** LangGraph nodes in the chat and voice graphs	**25** agent tools (12 email, 7 calendar, 6 general)	**6** RBAC roles in the multi-tenant model

2 The problem

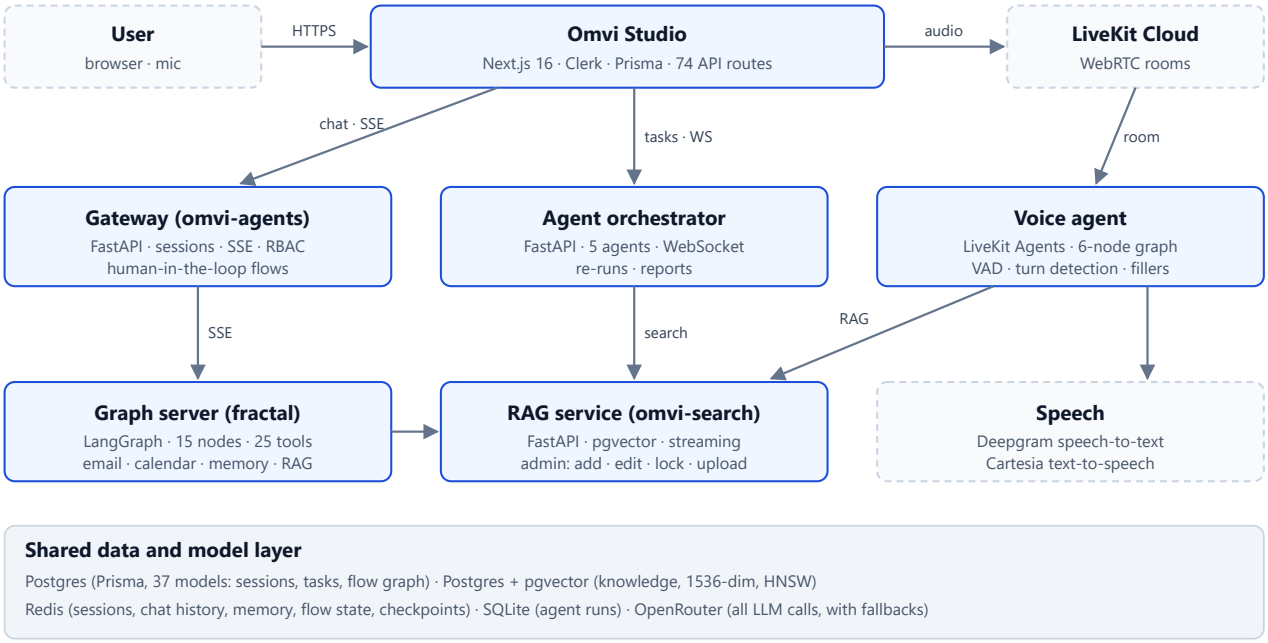
- **One assistant, two channels:** customers and staff should get the same answers whether they talk or type.
- **Grounded, not generic:** answers must come from the operator's own knowledge base, and the assistant must say so when it finds nothing.
- **Act, not just answer:** email, calendar and tasks, with confirmation before anything is sent.
- **Feel instant on a call:** long silences make a voice assistant feel broken.
- **Sellable again:** the vendor, Omvi AI, needed the product to serve the next telecom or enterprise client without rebuilding it.

3 Requirements I derived and built against

Requirement	How the system meets it
Talk or type, and get the same answers	One LangGraph agent behind both: chat streams over SSE through the gateway; voice runs the same reasoning inside a LiveKit worker.
Answer from the operator's knowledge, not the web	pgvector RAG over the company corpus with neighbouring-passage context and a similarity floor; says so plainly when nothing relevant is found.
Take actions in email and calendar	Dedicated intent resolvers with 12 email and 7 calendar tools over Google and Microsoft OAuth; drafts are confirmed before anything is sent.
Remember the user	Temporal and cross-session memory recall nodes over Redis chat history, with safeguards against fabricated recall.
Guide new users	Human-in-the-loop onboarding and daily-reminder flows that pause for the user with LangGraph `interrupt()` and resume where they left off.
Turn conversation into work	A call transcript becomes a titled session with up to 20 tasks across 8 lanes and their dependencies; five agents then complete each task.
Feel instant on a phone-style call	Under 2.5 s from the end of a question to the agent speaking, measured on the deployed system.
Serve many clients from one codebase	Organization-scoped state, 6 roles mapped to `module:action` scopes, per-endpoint scope checks and a cross-organization guard.
Let admins control the knowledge	Knowledge entries can be added, edited, locked against changes, deleted, or uploaded as PDF, DOCX, HTML or text.
Show cost and progress	Per-agent output, timing and token cost; a live usage dashboard; notifications when agents finish.

4 Architecture

Six services, so the agent, the voice runtime and the task agents can each scale and ship on their own while sharing one brain and one data layer.



Six services (blue) around one agent brain; dashed boxes are external. Chat flows Studio → gateway → graph server; voice flows LiveKit → voice agent; tasks flow Studio → orchestrator, with progress pushed back over WebSocket.

THE SIX SERVICES

- **Omvi Studio** (Next.js 16, Clerk, Prisma): the workspace. Sessions, chat, the voice call screen with live transcript, Board / Grid / Canvas task views, email and calendar cards, dashboard, knowledge admin, notifications, search.
- **Gateway** (FastAPI): the single stateful entry point. Redis sessions and history, SSE streaming, role and scope checks, LangGraph flows that pause for user input.
- **Graph server** (FastAPI + LangGraph): the agent brain, split out so it ships independently. Streams tokens back to the gateway over SSE.
- **Voice agent** (LiveKit Agents): joins each call room, listens, reasons with its own graph and speaks; kept warm so it joins quickly.
- **RAG service** (FastAPI + pgvector): retrieval and grounded answers, streamed or whole, plus the knowledge admin API.
- **Agent orchestrator** (FastAPI): watches for new tasks and runs each through five agents, pushing progress to the Studio over WebSocket.

5 How it works

A CHAT TURN

1. **Route.** A main orchestrator node decides the turn. Email and calendar requests go to their own decision → intent resolver → action chains; everything else passes an LLM domain classifier.
2. **Ground.** Knowledge questions retrieve from the RAG service; personal questions hit temporal or cross-session memory recall.
3. **Answer.** A tool-calling speaker node writes the reply, bound to knowledge search, web search, time and task creation. Only answer-writing calls stream to the user; routers stay internal.
4. **Pause when needed.** Onboarding, daily-reminder and content flows stop with `interrupt()`, checkpoint in Redis, and resume days later with `Command(resume=…)`.

A VOICE CALL

1. **Join.** A warm worker joins the LiveKit room with no cold start. Silero VAD, a multilingual turn detector and noise cancellation handle the audio.

2. **Listen.** Deepgram transcribes; a six-node graph (decide, greeting, general, memory recall, retrieval, content) handles the turn.
3. **Fast route.** A clear product question skips the LLM router and goes straight to retrieval.
4. **No dead air.** A short filler line is spoken the moment retrieval starts; the answer then streams sentence by sentence through Cartesia.
5. **Call to project.** On hang-up the transcript becomes a titled session with 1–20 tasks, lanes and dependencies.

FROM TASKS TO FINISHED WORK

1. **Pick up.** The orchestrator takes new tasks as they are written and also polls the task table every 3 seconds, because voice-created tasks never call its API.
2. **Run.** Up to three tasks at a time go through Omni (research), Planner, Builder, Reviewer and Nexus (write-up).
3. **Show.** Each stage streams lane transitions to the Studio and stores output, timing and tokens; the last stage produces a downloadable report. Refine re-runs with a user instruction instead of starting over.

6 Agent engineering

- **Streaming end to end:** on Python 3.10 nested LLM calls only reach the graph's event stream when a node hands down its run config, so each answer node passes it explicitly; the RAG service's own stream is relayed through the graph as custom events.
- **Graph controls speech:** I patched the LiveKit LangChain adapter so the graph, not the plugin, decides what is spoken.
- **Resilience:** every model call goes through OpenRouter with ordered fallback models and retries, so a rate-limited provider becomes a slower answer, not an error.
- **Knowledge:** documents chunked to about 1,800 characters, embedded with `text-embedding-3-small` into Postgres with an HNSW cosine index. Each query retrieves at least 8 passages above a similarity floor plus the passage after each of the top four. Answers by Gemini 2.5 Flash Lite with GPT-4.1 mini and GPT-4o mini as fallbacks.

7 Multi-tenancy and platform

- **Tenant isolation:** every request carries an organization, role and scopes; Redis state is keyed `{namespace}:{org}:{user}:{session}`; a middleware guard blocks cross-organization access.
- **Roles and scopes:** six roles (platform admin through client viewer) map to `{module}:{action}` scopes, enforced per endpoint. I authored the architecture plans the team adopted for serving many clients from one codebase.
- **Delivery:** shipped to GCP through GitHub Actions, later as 8 services on Railway (six apps plus Postgres with pgvector and Redis) with auto-deploy on merge and Playwright browser audits before release.

8 Key engineering decisions

Problem	Decision
Chat and voice drifting apart	One LangGraph brain; the voice worker runs the same reasoning and a patched adapter lets the graph control speech.
Silence while the agent thinks	Skip the router for clear product questions, speak a filler immediately, stream the answer by sentence.

Answers arriving all at once	Pass run config into answer nodes, tag user-facing calls, relay the RAG stream as graph events.
Slow first request after deploy	Build and compile the agent graph at startup; first reply went from 35 s to under 10 s.
Provider outages and rate limits	One OpenRouter gateway with ordered fallback models and retries in every service.
Voice-created tasks invisible to the agents	Orchestrator watches the task table directly, in addition to its trigger API.
Each new client needing a fork	Organization-scoped state, role-to-scope permissions and per-endpoint checks from day one.

9 Results

Voice latency measured on the deployed system on 3 November 2026: nine live calls, each a spoken question played into a browser microphone, timed from the live transcript.

Moment	Median	Range
Click "Voice" → agent greets the user	**7.5 s**	7.4 – 10.1 s
End of a knowledge question → agent starts speaking	**2.0 s**	1.7 – 2.6 s
End of a knowledge question → grounded answer starts	**3.9 s**	3.5 – 4.5 s
End of a general question → answer starts	**4.0 s**	2.8 – 4.0 s

- All nine spoken questions were transcribed correctly and answered from the knowledge base, including exact plan prices.
- A task created from a call finishes all five agents in about 15–20 seconds; Refine re-runs in about 13 seconds.
- The first reply after a deploy dropped from 35 s to under 10 s.

STACK Python · LangGraph · LangChain · FastAPI · LiveKit Agents · Deepgram · Cartesia · Silero · PostgreSQL + pgvector · Redis · SQLite · OpenRouter (Gemini 2.5 Flash Lite, GPT-4.1 mini, GPT-4o mini) · SSE · WebSockets · Next.js 16 · TypeScript · Prisma · Clerk · Docker · Railway · GCP · GitHub Actions · Playwright

LINKS Watch demo ↗

08 · EdTech / LMS

1 CASE STUDY

8.1 · HSC Chemistry — AI-Assisted E-Learning Platform

DEMO EDTECH / LMS

Client: an HSC Chemistry tutoring platform for Year 12 students in New South Wales · **Role:** full-stack and AI engineering · **Timeline:** 2025 · **Demo:** Watch ↗

A learning platform with modules, lessons, practicals, quizzes, exams and games, where the model generates practice questions in five formats, marks free-text answers against a rubric, and writes per-question exam feedback that becomes a PDF report. Analytics run inside Postgres.

THE PROBLEM

- Students need unlimited practice questions in the exam's own formats.
- Marking short and long responses by hand does not scale.
- Teachers want to see who is struggling, per module.

WHAT I BUILT

- An Express API of 193 handlers over a 28-table Supabase schema.
- Seven model-backed endpoints behind one config-driven gateway.
- A 76-page Next.js app with student and admin areas.

THE OUTCOME

- Generate, mark and report on exams with validated, repaired model output.
- Admins change model, temperature and token limits from a settings table.
- 113 API test cases across five suites.

1 By the numbers

7
model-backed endpoints through one gateway

5
question formats: MCQ, short answer, calculation, long response, true/false

28 / 26
tables and migrations

193
REST handlers across 33 routers

76
Next.js pages

113
API test cases

2 How it works

1. **Configure.** Every call loads the default row of a model-configuration table (model, temperature 0–2, up to 4,000 tokens, one of four prompt styles), so admins tune without a deploy.
2. **Generate.** Templates per question type ask for JSON; fields and types are checked, bad items dropped, fences and trailing commas stripped, and the call retried up to three times. Timestamps and seeds in the prompt push variety.
3. **Mark.** One prompt handles every type with the model answer as the rubric; inputs are capped, MCQ letters mapped to option text, the score clamped to the available marks, and a fixed fallback returned on failure.
4. **Report.** Exam feedback explains each MCQ option and scores each written answer with strengths and weaknesses; totals roll up per section and topic, and pdfkit renders the report.

5 **Analyse.** A plpgsql function buckets each module's students by average into four performance bands; about 20 admin analytics endpoints and materialized progress views feed the dashboard.

3 Key engineering decisions

Problem	Decision
Model output is unreliable JSON	Validate, clamp, repair, retry, and fall back; never trust a raw parse.
Tuning needs a developer	Model settings and prompt styles live in a database table with limits enforced at the database level.
Heavy analytics in app code	Performance distribution, smart-mix exams and progress are SQL functions and materialized views.
Admin changes need a trail	Admin write operations are wrapped in an activity log.

STACK Next.js 15 · React 19 · Redux Toolkit / RTK Query · shadcn/ui · Recharts · Express 5 · Supabase (Postgres, Auth, RLS) · Cloudflare R2 · OpenAI Chat Completions · pdfkit · Jest + supertest · Vercel

LINKS Watch demo ↗

09 · Content / SEO

1 CASE STUDY

9.1 · AI SEO Blog Generator DEMO

CONTENT / SEO

Client: Kognio AI product · **Role:** sole engineer · **Timeline:** 2026 · **Demo:** Watch ↗

One keyword, a tone and a length in; a publish-ready article out, with meta description, title options, slug and keyword analysis. A four-stage pipeline writes each section with its own token budget, then a polish pass restores the flow. A content library, inline editor, Google search preview and DOCX export sit on top.

THE PROBLEM

- One-shot prompts produce thin, repetitive long-form content.
- Writers need the SEO parts (meta, titles, keywords) as structured data, not prose.
- Cost needs a cap per user.

WHAT I BUILT

- A FastAPI pipeline of 7–11 sequential model calls per article.
- A Next.js 15 app with library, editor, SEO details and DOCX export in the browser.
- A generations table recording tokens and errors per run, and a daily quota.

THE OUTCOME

- Keyword research → outline → sections → polish, each structured and inspectable.
- Three length tiers and five tones applied through the prompts.
- 50 generations per user per day, enforced with HTTP 429.

1 By the numbers

4
stages, 7–11 model calls per article

600 / 1,200 / 2,000
target words for short, medium, long

12
REST endpoints across 3 routers

3
tables: users, blog posts, generations

50
generations per user per day

~3.8k
lines of code

2 How it works

1. **Keyword research.** JSON with the primary keyword, 5–8 secondary and LSI keywords, search intent and difficulty; an optional brief is folded in.
2. **Outline.** JSON with the H1, three title suggestions, a 150–160 character meta description, a slug, and 4–8 H2 sections with H3s and a brief each.
3. **Sections.** One call per section in markdown, with the tone and the per-section word target.
4. **Polish.** A lower-temperature pass smooths transitions and adds the intro and conclusion; the post, outline and title options are saved and the run logged.

3 Key engineering decisions

Problem	Decision
Length limits on one prompt	Section-by-section generation with a final polish pass.
Model lock-in	OpenRouter behind the OpenAI SDK; the model is one environment variable.
Runaway cost	A daily quota on the user row and a generations audit table with token usage.
Server-side document dependencies	DOCX built in the browser from the rendered HTML, loaded only on click.

STACK Next.js 15 · React 19 · TypeScript · Tailwind · Radix · react-markdown · FastAPI · PostgreSQL · SQLAlchemy (async) · Alembic · OpenRouter (Claude Sonnet) · html-docx-js · Docker

LINKS Watch demo ↗

10 · AI Infrastructure

1 CASE STUDY

10.1 · OptiML — LLM Compression & Fine-Tuning Framework

AI INFRASTRUCTURE

Organisation: OptiML (Log Two LLC), Apache-2.0-licensed framework · **Role:** lead developer on the core framework; top contributor by commits · **Timeline:** Apr 2024 – Jan 2026

A framework that makes compression part of fine-tuning: pruning, quantization, LoRA and multi-head latent attention as composable pieces of one training loop, driven by one config. I redesigned the core from a monolithic tuner into an event-driven system of callbacks, typed events and pluggable metrics, so a new technique is a callback rather than a change to the loop.

THE PROBLEM

- Each compression technique came with its own training script and duplicated statistics code.
- Adding a method meant editing the training loop itself.
- Configuration was scattered across code and scripts.

WHAT I BUILT

- Config as data: a layered YAML template, Pydantic validation and a fluent builder.
- An event-driven Tuner with a dispatcher, 23–25 callbacks and 35 typed events.
- Projection interfaces for pruning patterns and pluggable importance and Hessian metrics.

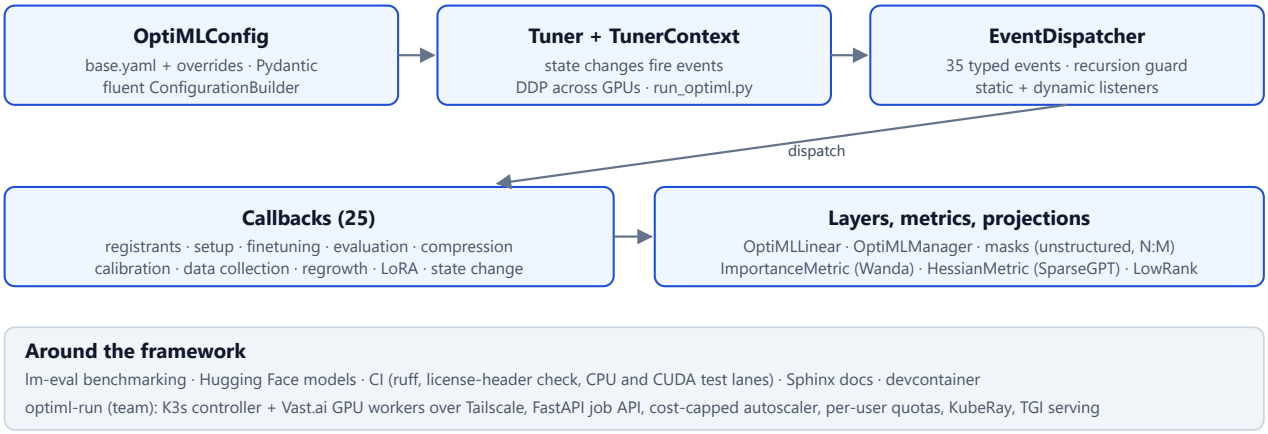
THE OUTCOME

- Magnitude, Wanda and SparseGPT pruning in three sparsity patterns on one API.
- LoRA, QLoRA, QAT, AWQ, MLA and regrowth built on feature branches.
- Multi-GPU training via DDP and lm-eval benchmarking built in.

1 By the numbers

430 commits, about 41% of the repository; top contributor	**15** PRs authored, 5 merged to main	**+18.4k / −12.5k** lines across the 5 open architecture PRs	**23 → 25** lifecycle callbacks; 35 typed events
6 techniques added on branches: LoRA, QLoRA, QAT, AWQ, MLA, regrowth	**3 × 3** pruning algorithms × sparsity patterns on main	**1.42%** perplexity rise for SparseGPT at 50% sparsity on LLaMA-2-7B (team result)	**9** feature branches of 50–85 commits each

2 Architecture



Config flows into a Tuner whose state changes fire typed events; callbacks subscribe and do the work, from calibration pruning to regrowth during training. Pruning patterns and metrics are looked up from config by name.

3 How it works

1. **Configure.** A YAML layered on a full default template, validated into one typed object; callbacks are built from the config.
2. **Load and shard.** Models load across GPUs with DDP; linear layers are wrapped as prunable layers with mask projections.
3. **Collect.** A data-collection callback gathers activations in continuous, sampling or calibration mode; metrics (Wanda's running average, SparseGPT's Hessian) accumulate.
4. **Compress.** Calibration pruning runs one-shot: collect → mask → log sparsity → optional eval; regrowth and compression callbacks act during fine-tuning on logical and micro steps.
5. **Adapt and merge.** LoRA as $W + \alpha/r \cdot BA$ with a low-rank metric and weight merging; adapters saved and loaded by the manager.
6. **Evaluate.** lm-eval runs through the framework's model wrapper; results feed the team's benchmarks.

4 Key engineering decisions

Problem	Decision
Every technique editing the training loop	Event-driven Tuner: private state changes only through update(), which fires events; setup callbacks remove themselves after one use.
Pruning patterns as ad-hoc functions	Projection interfaces with a registry and aliases, so a new pattern is a class named in config.
Statistics duplicated per algorithm	Metrics separated from algorithms behind one interface, chosen by a discriminated union.
Compression tangled with data collection	Split into separate callbacks, which made regrowth during training possible.
Config scattered across scripts	One base template with overrides and Pydantic validation; the Tuner takes a single object.

5 Results

- Five PRs merged to main: model loading, pruning-pattern refactor, combined compression and fine-tuning workflows, the config system, and the event-driven Tuner.
- Five open stacked PRs carry the typed event system, pluggable metrics, the 22-callback Tuner split, dynamic sparse training, and LoRA.
- Team benchmark on LLaMA-2-7B at 50% sparsity: SparseGPT +1.42% perplexity, Wanda +6.48%, magnitude +151%; task-specific calibration gave +30.99% exact match on one downstream set.

STACK Python · PyTorch · Hugging Face Transformers · Pydantic · DDP / NCCL · lm-eval · bitsandbytes · ruff · GitHub Actions · Sphinx · Docker · Kubernetes (K3s) · Ray · Text Generation Inference

11 · HR Operations / Agent Tooling (MCP)

1 CASE STUDY

11.1 · AI Onboarding Ops — Onboarding as an MCP Server LIVE

HR OPERATIONS / AGENT TOOLING

Client: Kognio AI product · **Role:** sole engineer · **Timeline:** Sep 2026 · **Live:** onboarding.kognio.ai ↗ · **Code:** Case study on GitHub ↗

Employee and client onboarding delivered two ways from one codebase: a web console for people, and a Model Context Protocol server that Claude Desktop, Claude in Slack or any agent can call with a personal token. Twenty-five narrow tools sit behind a single guard, so a button, a chat message and an MCP call produce the same audit row.

THE PROBLEM

- Onboarding is a checklist spread over HR, IT and managers, with no record of who did what.
- Agents given direct database access change data with no trail.
- Policy answers drift from the handbook.

WHAT I BUILT

- One tool registry; three front doors (REST writes, chat agent, MCP) through one guard.
- A state machine for employee, contractor and client onboarding with document gating.
- Policy Q&A over a versioned handbook with BM25 retrieval and citations.

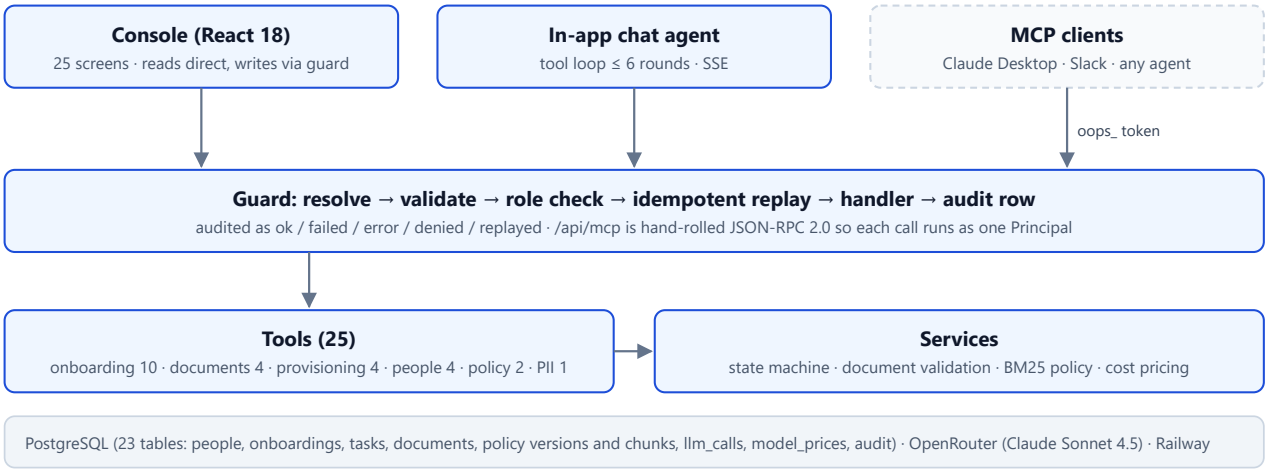
THE OUTCOME

- One instruction starts an onboarding; tasks stay closed until the document is accepted.
- Every model call priced at write time from a versioned price table.
- Restricted personal data behind a tool never exposed to chat or MCP.

1 By the numbers

25 tools registered, 24 exposed over MCP	**3 → 1** front doors through one guard	**6** guard steps, audited on every outcome	**3 + 3** MCP resources and prompts
3 templates (employee 13 tasks, contractor 6, client 10) on one state machine	**5** roles with row-level scoping	**69** REST endpoints · 25 screens	**34** tests, 13 of them on the guard

2 Architecture



Console reads bypass the guard so the audit log stays free of page-view noise; every write, from any door, goes through it.

3 How it works

AN EMPLOYEE ONBOARDING

1. **Start.** One tool call creates the person, the onboarding, the tasks from the template, and a document request per required document with its own upload link.
2. **Gate.** Completing a task that needs paperwork is refused until the document's status is validated; validation moves the waiting task to in progress.
3. **Validate.** Documents are checked deterministically: MIME allow-list, minimum size per type, a screenshot heuristic for signatures. No model decides.
4. **Provision and notify.** Separate tools provision accounts, schedule a check-in, post to a channel or update the CRM stage; only Slack and Google Calendar have live paths, the rest record a fallback.
5. **Finalize.** Status is recomputed from the tasks; finalize is refused while any task is open. The client flow (contract → workspace → billing → security review → kickoff → go-live) runs on the same machine.

4 Key engineering decisions

Problem	Decision
Three ways to change data, one trail	One guard for console writes, the chat agent and MCP; narrow tools so the audit log reads as sentences.
Agents and personal data	The PII tool is registered but flagged off for chat and MCP; audit rows redact sensitive keys.
Policy answers from the wrong version	Handbook versions supersede rather than overwrite; BM25 (k1 1.5, b 0.75) over active chunks; refuse with a fixed sentence when silent; the verbatim excerpt is returned when no model is configured.
Embeddings for a few hundred sections	BM25 keeps the product free of a second paid service and OpenRouter has no embeddings endpoint.
Cost drifting with price changes	Prices in a table with effective dates; each call priced when written, failed calls included.
Seed data that lies	The seed drives the real tools, so the audit log and durations are genuine; onboardings are backdated so time-to-productive means something.

STACK Python · FastAPI · SQLAlchemy 2.0 (async) · Alembic · PostgreSQL · MCP (JSON-RPC 2.0) · BM25 · OpenRouter (Claude Sonnet 4.5) · React 18 · TypeScript · Vite · TanStack Router · Tailwind · Docker · Railway

LINKS Open the live app ↗ · Case study on GitHub ↗

---

mjehanzaib99@gmail.com · linkedin.com/in/muhammad-jehanzaib-4036b5195 · github.com/mjehanzaib999