Haier AI Platform

Portfolio case study - retail store compliance, invoice extraction, legal document extraction, infrastructure, training, and deployment

This document presents my portfolio-style summary of the Haier AI program I helped build end to end. I worked across architecture, FastAPI pipeline design, production deployment, model support, evaluation, and operational controls. The largest use case was retail store compliance, where I combined image validation, category support, LLM reasoning, queue control, and monitoring into a production service.

My role across the program

I designed the multi-use-case architecture, built FastAPI and async pipelines, shaped model evaluation strategy, trained supporting CV gates, productionized deployments, and created operational controls for scale, monitoring, and reliability.

Primary use cases

Retail store compliance, invoice extraction, and legal document extraction; all built on a shared platform with S3, Redis, worker orchestration, OCR / CV / VLM layers, and admin-facing APIs or dashboards.

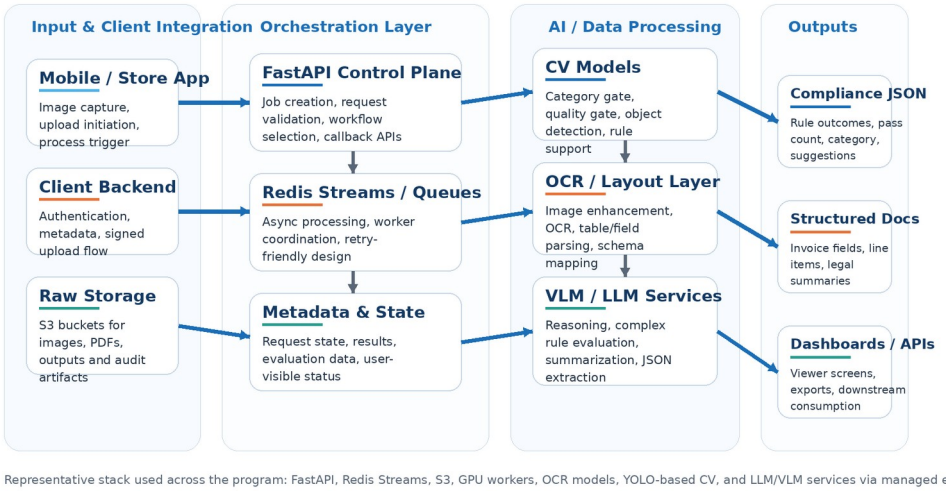
What made this project different

This was not only model experimentation. It required real production engineering: ingest design, traffic isolation, cost-aware routing, quality gates, queue control, monitoring, audit persistence, and deployment decisions.

Stack used across the program

FastAPI, Redis, PostgreSQL, S3, Uvicorn, Nginx, Python workers, OCR tooling, YOLO-based CV, Gemini, Bedrock Qwen, prompt pipelines, and internal admin/reporting services.

Shared Platform Architecture



Representative stack used across the program: FastAPI, Redis Streams, S3, GPU workers, OCR models, YOLO-based CV, and LLM/VLM services via managed e

Representative architecture used across the Haier program: shared ingress, orchestration, storage, AI / data processing, and downstream outputs.

1. Executive summary

I worked on a multi-use-case AI platform for Haier that covered image-based retail compliance, invoice extraction, and legal document extraction. My ownership was not limited to prompts or isolated model experiments; I worked across service design, async orchestration, validation, deployment, observability, and production tuning.

For retail store compliance, I designed the request flow from image intake to compliance JSON output. The system had to accept real store images, reject bad inputs early, choose the right backend under burst, persist all analysis results, and expose both operational and business visibility through an admin / monitoring plane.

For document intelligence workflows, I designed OCR-driven pipelines that separated extraction, parsing, validation, and reasoning, because these use cases required structured output rather than generic free-form summarization. This design decision made the system easier to evaluate, easier to debug, and easier to integrate downstream.

Program scope at a glance

Use case	Primary input	Core AI / approach	Output
Retail store compliance	Store / display images	Quality gate, wrong-category gate, object support, visual rule interpretation, cost-aware Gemini / Bedrock routing	Compliance JSON
Invoice extraction	Invoice scans and PDFs	Image enhancement, OCR, field mapping, schema validation, line-item extraction	Structured invoice JSON
Legal document extraction	Multi-page scans / PDFs	Page-level OCR, segmentation, metadata extraction, legal summarization	Structured case summary

2. Shared platform architecture

Although each use case solved a different business problem, I approached them as parts of one shared platform rather than three isolated scripts. That meant creating reusable foundations for ingress, storage, orchestration, result persistence, worker coordination, and client integration. This also helped me keep the system production-friendly: new flows could be added without rebuilding the entire stack each time.

Cross-program architecture patterns I used	Why this mattered in practice

- FastAPI control plane for validation, routing, and workflow selection
- Raw object storage for uploaded images, PDFs, and generated artifacts
- Redis streams / queues and state keys for async coordination and control-plane counters
- Persistent metadata and result storage for auditability and downstream consumption

- Each use case could reuse the same production patterns instead of becoming a one-off implementation
- Business and operational visibility became easier because requests, outputs, and metrics followed a common shape
- Failures were easier to isolate because ingestion, workers, OCR / CV, and reasoning were separated layers
- Scale tuning could happen incrementally by adjusting one layer without rewriting the whole pipeline

3. Use case 1 - Retail store compliance

This was the most operationally demanding use case. The input was one image captured from a store environment, but the output had to be much richer than a simple yes / no answer. The service had to identify whether the image was even worth processing, determine if the image belonged to the correct product category, pass the valid images through rule reasoning, and return a structured compliance result that downstream systems could consume.

How I approached the problem

- I designed the workflow so that obviously bad images could be rejected early before expensive reasoning was triggered.
- I treated category support as a real production requirement, because sending the wrong category image into rule reasoning creates noisy outputs and wasted spend.
- I kept the final result structured: valid / invalid flags, pass count, total rules, category, quality information, suggestions, and full payload persistence for audit or review.
- I built the service to run with more than one LLM backend so routing, spillover, and failover could be handled during burst or backend-specific failure windows.

Retail pipeline I built

Retail store compliance pipeline

- Image intake through multipart or S3-URL paths, with request validation and idempotent request claiming.
- Cheap pre-checks for invalid image conditions and wrong-category gating before expensive visual reasoning.
- Traffic manager chooses Gemini or Bedrock Qwen, then applies queue control and fallback behavior.
- LLM output is normalized into a structured compliance schema and persisted along with usage / cost metadata.
- Frontend and admin endpoints expose historical metrics, recent requests, and live operational counters.

What the response contained

- Invalid-image flags for blank, blur, irrelevant image, wrong category, and related rejection reasons
- Pass / fail result with pass_count and total_rules
- Classification summary, category, suggestions, and complete payload JSON

Why this structure mattered

- The same output could feed dashboards, QA review, exports, and downstream business consumption
- Field-level response shape made it easier to evaluate weak points systematically
- Full payload persistence reduced debugging time during model or prompt changes

Training and evaluation support I designed around retail compliance

I did not want the pipeline to depend only on a large reasoning model. For production reliability and cost control, I worked toward a layered design where cheaper computer-vision stages could filter or support images before expensive inference. That included thinking about category gating, quality gating, and object-support workflows rather than sending every image directly to the same endpoint.

- Built or planned quality-gate and wrong-category-gate workflows because invalid images should be stopped before rule reasoning.
- Used structured review loops so rule-level outputs could be inspected, compared, and tuned instead of judged only from anecdotal examples.
- Treated evaluation as both model quality and pipeline quality: invalid image rejection, routing behavior, latency, and fallback correctness were all part of reliability.

4. Retail store compliance - production infrastructure and operational design

The retail service became a full production system, not just a model endpoint. I built and tuned the runtime around FastAPI, Redis, PostgreSQL, Nginx, service isolation, and an admin monitoring plane. The key design goal was to keep the analyze routes responsive under burst while protecting downstream LLM backends from collapse.

Deployed topology and routing

- Nginx fronted the public domain and routed traffic to separate upstreams: the main backend on :8000, the OCR2 / retail compliance app on :8001, and the standalone admin backend on :8100.
- Heavy analyze paths were isolated behind dedicated routing so compliance traffic could be tuned independently from the main application stack.
- The retail app exposed analyze routes, frontend stats routes, and internal admin LLM routes, while dashboard traffic normally went through a separate admin control plane.
- Uvicorn was deployed with four workers, uvloop, httptools, backlog tuning, and a dedicated systemd service for production lifecycle management.

Selected production details from the retail service

- Runtime command: uvicorn apps.api.main:app --host 127.0.0.1 --port 8001 --workers 4 --loop uvloop --http httptools --backlog 2048 --timeout-keep-alive 10
- Primary persisted tables: requests, analyses, llm_usages
- Admin plane isolated on port 8100 with its own FastAPI backend and React frontend
- Request lifecycle: claim request idempotently, run selected pipeline, persist LLM usage, then persist normalized analysis

Layered burst-control strategy I implemented

I did not rely on a single limiter. The service used multiple layers because production stress does not look the same at each stage. Request admission, backend choice, and backend inflight protection all needed to be controlled separately.

Layer	Mechanism	Purpose
Admission	Local token bucket in main.py	Per-client and per-process request shaping before the worker does expensive work.
Global control	Redis shared rate counters	Cross-worker limits so burst control is not bypassed simply by running multiple workers.
Routing	TrafficManager counters and logic	Gemini-first routing with split / spillover once thresholds are crossed.

Layer	Mechanism	Purpose
Inflight protection	Redis semaphore per backend	Hard cap on concurrent backend calls so LLM services do not collapse under admitted traffic.

- Traffic counters such as rpm and rps10 were stored in Redis and used not only for monitoring but also for backend routing decisions.
- Inflight queue control was implemented as a distributed semaphore with Lua scripts, TTL-based stale-slot protection, and polling until capacity opened or wait time expired.
- At the documented production snapshot, configured caps included GEMINI_MAX_INFLIGHT=120 and BEDROCK_MAX_INFLIGHT=80.

Backend distribution and failover logic

The default behavior was Gemini-first, but I added logic so the system could spill traffic to Bedrock when counters crossed thresholds or when the primary backend was unavailable. I considered routing decisions and actual backend invocation separately, because one request could start on one backend and finish on another after fallback.

Distribution behavior

- Gemini was the default route until minute / short-window counters crossed defined thresholds.
- For a 0.5 split ratio, deterministic parity was used instead of random sampling to stabilize high-load behavior.
- When one backend was disabled, the system routed to the other backend automatically.

Fallback behavior

- Gemini quota-like failures temporarily disabled Gemini for a short TTL window and sent traffic to Bedrock.
- Bedrock exception counts could trigger backend disable behavior once the daily threshold was reached.
- Queue wait timeout could trigger alternate-backend fallback; both disabled state returned 503 with retry guidance.

Operational evidence from logs and production behavior

I also analyzed runtime logs instead of judging the service only from endpoint availability. That showed how the routing plan behaved under real error and overload windows.

Metric	Observed behavior
Peak observed counters	827 RPM and 251 requests / 10s from runtime traffic counters.
Combined routing decision share	Gemini 95.77%, Bedrock 4.23% in the sampled logs.
Actual LLM call share	Gemini 60.76%, Bedrock 39.24%; evidence of

Metric	Observed behavior
	fallback / secondary execution under stress windows.
Timeout concentration	723 request-timeout markers, matching the main 504 concentration in the sampled logs.
LLM latency samples	Gemini average ~3798 ms and Bedrock average ~4155 ms across combined samples; some earlier Bedrock windows were much slower.

- One important observation from log analysis was that initial routing counts were heavily Gemini-biased, but actual Bedrock invocation was far higher because fallback logic was actively rescuing requests during stress windows.
- This difference between routing decisions and real backend calls became operational evidence that the failover design was not merely theoretical; it was being exercised in production-like conditions.

5. Admin dashboard, monitoring, and control plane

I separated the monitoring plane from inference workers so reporting traffic would not compete directly with analyze traffic. The standalone admin service had its own backend and frontend, and exposed authenticated views for cost, token, latency, throughput, and live routing counters.

Admin backend capabilities

- Login plus passkey / TOTP flow for protected operational access
- Historical usage summary by backend and model, plus recent-request inspection
- Throughput reports, live traffic counters, CSV export, machine / S3 dump utilities, and log operations

Why I separated this plane

- Monitoring and ops queries could be heavy; keeping them off the inference app protected analyze throughput
- Live Redis-derived counters and persisted Postgres aggregates served different operational needs
- The separation made the production stack easier to reason about during debugging and support

This monitoring layer was not just for convenience. It gave me the visibility required to answer practical questions: where the throughput bottleneck really was, whether live counters matched persisted history, whether a backend was failing over more often than expected, and how much token / cost consumption was being accumulated by model and backend.

6. Use case 2 - Invoice extraction

Another major use case was invoice extraction, where the objective was to turn invoice scans or PDFs into structured JSON that downstream systems could consume. This was not a one-shot OCR task. I approached it as a document-intelligence pipeline with pre-processing, OCR, layout handling, schema mapping, and validation.

My design approach for invoice extraction

- I separated image improvement, OCR, parsing, and schema mapping because business consumers needed stable field extraction rather than raw text blobs.
- I treated line items, GSTIN / HSN-style business fields, and totals as structured extraction problems that needed validation and not only generative explanation.
- I designed the output for downstream integration, so the result shape could plug into business systems rather than remaining a demo artifact.

Pipeline layers

- Input intake and file normalization for scans or PDFs
- Image enhancement and OCR / layout capture
- Schema mapping for header fields and line items
- Validation, JSON formation, and downstream handoff

What I learned here

- OCR quality and layout structure often dominate output quality more than the final reasoning step
- Structured field extraction needs explicit schemas and post-validation to be production-friendly
- A reliable invoice system is built from staged extraction, not from a single generic prompt

7. Use case 3 - Legal document extraction

The legal document workflow dealt with multi-page PDFs and scans, where the business goal was to produce a structured legal summary instead of plain OCR text. Compared to invoice extraction, the challenge here was longer context, multi-page continuity, and the need to extract important metadata along with narrative synthesis.

- I designed page-level OCR and segmentation so the system could treat large legal files more systematically instead of forcing everything into one unstructured step.
- The output had to combine extracted metadata and readable legal summary, which made schema design and output normalization important.
- This use case reinforced a common pattern across the Haier program: reliable outputs come from staged workflows where parsing and reasoning are coordinated, not collapsed together blindly.

9. Infrastructure, deployment, and integration

A major part of my contribution was infrastructure thinking. I did not treat the solution as a notebook-only AI project. I worked on how these use cases would be exposed, deployed, isolated, monitored, and integrated with client-facing systems.

Infrastructure principles I applied

- Separated upload / ingress responsibility from

Deployment and operational concerns I handled

- FastAPI service design, worker tuning,

heavy processing wherever possible
- Used object storage and asynchronous patterns for large media and retry-friendly design
- Isolated heavy routes and supporting services to reduce blast radius during load
- Persisted request and result data so the platform had auditability and operational memory

environment configuration, and reverse-proxy routing
- Redis-backed counters, state, queue control, and cache usage
- Service lifecycle management, log handling, and production support flows
- Integration patterns for downstream clients, dashboards, and callback / API consumption

This work mattered because production AI systems fail for many reasons other than model quality. Queue behavior, timeout control, invalid-input protection, and route isolation often decide whether the service remains usable under real demand. My approach was always to keep the platform not only accurate, but also operable.

10. Training, evaluation, and iteration

Across the Haier use cases, I treated training and evaluation as an engineering system rather than a one-time experiment. For image workflows, that meant thinking about category taxonomies, quality gates, wrong-category rejection, dataset readiness, and error review loops. For document workflows, it meant schema-aware validation, OCR quality analysis, and sample-based comparison across approaches.

- I considered both model quality and production quality. A pipeline that looks good in a few hand-picked examples but fails under real ingress conditions is not yet production-ready.
- I used iterative review loops: inspect outputs, find failure patterns, adjust the stage responsible, and re-measure rather than making blind global changes.
- I moved toward layered pipelines because some tasks benefit more from validation or support models than from pushing every problem into the most expensive reasoning model.

8. Key engineering decisions and my approach

Decision area	What I did	Why it mattered
Use-case architecture	Built one shared platform with reusable ingestion, storage, orchestration, persistence, and output patterns.	Reduced duplication and made new workflows easier to launch.
Retail optimization	Added cheap quality/category gates before expensive reasoning, then introduced traffic shaping and backend routing.	Lowered wasted LLM spend and stabilized production load.
Document intelligence	Separated OCR/layout extraction from reasoning and	Improved controllability, debuggability, and field-level

Decision area	What I did	Why it mattered
	schema mapping instead of treating everything as one generic prompt.	validation.
Infrastructure	Isolated heavy routes, tuned workers, used Redis for queue/control state, and separated the admin plane from inference workers.	Protected the main system under burst and made operations easier to observe.
Evaluation	Built rule-level review loops, sample-based checks, and production log analysis instead of relying only on demo success.	Allowed me to improve weak points with evidence.

11. Deliverables I contributed to

- End-to-end architecture thinking for multiple Haier AI use cases
- Retail compliance pipeline design, invalid-image handling, backend routing, and production traffic control
- Invoice extraction and legal document extraction workflow design with structured outputs
- FastAPI services, persistence flows, worker orchestration patterns, and admin / monitoring APIs
- Infrastructure setup decisions covering routing, service isolation, logging, caching, and deployment operations
- Evaluation support loops connecting model behavior with system reliability and downstream usability

12. Portfolio framing and closing note

This project is a strong portfolio example because it combines the parts of AI work that matter in real delivery: use-case decomposition, pipeline design, model support, production engineering, monitoring, and iterative improvement. My role was not just to build a prototype that could work on a few examples; it was to shape a system that could be deployed, observed, and improved over time.

In summary

- I designed and implemented a multi-use-case Haier AI platform covering retail compliance, invoice extraction, and legal document extraction.
- I worked across service architecture, async orchestration, training / evaluation support, infrastructure setup, and operational monitoring.
- The retail compliance use case especially demonstrates my production approach: invalid-input defense, queue control, routing, failover, persistence, and observability around expensive AI inference.