SMARTFOUNDRY · WHITEPAPER

Enterprise AI Architecture

A practitioner's guide to building production-grade AI platforms

~10 MIN READ | FOR A TECHNICAL AUDIENCE | SMARTFOUNDRY.IO

Abstract. Most enterprise AI programs stall in the gap between a promising prototype and a system people can depend on. This paper treats that gap as an architecture problem. It walks through a reference platform for large language model (LLM) applications, the retrieval and data layers that ground them, the operational lifecycle that keeps them healthy, and the security, cost, and observability concerns that decide whether a deployment survives contact with real users. It is written for engineers and architects who already know how to ship software and now need to ship AI.

01 — WHY THIS MATTERS

Executive Perspective

Enterprises are investing heavily in AI, yet a large share of initiatives never leave the demo stage. The reason is rarely the model. A capable model wired into a fragile system produces a fragile product. The discipline that separates a science project from a platform is the same discipline that separates a script from software: clear interfaces, explicit state, testability, observability, and an operational lifecycle.

SmartFoundry treats AI as a systems-engineering challenge. The model is one component among many: retrieval, orchestration, evaluation, identity, audit, and cost control all sit around it. When those components are designed deliberately, the same platform can absorb a model swap, a new vendor, or a tenfold increase in traffic without a rewrite.

> **Key takeaway —** Architecture, not model selection, is the durable differentiator. Models change every few months; a well-factored platform outlives any single one of them.

A useful test of production-readiness is to ask what happens on a bad day. What is the latency budget at the 95th percentile, and what does the system do when the model is slow or unavailable? How are partial failures, timeouts, and rate limits handled? Can a regression be detected and rolled back in minutes? Prototypes answer none of these; platforms answer all of them, and that difference is architectural rather than model-deep.

02 — THE PROBLEM

Market Challenge

Three failure modes recur across organizations. The first is fragmentation: every team builds its own prompt-and-API glue, so there is no shared retrieval layer, no shared evaluation, and no shared guardrails. The second is the absence of governance, which means no one can answer who can call which model, on what data, with what controls. The third is unclear return on investment, because without evaluation and cost telemetry there is no way to prove a feature is accurate enough or cheap enough to keep running.

These are organizational symptoms of a missing platform. The fix is not a bigger model or a new framework; it is a small set of shared, well-owned services that every AI feature is built on top of.

- No shared retrieval or memory layer leads to duplicated, inconsistent answers.
- No evaluation harness means quality regressions ship silently.
- No identity or audit story blocks adoption in regulated environments.
- No cost attribution makes it impossible to reason about unit economics.

Buying a packaged assistant does not remove the platform gap; it relocates it. The moment a vendor tool must read private data, respect permissions, be evaluated, and be billed back to teams, the same retrieval, governance, and observability concerns reappear. The organizations that pull ahead assign clear ownership of a shared AI platform rather than letting each team solve these problems in isolation.

03 — THE BLUEPRINT

Reference Architecture

A production LLM application is best understood as a request pipeline with clear stages. A request arrives at an API gateway, is authenticated and rate-limited, then passes to an orchestration layer that assembles context, calls the model, validates output, and returns a response. Retrieval, tools, caches, and guardrails are services this orchestrator calls; they are not buried inside prompt strings.

```
client -> API gateway (authn, rate limit, quotas)
        -> orchestrator
             |- retrieval service (embeddings + vector store)
             |- tool / function calls (typed, authorized)
             |- prompt assembly + model router (Bedrock / vLLM)
             |- output guardrails + schema validation
        -> response + trace (logged to observability)
```

On AWS this maps cleanly to managed services: API Gateway and Lambda or ECS Fargate for the orchestrator, Amazon Bedrock for managed model access, OpenSearch or a managed vector database for retrieval, S3 for source documents, and Secrets Manager for credentials. The specific services matter less than the seams between them. Keep the model behind a router so you can route by cost, latency, or capability, and so a vendor change is a configuration change rather than a code change.

> **Key takeaway —** Design for substitution. Every external dependency (model, vector store, embedder) should sit behind an interface you own, so swapping it never touches business logic.

Keep the orchestrator stateless so it scales horizontally and fails over cleanly; push session and conversation state into a dedicated store. Make requests idempotent with a client request id so retries do not double-charge or double-act, and support streaming responses end to end so the gateway, orchestrator, and client all handle partial output. These properties are far cheaper to design in at the start than to retrofit under load.

04 — GROUNDING THE MODEL

RAG and Enterprise Data

Retrieval-Augmented Generation (RAG) connects a general model to your private knowledge so answers are grounded in current, authoritative data instead of the model's frozen training set. The pipeline has two halves: an offline indexing path that turns documents into searchable vectors, and an online path that retrieves the most relevant chunks at query time and places them in the prompt.

Quality is dominated by unglamorous choices: how you chunk documents, what you embed, and how you rank results. Oversized chunks dilute relevance; tiny chunks lose context. Hybrid search that combines vector similarity with keyword (BM25) scoring, followed by a re-ranking step, consistently beats pure vector search on enterprise corpora.

```
# Offline indexing
for doc in corpus:
    chunks = split(doc, size=800, overlap=120)   # tokens
    vectors = embed(chunks)                       # e.g. Titan / e5
    store.upsert(vectors, metadata={'source': doc.uri, 'acl': doc.acl})

# Online query
hits = store.hybrid_search(embed(q), bm25(q), k=20, filter=user.acl)
context = rerank(q, hits)[:5]
answer = llm(prompt(system, context, q))
```

Two rules keep RAG trustworthy. Carry access-control metadata on every chunk and filter retrieval by the caller's permissions, so the model can never surface a document the user could not otherwise read. And always cite sources in the response, which lets users verify answers and gives you a signal for evaluation.

Treat retrieval as something you measure, not assume. Build a small labeled set of questions with known relevant documents and track recall@k and precision so you can compare chunking and embedding choices objectively. Keep the index fresh with incremental, event-driven re-indexing as source documents change, and prefer semantic or structure-aware chunking over naive fixed windows when documents have clear sections.

> **Key takeaway —** If answers are wrong, suspect retrieval before the model. In most enterprise RAG failures the model never received the right context in the first place.

05 — BEYOND CHAT

Agentic Systems

Agents extend a model from answering questions to taking actions: querying a database, opening a ticket, calling an internal API. An agent loops between reasoning and acting, choosing tools until a goal is met. This unlocks real automation, and it also widens the blast radius, so the engineering bar is higher than for a chatbot.

- Expose tools as typed, individually authorized functions, never as raw shell or SQL.
- Scope each tool to least privilege and log every invocation with its arguments.
- Bound the loop: cap iterations, set timeouts, and require confirmation for risky writes.
- Treat tool output as untrusted input that can carry prompt-injection payloads.

The most common production failure is an over-powerful tool. A 'run query' tool with broad database rights is a liability; a set of narrow, parameterized tools ('get_order', 'refund_order') is auditable and safe. Design the tool surface first, then let the model compose those tools.

Give agents only the memory they need and make planning explicit. A reason-then-act loop with a written scratchpad is easier to debug than an opaque chain, and a human approval step before irreversible actions (payments, deletions, external messages) is a cheap, powerful safeguard. Persist the full action trace so any agent run can be replayed and audited later.

06 — OPERATING THE MODEL

LLMOps Lifecycle

Prompts, retrieval settings, and model versions are production artifacts and deserve the same rigor as code. LLMOps is the practice of versioning those artifacts, testing changes before release, deploying them progressively, and monitoring behavior continuously.

- Version prompts and configs in source control; never edit them live in production.
- Gate every change behind an evaluation suite run in CI (see Section 09).
- Roll out model or prompt changes with canaries and a fast rollback path.
- Capture real traffic (with consent) to grow regression and red-team datasets.

> **Key takeaway —** A prompt change is a deploy. If it can alter user-facing behavior, it goes through review, evaluation, and staged rollout like any other production change.

Promote prompts and models the way you promote code. A prompt registry with versioning lets you A/B two variants on live traffic and compare them on quality and cost, while shadow deployments run a new model on real requests without showing its output to users, surfacing regressions before they ship. Tie every released version back to the evaluation run that approved it.

07 — UNIT ECONOMICS

Cost and Performance

LLM features have a per-request marginal cost that traditional software does not, so cost is a first-class design constraint. The largest levers are token volume and model choice. Trimming retrieved context, capping output length, and routing easy requests to smaller models often cut spend

by more than half with no quality loss.

- Cache aggressively: identical or semantically similar prompts can reuse responses.
- Route by difficulty so a small model handles routine work and a large model handles the rest.
- Stream responses to improve perceived latency without changing total cost.
- Attribute cost per feature and per tenant so you can reason about ROI.

Latency and cost trade against each other. Measure both at the percentile that matters to users (p95, not the mean) and set explicit budgets so the platform fails predictably under load rather than silently degrading.

Caching has two tiers worth building: exact-match caching for identical prompts and semantic caching that reuses answers for sufficiently similar questions. Context compression, summarizing or pruning retrieved text before it reaches the model, cuts input tokens directly. For steady, high-volume workloads, provisioned throughput is usually cheaper and more predictable than on-demand pricing; reserve on-demand for spiky traffic.

08 — TRUST AND SAFETY

Security and Governance

AI introduces attack surfaces that classic application security does not cover. Prompt injection, where untrusted content instructs the model to ignore its rules, is the signature risk; data leakage through over-broad retrieval is a close second. Defense is layered, not a single switch.

- Authenticate and authorize every model and tool call with least privilege.
- Filter retrieval by the caller's permissions so the model cannot exceed the user.
- Validate and constrain output (schemas, allow-lists) before it triggers any action.
- Run input and output guardrails for prompt injection, PII, and policy violations.
- Encrypt data in transit and at rest, and retain audit logs for every interaction.

```
guardrail:
  input:  [prompt_injection, pii_detect, topic_denylist]
  output: [pii_redact, toxicity, schema_validate]
  on_violation: block_and_log
```

> **Key takeaway —** Map controls to a recognized framework such as the NIST AI Risk Management Framework early. Retrofitting governance after launch is far more expensive than designing it in.

Defending against prompt injection means never trusting tool or document content as if it were instructions. A practical pattern separates a privileged planner model, which never sees raw untrusted

text, from an unprivileged worker that processes content but cannot call sensitive tools. Combine this with strict per-tenant data isolation so one customer's data and prompts can never reach another's context.

09 — KNOWING IT WORKS

Observability and Evaluation

You cannot operate what you cannot see. Every request should emit a trace that records the prompt, retrieved context, model and version, token counts, latency, cost, and the final output. Traces make incidents debuggable and feed the datasets that drive evaluation.

Evaluation is what makes AI changes safe to ship. Maintain a labeled test set of representative inputs and run it automatically on every change, scoring for correctness, groundedness (is the answer supported by retrieved sources), and safety. Combine deterministic checks with model-graded scoring, and track results over time so regressions are caught before users see them.

```
for case in eval_set:
    out = pipeline(case.input)
    assert grounded(out, case.sources)     # no unsupported claims
    score = judge(out, case.reference)      # model-graded rubric
    record(case.id, score, out.cost, out.latency)
gate(mean_score >= threshold)               # fail CI if regressed
```

Offline evaluation catches known problems; online evaluation catches the unknown ones. Capture implicit signals (thumbs, edits, abandonment) and sample real traffic for human or model-graded review, then watch for drift as inputs and the world change. Standardize traces on OpenTelemetry so AI telemetry lives alongside the rest of your observability rather than in a silo.

10 — IN SUMMARY

Conclusion

Enterprise AI succeeds when it is engineered like a platform: substitutable components, grounded retrieval, a disciplined operational lifecycle, explicit cost budgets, layered security, and relentless evaluation. The model is the easy part. The platform around it is what turns a demo into a dependable product.

> **Key takeaway —** Start small but build the seams right: one shared retrieval service, one evaluation harness, one guardrail layer, one cost dashboard. Everything else compounds from there.

A pragmatic sequence: stand up one retrieval service and one evaluation harness first, add guardrails and tracing next, then a model router and cost dashboard. Ship one real feature on this foundation,

learn from production, and only then generalize the platform for the next team.