Production AI Engineering

Joona-Pekka Kokko · jokokko.com · github.com/jokokko

TL;DR — top 5 if you read nothing else

- **(1) Build an eval set before anything else.** No eval set, no way to tell if a change is an improvement. Every other recommendation here is downstream of this.
- **(2) Use prompt caching.** ~10× cheaper input tokens for stable prefixes. Anthropic exposes `cache_control`; OpenAI auto-caches. Put system prompt, tool schemas, and large reused docs at the front.
- **(3) Hybrid retrieval, not pure vector.** Vector search misses error codes, identifiers, version numbers. Combine with BM25 via RRF, then rerank with a cross-encoder. Long context windows do not let you skip this.
- **(4) Structured outputs via native APIs, not "please return JSON".** `response_format=json_schema`, tool-use schemas, or constrained decoding. Validate with Pydantic/Zod and repair on failure.
- **(5) For agents: budget caps, sandboxed code, HITL in trusted UI.** Loop budgets stop runaway spend. Sandbox anything that runs code. Destructive actions get human confirmation in the host UI, not a model-generated "are you sure?" — that's suppressible by injection.

---

1. Foundations

1.1 LLMs vs classical services

Classical software is deterministic. LLMs are not. The contrast:

Dimension	Classical service	LLM-backed service
Determinism	Reproducible	Stochastic; not even reproducible at temperature 0
State	App-managed	Stateless model — "memory" is whatever you put in context
Failure modes	Exceptions, timeouts	Plus: hallucination, instruction drift, format violations, prompt injection, context overflow
Latency	Sub-100ms	500ms–60s+ depending on output and reasoning
Cost	CPU-time, fixed infra	Token-metered, user-driven, hard to forecast
Testing	Unit tests	Eval sets, judges, statistical thresholds
Versioning	Code	Code + prompts + model + tool schemas + retrieval index

The LLM is an unreliable remote service with a natural-language API and a hard token budget.

1.2 The request cycle

LLM APIs take a structured array of messages with roles (`system`, `user`, `assistant`, `tool`), not a string. The model is stateless — every turn must contain whatever history matters.

Context budgeting. Total tokens = system prompt + retrieved context + tool schemas + history + tool outputs + completion. You'll exceed limits faster than expected once tools and retrieval kick in. Measure tokens; characters lie (especially for code, non-English, and unusual symbols).

History management. For long sessions:

- *Sliding window* — keep the last N turns. Cheap. Loses long-range facts.
- *Summarization* — periodically compress older turns into a system-side summary. Watch for stale or precision-lossy summaries.
- *Importance-weighted retention* — score turns and pin high-value ones (decisions, identifiers, corrections). Better fidelity, more code.
- *External memory + retrieval* — embed past turns into a vector store, retrieve relevant ones each turn. Closest to real long-term memory; adds retrieval failure modes.

Prompt caching is the highest-ROI optimization most teams skip. Anthropic gives you explicit cache breakpoints (`cache_control`); OpenAI auto-caches stable prefixes. Cached input tokens are typically ~10× cheaper and faster. Order matters: only content before the first variable input gets cached. Structure prompts so the system prompt, tool schemas, and large reused documents sit at the front.

1.3 Picking a model

Selection criterion: cheapest model that hits the eval bar.

- **Capability tier.** Frontier models for novel reasoning, multi-step planning, code generation. Small/fast models for classification, extraction, routing, simple rewriting. Tier gap is huge on hard tasks, mostly noise on easy ones.
- **Reasoning models vs. standard.** Reasoning models (extended thinking, o-series) trade latency and tokens for accuracy on multi-step problems. Not a free upgrade — slower and more expensive on short factual or stylistic tasks, sometimes worse.
- **Context window.** Quoted limits are best-case. Models degrade well before the stated max. Test with real inputs.
- **Tool-use fidelity.** Some models follow tool schemas reliably; others hallucinate parameter names. Verify on your tools, not benchmarks.
- **Latency.** Time-to-first-token, throughput, and total time-to-completion are different numbers. Pick the one your UX cares about.
- **Cost per 1M tokens** for input and output, weighted by your typical input/output ratio.

Routing. Once you have multiple task types, route. A small classifier (or a cheap LLM call) sends simple queries to a cheap model and complex ones to the expensive one. Add a fallback chain: on schema-validation failure, transient error, or refusal, retry with a stronger model before surfacing the failure to the user.

Prompts are code. Prompts, model IDs, sampling params, tool schemas, and output schemas are all behavior-affecting config. Version them with the application, attach to releases, gate changes through CI with eval runs. Unversioned prompt drift in prod is one of the most common "it used to work" root causes.

1.4 Transformer mechanics

- **Tokenization** splits text into sub-word units. Token counts diverge from word counts; billing and context limits are token-denominated.
- **Embeddings** map tokens to high-dim vectors. Same machinery powers retrieval, similarity caching, dedup.
- **Self-attention** weighs each token against every other token via query–key dot products through softmax. Quadratic in sequence length — long context is genuinely expensive, not just "more tokens".

The attention computation:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(Q \cdot K^{T} / \sqrt{d_k} \right) \cdot V$$

- **Q (query):** what this position is looking for.
- **K (key):** what each position offers.
- **V (value):** what each position contributes if attended to.
- $\sqrt{d_k}$ scales the dot products. Without it, in high dimensions the dot products grow large, softmax saturates (one weight near 1, the rest near 0), gradients collapse, the model can't learn nuanced multi-token attention. Square-root is right because dot products of independent unit-variance vectors have variance proportional to d_k.

What matters in practice:

Component	Engineering implication
Tokenizer	Drives token cost and context utilization. Budget in tokens.
Embeddings	Powers retrieval, similarity caching, dedup, clustering.
Self-attention	Quadratic — long context is expensive.
Positional encoding	Order matters; the model knows where tokens sit.
Softmax + temperature	Your only real lever on output randomness.
KV cache	Reused prefixes save real money. Design for cache-friendliness.

1.5 Controlling randomness

Logits → softmax → token. Temperature `T` rescales as `logits/T` before softmax:

- **T → 0:** greedy decoding. Most repeatable. Use for extraction, classification, structured outputs, code where exact form matters.
- **T = 0.7–1.0:** balanced. Fine for most chat and content generation.
- **T > 1:** flatter distribution, more variation. Use sparingly.

`temperature=0` **is not deterministic.** Floating-point non-associativity in batched matmul means you'll see occasional differences across model versions, hardware, batching, even time. If you need bit-exact repeatability, you don't have it. Design around statistical stability, not equality.

`top_p` **(nucleus sampling)** truncates the distribution to the smallest set of tokens whose probability mass exceeds `p`. Often a better lever than temperature alone. Don't aggressively tune both at once.

---

2. Context engineering and RAG

2.1 Prompt engineering principles

Principles that consistently improve outputs:

Use the system prompt for stable behavior. Persona, format constraints, refusal policy, style — anything that doesn't change per-request goes here. It's also the cleanest cache target.

Delimit cleanly. XML tags (`<context>...</context>`, `<question>...</question>`) work especially well on Anthropic models. Markdown headers also work. Point is unambiguous boundaries between instructions, user input, and retrieved data — both for the model and for prompt-injection defense.

Position matters, and "lost in the middle" is real. Models attend best to the start and end of long contexts, worst to the middle. So:

- Critical instructions: repeat at both start and end for long contexts.
- Most-relevant retrieved chunks: beginning or end of the retrieved block, not buried in the middle.
- Don't assume a fact in the middle of a 100k-token context will be reliably retrieved by attention.

Few-shot examples. Two to five well-chosen examples typically beat long instructions. Demonstrate edge cases and the exact output format. Dynamic few-shot — retrieve the most similar examples from a curated library at request time — beats static examples on heterogeneous workloads.

Chain-of-thought for non-reasoning models. Asking for step-by-step thinking before the answer measurably helps on multi-step problems. For reasoning models, this is built in — adding "think step by step" is redundant and sometimes counterproductive.

Force citations. When grounding on retrieved context, require inline citation of the source chunk for each claim. Hallucinations drop sharply because the model commits to evidence at generation time instead of confabulating around it.

Heuristic framing is unreliable. "This is mission-critical", "lives depend on this" — inconsistent across models, often counterproductive on modern frontier ones. Structural clarity beats emotional pressure.

2.2 RAG: ingestion

RAG grounds outputs in your data. Three stages:

Normalization. Convert source formats (PDF, HTML, DOCX, code, transcripts) to clean text with structure preserved. PDF extraction has more failure modes than people expect — flattened tables, interleaved columns, lost headers — and it silently destroys retrieval downstream. Inspect your normalized output before you ever embed.

Chunking. Strategies, in increasing sophistication:

- *Fixed-size with overlap* (512–1024 tokens, 10–15% overlap) — cheap, decent baseline.
- *Recursive splitting* on structural separators (headers → paragraphs → sentences) — respects document structure.
- *Semantic chunking* — split where embedding similarity drops between adjacent sentences. Better topic coherence, more compute.
- *Hierarchical / parent-child* — embed small focused chunks for retrieval precision, return larger surrounding context for generation. Solid default for technical docs.

Chunk enrichment. Prepend each chunk with metadata breadcrumbs: doc title, section path, date, author, source URL. A 200-token chunk with "Title: X / Section: Y" prefixed retrieves dramatically better. Anthropic's "contextual retrieval" technique generates a per-chunk situating sentence with a cheap LLM call before embedding — measurable wins on hard queries.

Embedding. Pick a model that matches your domain (general English, multilingual, code). Test on your queries. Write to an ANN index (FAISS, pgvector, Qdrant, Pinecone) with metadata stored alongside.

2.3 Retrieval

Hybrid search is the default, not an optimization. Pure vector search misses exact-match cases: error codes, identifiers, proper nouns, version numbers, code symbols. BM25 catches these. Combine via Reciprocal Rank Fusion (RRF). Production RAG systems converge on hybrid because the failure modes are complementary.

Metadata pre-filtering. Hard-filter the candidate set before semantic search: `tenant_id`, `date >= ...`, `department = ...`, `language = ...`. Treats the vector index as search-within-results, not a global semantic database. Skipping this is a

common cause of cross-tenant leaks and stale results.

Query rewriting. A cheap LLM call rewrites vague or conversational queries into focused search queries. Especially important in multi-turn dialogue where pronouns and context shift the actual information need.

HNSW tuning (the dominant ANN structure):

- `M` (max connections per node): higher = better recall, more memory.
- `ef_construction`: higher = better graph quality, slower build.
- `ef_search`: higher = better recall, higher latency. Tune per query, not globally.

Measure recall against an exhaustive baseline on a sample. Fast indexes that miss 30% of relevant docs are a silent failure mode.

2.4 Reranking

Retrieval is high-recall, low-precision by design. Rerank to fix that.

Cross-encoder rerankers (Cohere Rerank, BGE reranker) score query+candidate as a single input. Slower than bi-encoders, materially more accurate. Standard pattern: hybrid retrieve top-50 to top-100, rerank to top-5 to top-10.

A 200k-token window does not eliminate reranking. Two reasons:

- Cost and latency scale with input tokens.
- "Lost in the middle" — a fact buried among 80 chunks is less reliably attended to than the same fact in a focused 5-chunk context.

Measure faithfulness and answer quality, not document presence in the candidate set.

2.5 Long-context vs. RAG

- **Long-context wins** for small corpora that fit (a single contract, repo, paper), cross-cutting relationships, bounded recency.
- **RAG wins** for large corpora, frequent updates, tenant isolation, or when cost-per-query matters at scale.
- **Hybrid wins** when a small "always-on" core context coexists with a large retrievable corpus — agent loops over a codebase, for instance.

Prompt caching shifts the math: caching a 100k-token document prefix makes long-context far cheaper to reuse across turns. Re-evaluate annually.

2.6 Retrieval evaluation

Metric	What it measures
Recall@k	Fraction of relevant docs in top-k
MRR	Mean reciprocal rank of the first relevant doc
nDCG	Position-weighted ranking quality
Faithfulness	Whether the generated answer is supported by retrieved context
Answer relevance	Whether the answer addresses the actual question

Evaluate retrieval **in isolation** with a curated query → relevant-docs gold set before evaluating end-to-end. Otherwise the LLM's own knowledge masks retrieval failures and you'll fix the wrong thing. Tools: Ragas, DeepEval, TruLens, or a few hundred lines of your own.

Maintain the gold set under version control. Add new examples whenever you find a regression. Run retrieval evals on every index change, embedding-model upgrade, and chunking-strategy change.

2.7 Semantic caching

Two-tier:

- **(1) Exact-match.** Hash (canonicalized prompt + model + params) → response. Trivial, free, high hit rate for repeated queries.
- **(2) Semantic.** Embed the incoming query, look up the nearest cached one; if cosine similarity > threshold, return the cached answer.

The semantic cache is dangerous if the threshold is loose. "What is X" and "What is *not* X" can sit at similarity 0.95. Treat the threshold as task-specific, evaluate on real traffic, and only cache stable factoid-style answers — not anything that depends on context, time, or user state.

---

3. Agents

3.1 Agent loop

A control loop wrapping an LLM with tools. Minimum loop:

- **(1)** Send the model the goal, available tools, and conversation/scratchpad so far.
- **(2)** Model emits either a final answer or a tool call.
- **(3)** Execute the tool, capture the result, append to context.
- **(4)** Loop until the model emits a final answer or a stop condition fires.

This is ReAct. Use the model's native tool-calling API, not prompt-based parsing — eliminates a class of formatting failures.

3.2 The Model Context Protocol

MCP is Anthropic's open standard for connecting models to tools and data without per-integration glue code. JSON-RPC 2.0 over stdio, SSE, or streamable HTTP.

Roles:

- **Host** — the application embedding the model (Claude Desktop, IDE, your app).
- **Client** — the per-server connection in the host that handles the protocol.
- **Server** — the process exposing tools (callable functions), resources (data the host can read), and prompts (templated workflows).

Transports:

- **stdio** — server is a subprocess of the host. Simplest, most secure, local only.
- **SSE / streamable HTTP** — networked. Required for remote services. Auth is your problem — typically OAuth 2.1, sometimes bearer tokens.

Capability negotiation happens at session init. Don't assume capabilities; check.

Security. MCP gives the model access to tools that may have side effects:

- Servers should run with least privilege. A filesystem MCP server pointed at `/` is a footgun.
- Untrusted MCP servers can attempt prompt injection via tool descriptions or returned content. Treat tool output as untrusted input.
- For destructive actions, require explicit user confirmation in the host UI — not an LLM-generated "are you sure?", which is suppressible by injection.

3.3 Workflow patterns

Patterns combine; production systems mix them.

Plan-then-execute. Force the model to write a plan (numbered steps, expected tools per step) before any tool call. Reduces thrashing on complex tasks, produces an auditable artifact. Costs an extra LLM call.

ReAct loop. Open-ended thought–action–observation. Best for exploratory tasks where the path can't be planned upfront (research, debugging, data exploration).

LLM pipeline / assembly line. Sequential, fixed stages, each with a narrow specialized prompt. Most reliable for known business workflows. Easier to evaluate per-stage. Less flexible.

Agentic RAG / multi-hop retrieval. Agent searches, reads, identifies gaps, searches again. Strong for research-style queries that don't decompose into a single retrieval.

Parallel sub-agents. Independent calls executed concurrently for genuinely independent subtasks (summarize each of 10 documents, then aggregate). Latency win; doesn't help if the subtasks have dependencies.

Orchestrator + specialist workers. A controller agent delegates to specialists, each with a narrow toolset and dedicated system prompt. Reduces per-agent cognitive load and tool confusion at the cost of additional handoff points where context can be lost. Indicated above the complexity threshold where single-agent tool selection accuracy degrades.

3.4 Structured outputs

When tool results or final answers must conform to a schema, prompting alone is insufficient.

- **Native structured output APIs** (OpenAI's `response_format=json_schema`, Anthropic's tool-use schemas, Gemini's controlled generation) constrain decoding to valid JSON matching your schema. Use these instead of post-hoc parsing.
- **Constrained / grammar-based decoding** (Outlines, llama.cpp grammars, vLLM guided decoding) enforces schema

or regex during token sampling on open models. Stronger guarantee — the model literally can't emit invalid output — at some quality cost if the schema is over-specified.
- **Validate, then repair.** Even with native structured outputs, validate with Pydantic / Zod / JSON Schema before downstream use. On failure, re-prompt with the validation error rather than crashing.
- **Schema design matters.** Deeply nested or ambiguous schemas degrade quality. Prefer flat, well-named fields with clear `description` strings — the model sees those and treats them as inline docs.

3.5 Tool design

The tool schema is a prompt. The model picks tools and fills parameters based on names, descriptions, and parameter docs.

- **Name tools by intent**, not implementation: `find_customer_by_email`, not `db_query_v2`.
- **Description states when to use this tool and when not to.** Negative guidance ("don't use this for X — use Y instead") prevents misrouting.
- **Parameter descriptions are a contract.** Include format, units, examples. `{"type": "string", "description": "ISO 8601 date, e.g. 2026-04-27"}` outperforms `{"type": "string"}` consistently.
- **Keep the toolset small per agent.** Tool selection accuracy degrades past roughly a dozen tools. Beyond that, route between specialist agents with smaller toolsets.

3.6 Resilience

Agents fail in ways ordinary services don't.

- **Retry with backoff** on transient errors (rate limits, timeouts, 5xx). Exponential with jitter.
- **Structured errors back to the model.** When a tool fails, return `{"error": "...", "hint": "..."}` rather than raising — the model gets a chance to self-correct. Don't expose stack traces.
- **Loop budgets.** Hard caps on iterations, total tokens, total wall-clock, total cost. Without these, an agent in a degenerate loop burns budget for the duration of the cap. Surface cost telemetry per request.
- **Output sanitization.** Strip raw HTML, control characters, oversized payloads, credential-shaped strings before feeding tool output back to the model. Tool results are semantically untrusted user input from the model's perspective.
- **Trajectory evaluation.** Final-answer correctness alone isn't enough. Evaluate the trajectory: tool-call correctness, redundant calls, error recovery, plan adherence. Needs a dataset of recorded trajectories with annotations, not just final-answer comparison.

3.7 Side effects

Any agent that writes — DB, side-effecting API, email, deploys — needs hard guardrails.

- **Read before write.** Inspect current state before any mutation.
- **Atomic transactions** for multi-step writes. Half-completed agent actions are an operational nightmare.
- **Idempotency keys** on external calls so retries don't double-charge or double-send.
- **HITL confirmations** for destructive or financial actions. **In the host's trusted UI**, not a model-generated confirmation question — that's suppressible by injection.
- **Sandboxed code execution.** Anything that runs code goes in an isolated container with no network egress to internal services and a read-only filesystem outside its workspace. Non-negotiable.
- **Audit logs.** Every tool call with arguments, results, and reasoning trace. Required for incident investigation and post-hoc behavioral analysis.

---

4. Evaluation

4.1 The evaluation loop

A team without an eval set is shipping vibes. Every prompt change, model upgrade, retrieval tweak, and tool-schema edit is a regression risk. Without a measurement harness, there is no way to tell whether a change is an improvement.

The cycle:

- **(1) Analyze.** Read traces from real (or simulated) traffic. Identify failure patterns qualitatively.
- **(2) Measure.** Build automated evaluators that score those patterns at scale.
- **(3) Improve.** Change one variable. Re-run. Compare.
- **(4) Monitor.** Run a subset continuously in prod; alert on drift.

4.2 Error analysis

Treat it as qualitative coding, not bug triage.

- **Open coding.** Read 50–100 real traces. Tag each failure with a short descriptive label ("hallucinated price", "missed pre-filter", "bad tool selection", "format violation"). Don't normalize yet.
- **Axial coding.** Group tags into a taxonomy: retrieval, reasoning, format, policy. The taxonomy tells you what to measure.
- **Saturation.** Stop when new traces stop producing new categories. Usually earlier than expected — often within 50 traces for a focused workflow.

The taxonomy drives the next step: each category becomes either an automated check, a reranker tweak, a prompt revision, or a tool-schema fix.

4.3 LLM-as-judge

Human review doesn't scale past a few hundred examples. Automated judges fill the gap.

- **Calibrate against humans.** SMEs label a gold set. Measure judge agreement vs. SMEs as TPR/TNR or Cohen's kappa. A judge that doesn't agree with humans is a random number generator with extra steps.
- **Pairwise > pointwise.** "Which of A and B is better" is more reliable than "score A from 1–5". Position bias is real — randomize order or evaluate both orderings and average.
- **Multi-judge ensembles.** Two or three different judge models, majority vote or score average. Reduces single-model bias. Especially important when the judge is the same family as the system being evaluated (you'll get inflated scores).
- **Split your gold set** into a portion for designing/tuning the judge and a held-out portion only used for measuring final judge quality. Specific ratios (20/40/40, 60/20/20) are fine — what matters is that the test split is never used in iteration.
- **Judge prompts also drift.** Version them. Re-validate against the gold set when you change them.

4.4 Human evaluation

Human evaluation is the ground-truth signal. Requirements for it to produce usable labels:

- **Annotation guide.** Written, versioned, with edge cases. Without it, two annotators produce different distributions.
- **Inter-annotator agreement.** Cohen's kappa or Krippendorff's alpha. Below ~0.6 your guide is too vague or the task is too subjective; above ~0.8 you can trust the labels.
- **Active sampling.** Don't label uniformly. Oversample edge cases, model disagreements, and recently-failed traces.

4.5 Synthetic data and adversarial testing

- **Synthetic test cases.** Generate (channel × intent × persona) tuples to expand coverage of underrepresented combinations. Useful for pre-launch coverage when production data is scarce.
- **Perturbations.** Inject typos, irrelevant content, conflicting instructions, code-switching. Tests robustness to messy real input.
- **Red-teaming.** Adversarial prompts targeting prompt injection, jailbreaks, PII extraction, harmful content, tool misuse. Run before launch and continuously after. Public datasets exist (BBQ, WinoBias for bias; HarmBench, AdvBench for safety) but the highest-value red-teaming targets *your* system's specific tools and data.
- **Bias and fairness testing.** Demographic parity, stereotype probes, refusal-rate symmetry across protected attributes. Set thresholds; block deployment on regression.

---

5. Production

5.1 Quantization and serving

For self-hosted open models, quantization (FP16 → INT8/INT4) cuts memory ~2–4× and latency ~1.5–3×.

- **GPTQ** — post-training, weight-only. Fast inference, mature tooling.
- **AWQ** — activation-aware. Protects highest-impact weights based on activation magnitudes. Generally better quality preservation than GPTQ at the same bit width.
- **GGUF + llama.cpp quantizations (Q4_K_M, Q5_K_M)** — mixed precision per layer. Default for CPU / Apple Silicon / heterogeneous inference.
- **FP8 / NVFP4** — hardware-accelerated low-precision on H100/Blackwell. Often near-lossless.

Always evaluate the quantized model on your task. INT4 can degrade arithmetic, code, and long-context reasoning more than benchmarks suggest. Right bit width is the lowest one that holds your eval scores.

5.2 Fine-tuning

Default to RAG and prompt engineering. Fine-tune only when:

- You need consistent style or format that prompting doesn't reliably produce.
- You're encoding a stable, repeatedly-invoked skill or domain vocabulary (worth amortizing).
- You have hundreds-to-thousands of high-quality examples (not dozens).
- You can absorb the cost of periodic retraining as the base model and your data evolve.

PEFT methods:

- **LoRA** — trainable low-rank adapters; base frozen. Cheap to train, cheap to serve (multiple LoRAs per base).
- **QLoRA** — LoRA on a 4-bit-quantized base. Lets you fine-tune large models on a single consumer GPU.
- **DPO / KTO / ORPO** — preference-based fine-tuning from pairwise comparisons. Often more sample-efficient than SFT for alignment-style tasks.

Distillation — train a small model on a strong model's outputs — is the standard approach for high-volume narrow tasks. Inference cost drops materially once quality is validated on a held-out set.

Default position: do not fine-tune. The frontier moves fast enough that fine-tunes obsolete within months; a stronger base model with better prompts ports forward.

5.3 Guardrails

Threat model is wider than for ordinary services because the LLM itself is an attack surface.

Input side:

- **Prompt injection is unsolved.** Anything reaching the prompt — retrieved docs, tool outputs, user content, search results — can carry instructions the model will follow. Mitigations: clear delimitation, tool sandboxing, withholding irreversible-action privileges from injected-content paths, output filtering. The design objective is blast-radius reduction; injection-immunity claims are not credible.
- **PII redaction** before logging or third-party calls. Use a dedicated NER/PII model, not the main LLM (which has its own injection risk).
- **Jailbreak detection.** Lightweight classifier (Llama Guard, ShieldGemma, custom) on user input. Useful, not sufficient — defense in depth.

Output side:

- **Output filtering.** Same approach, applied to model output before user-visible delivery.
- **Schema validation.** As in §3.4. Invalid → repair or reject.
- **Faithfulness check.** For grounded responses, verify the answer is supported by retrieved context (LLM-judge or NLI model). Reject or flag unsupported claims.

Operational:

- **Data residency.** Regulated workloads need inference in jurisdictions you control. Embeddings aren't exempt — they encode the source content.
- **Tenant isolation.** Vector indexes, caches, conversation history must be partitioned by tenant. Cross-tenant retrieval is one of the most common LLM-app data leaks.
- **Rate limits and budget caps.** Per-user, per-tenant, global. Without these, one misbehaving client (or attacker) can exhaust your monthly model budget.

5.4 Observability

Standard metrics aren't enough. You need qualitative trace data alongside quantitative telemetry.

Per-request:

- Full trace: input messages, system prompt version, model + params, tool calls with args and results, intermediate "thinking" content, final output.
- Token counts (input cached / input uncached / output / reasoning) with cost.
- Latency: time-to-first-token, time-to-completion, per-tool-call duration.
- Outcome: success / schema-failure / refusal / tool-error / loop-budget-exceeded.

Aggregate:

- p50/p95/p99 latency.
- Cost per request, per user, per workflow.
- Eval scores running on a sampled slice of prod traffic.
- Drift indicators: faithfulness, refusal rate, tool-selection distribution, output-length distribution. Statistically significant shifts here usually precede user-visible regressions.

User feedback:

- Explicit (thumbs up/down, written feedback).
- Implicit (regenerations, edits, abandoned sessions, follow-up corrections).

Implicit signals are noisier but higher-volume. Both feed back into the eval set.

A/B and canary. Serve two variants in parallel to a slice of traffic. Compare on the metrics above with proper statistical tests (two-sample with multiple-comparison correction). A/B comparison is the only mechanism that validates a prompt or model change against a real-traffic baseline; without it, changes ship blind.

Streaming UX. Tokens stream, users hit cancel, agents take 30 seconds. Implement clean cancellation: propagate stop signals to the provider, abort in-flight tool calls, clear partial state. Otherwise your system happily burns tokens on responses no one will read.

5.5 Pre-launch checklist

Model and config. Picked against eval set. Temperature/sampling tuned per task. Routing and fallback chains defined. Cache breakpoints placed for stable prefixes. Cost and latency budgets set.

Prompts. System prompt versioned. Stable prefixes positioned for caching. Few-shot (static or dynamic) in place. Critical instructions positioned to survive lost-in-the-middle. Tool schemas reviewed for clarity and minimality. Prompts in CI with eval gates.

RAG. Hybrid retrieval (vector + BM25) with RRF. Metadata pre-filtering enforced for tenant and ACL boundaries. Reranking active. Chunk enrichment / contextual retrieval applied. HNSW parameters benchmarked against exhaustive baseline. Retrieval evaluated in isolation.

Agents. Tool schemas validated. Toolset sized appropriately per agent. Loop / token / cost budgets enforced. Tool errors structured. Tool outputs sanitized. Code execution sandboxed. HITL on destructive actions in trusted UI. Trajectory eval set in place.

Safety. Input and output filtering deployed. PII redaction before logging and third-party calls. Red-team suite run; failures triaged. Bias evaluation run. Prompt-injection test cases in eval set.

Evaluation. Error taxonomy from open/axial coding. LLM judge calibrated against human gold set with held-out test split. Multi-judge ensemble for high-stakes evals. Drift monitoring with alerts. Eval gates in CI for prompt and model changes.

Telemetry. Full traces logged with PII redaction. Token, cost, latency captured per request. Implicit and explicit feedback collected. A/B framework operational. Streaming cancellation handled cleanly.