

AI Spend Audit Checklist

A public, reproducible checklist for auditing metered AI/LLM spend (API model usage). Designed for companies of roughly 50–500 people spending \$5k+/month on model inference, but usable at any scale. Every finding this audit produces must be recomputable by a finance team from provider invoices - if a number can't be reproduced from billing data, it doesn't go in the report.

Core principle (sequence is fixed): Visibility → Attribution → Baseline → Leak detection → Controls → Ranked fixes. You cannot cut a cost until you know which feature is creating it.

[AI Spend Audit Checklist](#)

[Part 0 - Access & prerequisites](#)

[Part 1 - Inventory & shadow spend discovery](#)

[Part 2 - Cost attribution](#)

[Tagging schema - every model call should carry:](#)

[Attribution quality checks:](#)

[Part 3 - Baseline freeze](#)

[Cost baseline \(per feature, from provider billing data\):](#)

[Quality baseline \(set BEFORE any optimization\):](#)

[Measurement rules \(agree these in writing now, not after the fact\):](#)

[Part 4 - Leak detection sweep](#)

[4.1 Model-tier mismatch \(typically the biggest leak\)](#)

[4.2 Token waste](#)

[4.3 Caching](#)

[4.4 Batching & scheduling](#)

[4.5 Agentic & retry pathologies](#)

[4.6 Zombie & duplicate spend](#)

[Part 5 - Controls & governance check](#)

[Part 6 - Rate & commercial review](#)

[Part 7 - Ranking & written diagnosis](#)

[Appendix A. Core KPIs to compute during the audit](#)

[Appendix B. Common failure modes this checklist exists to catch](#)

[Sources & method](#)

Part 0 - Access & prerequisites

Everything in this audit is read-only. No production data or customer content needs to leave the environment.

- ☐ Read-only access to billing/usage exports from **every** model provider in use (OpenAI, Anthropic, Google Vertex, Azure AI Foundry, AWS Bedrock, Mistral, open-weight hosts like Together/Fireworks, etc.)
- ☐ Cloud billing exports (AWS/GCP/Azure) filtered to AI-related SKUs - managed model services, GPU instances, vector DBs, embedding services
- ☐ Gateway/proxy configuration (LiteLLM, Portkey, OpenRouter, Bifrost, custom) - or confirmation that no gateway exists (that itself is a finding)
- ☐ Repository read access scoped to directories containing model calls, prompts, and orchestration code (LangChain/LlamaIndex/LangGraph/custom)
- ☐ Observability/tracing data if present (Langfuse, Helicone, Datadog LLM Observability, Braintrust, OpenTelemetry/OpenLLMetry)
- ☐ Finance contact identified who will recompute and sign off on numbers
- ☐ Customer identifiers hashed before any cost attribution - never attribute against personal data
- ☐ NDA / data processing terms in place if a third party runs the audit

Part 1 - Inventory & shadow spend discovery

Goal: a complete list of everywhere money leaves the building for AI. Most organizations fail this step; the majority of tech executives lack full real-time visibility into AI spend.

- ☐ List every model provider account, org, and project - including free-tier and trial accounts that may have converted to paid
- ☐ List every active API key per provider; map each key → owning team → application → environment
- ☐ Flag keys with no identifiable owner or purpose (candidates for immediate rotation/revocation)
- ☐ Search expense reports and corporate card statements for AI tool charges (per-seat subscriptions, individual API accounts) not visible in central billing
- ☐ Identify AI features bundled inside existing SaaS contracts (premium AI tiers, usage add-ons) that never went through AI procurement
- ☐ Identify model calls that **bypass the gateway** - direct SDK calls, third-party integrations calling providers directly, scripts with hardcoded keys. Compare gateway request volume against provider-reported request volume; the gap is bypass traffic. (This is one of the most common and most valuable audit findings.)
- ☐ Check for committed-spend contracts, provisioned throughput (e.g., PTUs), or scale-tier agreements - record floors, terms, and renewal dates. **If a commitment floor sits above current usage, consumption cuts save nothing until it is renegotiated. Record this before optimizing anything.**
- ☐ Record self-hosted/open-weight inference costs (GPU instances, vLLM/SageMaker endpoints) - these are AI spend even though they don't appear on a model provider invoice

Output: AI spend inventory - one row per provider/account/key with owner, purpose, and monthly cost.

Part 2 - Cost attribution

Goal: the invoice stops being one number and becomes a list of features with a price next to each.

Tagging schema - every model call should carry:

- ☐ **Feature / endpoint** that triggered it (the single most valuable dimension)
- ☐ **Environment** (production / staging / development / test / CI)
- ☐ **Customer or tenant** (hashed ID) - needed for unit economics and abuse detection
- ☐ **Exact model and version** ("gpt-4" is not granular enough; pricing differs by variant and version)
- ☐ **Token counts, split by type:** input, output, cached-read, cache-write, and reasoning/thinking tokens. Never conflate cached and uncached input tokens - cache discounts range from ~50% to ~90% depending on provider, and conflating them corrupts every downstream number
- ☐ **Request metadata:** retry count, agent/session ID, tool-call depth (for agentic workloads)

Attribution quality checks:

- ☐ Compute % of **total spend that is attributable** to a feature (target: >90%; report the actual number)
- ☐ Everything unattributable is **flagged and listed, never smoothed over** or prorated silently - unexplained spend is a finding, not a rounding error
- ☐ Reconcile attributed totals against provider invoices monthly - attributed + flagged must equal the invoice
- ☐ Verify dev/test/experiment traffic is separated from production (dev workloads quietly running on production keys is a recurring leak)
- ☐ For multi-agent systems: attribute at the session/run level, not just the call level, so a single user action's true cost (all sub-calls, retries, tool calls) is visible

Output: Cost-per-feature dashboard, rebuilt weekly, reconciled to invoices.

Part 3 - Baseline freeze

Goal: a starting position frozen from the organization's own invoices, so every later claim of savings has a denominator nobody can argue with.

Cost baseline (per feature, from provider billing data):

- ☐ Cost per request
- ☐ Tokens per call (input / output / cached / reasoning, separately)
- ☐ Calls per feature per period
- ☐ Cost per unit of business work (per document, per ticket, per user session - pick the unit that maps to the product)
- ☐ Current list price per model in use, snapshotted with date (prices change; savings math must pin the price version)

Quality baseline (set BEFORE any optimization):

- ☐ A **quality pass mark per feature**, agreed with the feature owner (accuracy %, rubric score, user-satisfaction threshold, task-completion rate - whatever fits)
- ☐ An automated quality test harness that can evaluate a candidate change against that pass mark (custom evals or off-the-shelf tools)
- ☐ Baseline latency per feature (some optimizations trade latency; know the starting point)

Measurement rules (agree these in writing now, not after the fact):

- ☐ Savings are measured as **change in cost per unit of work**, never change in the monthly total - traffic growth must neither create nor erase a claimed saving
- ☐ A change counts only after it **holds its quality pass mark for 30 days in production**; reverted changes are struck from the count
- ☐ **Provider list-price cuts are excluded** from claimed savings
- ☐ Finance recomputes every claimed line from provider invoices; anything they can't reproduce is struck

Output: Frozen baseline document - signed/dated, with the measurement rules attached.

Part 4 - Leak detection sweep

Goal: find where money is being wasted. Work through each category; for each hit, estimate monthly recoverable amount at current list prices.

4.1 Model-tier mismatch (typically the biggest leak)

- ☐ Classify traffic into job types: extraction, classification, scoring, tagging, rewriting, summarization, open-ended generation, reasoning
- ☐ For each job type, ask: does this need the top-tier model, or would a mid-tier or small model hold the quality pass mark? (Flagship-to-small output-token price gaps run roughly 15–20×; simple sentiment or classification tasks rarely need frontier reasoning)
- ☐ Flag reasoning/thinking modes enabled on tasks that don't need them - reasoning tokens bill as output and several APIs enable them by default
- ☐ Flag any single narrow, high-volume job dominating the bill - candidate for a fine-tuned or purpose-built small model (only worth it if projected saving clearly exceeds training + hosting + maintenance; expect this to apply in a minority of audits)

4.2 Token waste

- ☐ Measure system-prompt size × monthly invocation count per feature - an 800-token system prompt across 100k calls/month is 80M input tokens before any user content
- ☐ Identify few-shot examples, boilerplate, and instructions in prompts that no longer earn their token cost (test removal against the quality harness)
- ☐ Check for uncapped output lengths - set max-token limits per feature based on observed p95 need
- ☐ Check RAG pipelines for over-retrieval (stuffing whole documents where passages suffice; retrieval commonly multiplies input size 4–6×)
- ☐ Check conversation/agent memory strategy - unbounded context accumulation means every turn resends the full history; look for missing summarization/truncation/compaction

4.3 Caching

- ☐ Measure what % of input content is repeated across calls (system prompts, templates, tool schemas, retrieval headers)
- ☐ Verify repeated content is structured as a **contiguous prefix block** so provider prompt caching actually engages (cached input typically bills at ~10–50% of standard depending on provider)
- ☐ Measure actual cache hit rate vs. potential; misordered prompts silently defeat caching
- ☐ Evaluate semantic/response caching for high-repetition query patterns (FAQ-like traffic)

4.4 Batching & scheduling

- ☐ Identify deadline-free workloads (tagging jobs, backfills, nightly enrichment, report generation) still running on the real-time endpoint - batch queues typically price at ~50%
- ☐ Check whether batch-eligible traffic share is being exploited at all

4.5 Agentic & retry pathologies

- ☐ Measure calls per user-visible task for agent workloads; investigate outliers (production agent runs commonly consume 3–10× prototype estimates; retry loops can reach 50×)
- ☐ Look for retry storms: failed calls retried with identical oversized context, tool integrations retrying on partial responses, context-window-exceeded errors triggering blind retries
- ☐ Check for redundant re-reads: agents re-fetching system prompts, tool descriptions, or documents every step
- ☐ Verify circuit breakers exist: per-run token ceilings, max-step limits, and kill switches for runaway agents
- ☐ Check retry/backoff configuration on all API error handling (exponential backoff with caps, not infinite retries)

4.6 Zombie & duplicate spend

- ☐ Find features or endpoints with cost but no (or negligible) usage/user engagement - orphaned experiments still burning tokens
- ☐ Find duplicate capability: multiple teams paying for the same job via different providers/tools
- ☐ Find scheduled jobs (heartbeats, polls, cron-driven calls) whose frequency exceeds any business need
- ☐ Check provisioned/reserved capacity utilization (PTUs, provisioned throughput, GPU reservations) - low utilization on fixed capacity is pure waste

Output: Leak register - every leak with mechanism, affected traffic share, monthly recoverable estimate at current prices, and confidence level.

Part 5 - Controls & governance check

Goal: verify the guardrails that prevent the next surprise bill. Absence of any item below is a finding.

- ☐ Budget alerts per provider account AND per feature/team (account-level alone is too coarse)
- ☐ Hard caps or throttles available for non-critical workloads
- ☐ Per-user / per-tenant spend tracking with anomaly alerts (e.g., single user exceeding \$X/24h)
- ☐ Anomaly detection comparing spend against historical baseline (spikes investigated within 24h, not at month-end)
- ☐ API key lifecycle governance: provisioning process, naming convention, owner mapping, rotation, revocation on offboarding
- ☐ Gateway coverage policy: new model calls must route through the gateway; bypass is detectable and flagged
- ☐ Quality regression alerts alongside cost alerts (a cost drop with a quality drop is not a saving)
- ☐ Model version pinning / change control (silent provider model swaps change both cost and quality)
- ☐ Spend visibility shared with the teams generating it (showback at minimum) - cost data only finance sees changes nothing
- ☐ A named owner for AI spend overall, and per feature

Part 6 - Rate & commercial review

- ☐ Compare effective paid rates against current list prices per model (are negotiated discounts actually applying?)
- ☐ Review committed-spend agreements: floor vs. actual usage, renewal dates, renegotiation triggers
- ☐ Evaluate batch, flex/priority tiers, and cache pricing actually available vs. actually used
- ☐ Track provider price changes weekly - savings math and routing decisions go stale otherwise
- ☐ For sustained predictable volume: evaluate commitment discounts - but only after 60-90 days of stable production usage, never on projections

Part 7 - Ranking & written diagnosis

Goal: a short written diagnosis someone can fund from.

- ☐ Rank every leak by **return per engineering day** (monthly recoverable ÷ estimated implementation effort), not by raw size
- ☐ For each of the top items: the fix, its cost to implement, the quality risk, and how the result will be verified
- ☐ State explicitly what could not be attributed and why
- ☐ State explicitly if nothing on the list is worth doing - that is a legitimate finding
- ☐ Recommended implementation order (typically: token discipline first - it shrinks everything downstream - then caching, batching, routing; purpose-built models last and only if one workload justifies it)
- ☐ Re-check every costed figure against current prices before publishing

Output: Diagnosis memo - spend map, leak register ranked by return per engineering day, unattributed-spend appendix, measurement rules.

Appendix A. Core KPIs to compute during the audit

KPI	Formula	Why
Cost per request/inference	total inference cost ÷ request count	Core unit-economics denominator
Cost per feature	attributed spend by feature tag	Makes the invoice actionable
Cost per unit of business work	feature cost ÷ business units processed	Connects spend to value
Tokens per call (by type)	input / output / cached / reasoning, separately	Reveals prompt bloat & cache health
Cache hit rate	cached-read tokens ÷ cacheable tokens	Measures caching effectiveness
Attribution coverage	attributed spend ÷ total invoice	Audit-quality metric; report it
Calls per task (agents)	model calls ÷ user-visible tasks	Surfaces loops and retries
Model-task quality alignment	quality score needed vs. quality score of model used	Flags premium models on easy jobs
Anomaly rate	spend spikes flagged ÷ period	Governance health

Appendix B. Common failure modes this checklist exists to catch

1. Optimizing **before** attributing (guessing which feature costs money)
 2. Measuring **monthly totals** instead of cost per unit (growth masquerading as failure, or shrinkage as success)
 3. **Conflating** cached and uncached input tokens in the baseline
 4. Ignoring gateway-bypass traffic (the invoice and the dashboard disagree, and **nobody reconciles** them)
 5. Shipping cost cuts **without a quality pass** mark (silent degradation)
 6. Counting savings from provider **price cuts** as optimization wins
 7. Dev/**test traffic** on production accounts
 8. Agent **retry loops** discovered at invoice time instead of alert time
 9. Committed-spend floors making consumption cuts worthless until renegotiated
 10. **Shadow AI**: keys, subscriptions, and bundled SaaS AI features invisible to central billing
-

Sources & method

Built from three inputs: (1) the audit methodology described publicly at otono.me (basis · AI cost engineering) - cost attribution first, frozen per-unit baselines, quality pass marks, invoice-recomputable savings; (2) FinOps Foundation materials (finops.org): *FinOps for AI Overview*, *Token Economics: Managing AI Value in SaaS-Model Token Costs*, the FinOps Framework capabilities (allocation, anomaly management, unit economics, rate optimization), and the crawl-walk-run maturity model; (3) practitioner reports from engineering and FinOps communities (Reddit r/FinOps, r/LLMDevs and adjacent agent-infrastructure communities, LLM observability vendors' field writeups) on shadow AI spend, agent retry economics, context accumulation, cache-token accounting, and gateway bypass.

License suggestion if published: CC BY 4.0 (matching FinOps Foundation content licensing, which requires attribution for adaptations).