AI API Cost Analysis

What the conversational AI costs to run, what drives the token bill, and the knobs that control it. The conversational AI is the one feature an outside vendor bills you for each time it runs. This document explains what makes that bill go up or down, and covers the REST (web service) interface that feeds it, so an engineering and FinOps (finance-operations) reader can size spend, set budgets, and decide where to cap.

Audience: engineering, FinOps / finance owning AI and API spend

Scope: the metered (per-use billed) AI layer (LangChain4j → your model endpoint) and the wider REST surface that feeds it

Document: engineering reference, 21 June 2026

Contact: contact_us@cybriq.io

0 How to read this document

There is exactly one metered, per-token cost path in CrossConnect: the conversational AI. (Metered means you pay per use; per-token means the model charges by the small chunks of text it reads and writes.) Everything else runs on infrastructure you already own and have already paid for: CPU, memory, and the Postgres database. This document leads with the AI cost model because it is the only line item with an outside vendor invoice attached, then sets it against the wider API surface so you can see how a single AI question fans out into the database underneath it.

GA shipped & on by default   **CONFIGURABLE** shipped, operator-tuned

COST LEVER a knob that raises or lowers the bill

Every model parameter, default, and property name below is read straight from the shipping code and the `application.yml` config file, so the facts are real. The cost *math* (the latency bands, endpoint tiers, and how cost grows with fleet size) is a directional estimate from reading the code paths, not a measurement taken under load. Treat the relative ranking (which thing costs more than which) as solid, and the absolute numbers as a starting budget to confirm once you run it under load. We do not quote model list prices here, since those change often. Your bill is the model id and the token counts described below, multiplied by your provider's current price list.

1 The AI cost model in one page
2 Anatomy of one AI request
3 What drives the token count
4 Defaults, model, and provider switching
5 The cost levers
6 The stub path: AI off, zero spend
7 Audit, attribution & chargeback
8 The wider REST surface that feeds the model
9 The heaviest endpoints behind a turn
10 Caching, the non-AI cost multiplier
A AI cost configuration reference
B Endpoint cost tiers

1 The AI cost model in one page

The only spend that grows as you use the product more is the model API (the call out to the AI service). Each call to `POST /api/v1/assistant/ask` sends a prompt (the question plus its supporting data) to your model endpoint and is billed per token on both sides: the text going in and the text coming back. The cost of a single turn (one question and answer) is the tokens in the request times the input rate, plus the tokens in the reply times the output rate. Nothing else on the platform carries a per-call charge.

claude-opus-4	1024	4	stub
default model id	default max output tokens	max tool-call loop iterations	default provider (no spend)

Two facts frame the whole bill. First, the AI ships **off by default**. Out of the box the provider is the built-in `stub`, a fixed renderer that formats data mechanically and never calls a paid model, so a fresh install has zero AI spend (§6). You turn the paid AI on deliberately. Second, once it is on, the **input** side drives the cost, not the output. The system prompt (the assistant's standing instructions), the tool-result data, and any multi-step agentic loop are all input tokens, and together they usually dwarf the size-limited reply.

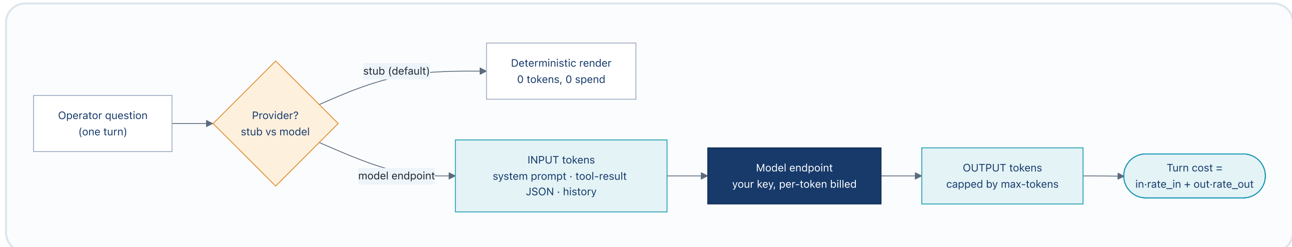

Figure 1. The metered path. With the default `stub` provider a turn costs nothing. Point the assistant at a paid model endpoint and a turn is billed on input plus output tokens. The input side (prompt + tool-result data + any loop) is the larger half, and the one you can most easily control.

2 Anatomy of one AI request

A turn is not always a single model call. It is a prompt assembled from several parts, sometimes a short loop where the model fetches data step by step, and a size-limited reply. The cost is simply the total tokens that assembly adds up to. The request lifecycle is implemented in `ai/AssistantService.java` and the two client adapters (`AnthropicLlmClient`, `ChatModelLlmClient`).

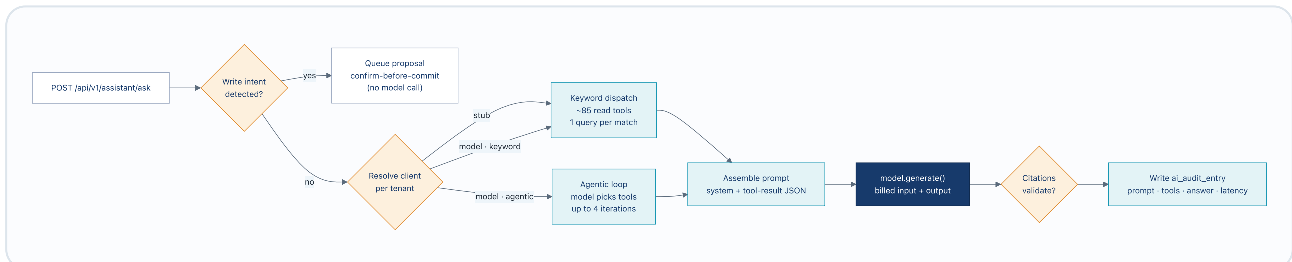

Figure 2. One turn, end to end. A prompt that asks to change something stops early: it becomes a proposal a human must confirm, and never reaches the paid model. A read-only prompt gathers tool results (by matching keywords, or in agentic mode by a model-driven loop capped at four steps), generates the answer once, checks its citations, and is fully logged. Every model call is a potential billing event, and the keyword and change-request paths limit how often one fires.

One design rule keeps cost bounded: the AI only advises, it never makes a change itself. Anything it proposes is queued for a human to approve, which means the expensive open-ended "go fix it" agent loop simply does not exist here. The most a single turn can spend is fixed in advance by the four-step tool cap and the output-token ceiling.

3 What drives the token count

Tokens are the unit you pay for, so controlling the bill means controlling the tokens. Five things set the size of a turn. The system decides the first three; the user shapes the last two.

Driver	What it adds	Who controls it
System prompt	A fixed block of instructions (the assistant's rules plus how it must cite sources) added to the start of every turn. A constant baseline of input tokens on every turn.	Set by the product, not per question. The same for every question.
Tool-result JSON	The records the tools fetched, written into the prompt as data. This is the largest and most variable input. A "list every device" answer carries far more tokens than "is device X up?"	The question, plus limits on result size. The stub render trims lists to 12 items.
Conversation history	Earlier turns carried forward when a conversation continues, so a long thread drags its own tail along with it.	How long the conversation is; start a fresh one to reset.
Agentic loop depth	In agentic (multi-step) mode, each step re-sends the growing message list (the earlier tool calls and their results) to the model. Up to four steps, so a deep chain of tool calls multiplies the input tokens.	`CROSSCONNECT_AI_AGENTIC` and the 4-iteration cap.
Output length	The reply, hard-capped by `max-tokens` (default 1024). The one part of the turn with a firm ceiling.	`CROSSCONNECT_AI_MAX_TOKENS`.

> **The tool-result data is the cost center.** In the keyword path, the prompt is matched against a large keyword-dispatch table (~85 distinct read tools, gated by 230+ keyword phrases), and each tool that matches runs exactly one read query, whose result is packed into the prompt the model sees. A broad question that matches several patterns, or returns a large list, drives the input token count up directly. In agentic mode the model picks from a catalog of available tools and can chain calls across up to four steps, so the same broad question can cost several round-trips to the model instead of one.

The takeaway for a budget: a narrow question about one thing ("status of device X") is a small, predictable turn. A broad question across the whole fleet ("summarize every compliance gap") is the expensive one, because it drags a large set of results into the prompt. Output is capped, so input is where a question gets expensive.

4 Defaults, model, and provider switching

The AI layer can swap between providers through LangChain4j (`0.36.2`), so you are not locked to one vendor. The deployment-wide default, plus any per-customer override, decides which model id applies, and therefore which price list. All values below are read from the shipping configuration.

Setting	Default	Cost effect
Provider	`stub`	Zero spend until you switch to a paid model provider. OFF BY DEFAULT
Model id (deployment)	`claude-opus-4-20250514`	Picks the price list. This is the highest-capability tier; the per-token rate is the lever, see below.

Setting	Default	Cost effect
Temperature	`0.0`	Makes answers repeatable. No direct cost effect, but steady answers cut down on users re-asking, which would cost again.
Max output tokens	`1024`	Hard ceiling on the output half of every turn.
Per-tenant model fallback	`claude-sonnet-4-20250514`	When a customer turns on AI without naming a model, the system picks the faster, cheaper Sonnet tier rather than Opus.
Per-tenant max-tokens floor	`2048`	Customer-configured and agentic/streaming paths set output no lower than 2048 so detailed, tool-heavy answers are not cut off, a deliberate trade of a higher ceiling for completeness.

> **Model choice is the single biggest rate lever.** The deployment default is the top-capability model. The system already steers *per-customer* AI to a faster, lower-cost model when no model is named, which tells you the intended dial: send routine, high-volume questions to the cheaper tier and save the premium tier for the questions that truly need it. Switching the model id is a configuration change, not a code change.

Provider switching. A customer (tenant) can bring their own account and endpoint. The resolver (`ai/LlmClientResolver.java`) reads that customer's `ai_settings` (provider, model, base URL, temperature, max-tokens, encrypted key) and builds a matching client against an Anthropic, OpenAI, or any OpenAI-compatible endpoint. The deployment-wide default applies when a customer has no settings of their own. This matters for cost ownership: spend lands on, and can be capped by, the customer whose key served the call.

5 The cost levers

Every setting that moves the bill, roughly in order of how much it matters. All are configuration; none require a code change.

1 · Provider on/off COST LEVER

Leave the default `stub` and AI spend is zero. The platform still answers from tool results, just without AI-written prose. The largest lever is whether the paid path runs at all.

2 · Model tier COST LEVER

The model id sets the per-token rate. Send routine questions to the cheaper tier and save the premium model for the hard ones. Per-customer settings let you mix tiers across customers.

3 · Max output tokens COST LEVER

Caps the output half of every turn. Default 1024; the agentic/per-customer floor is 2048. Lower it to cap reply cost, and raise it only where a cut-off answer would hurt.

4 · Agentic vs keyword COST LEVER

Agentic mode can chain up to four trips to the model for one question; keyword mode is a single trip over data already fetched. Forcing keyword mode (`CROSSCONNECT_AI_AGENTIC=false`) trades some flexibility for one call per turn.

5 · Question scope

Tool-result data is the input cost center. Narrow questions return small results; fleet-wide questions drag large lists into the prompt. List caps (the stub trims to 12) keep the worst case in check.

6 · Bring-your-own key CONFIGURABLE

Per-customer keys mean spend lands on that customer's own account, so cost is traceable to them and can be rate-limited at the provider, instead of being pooled on one shared deployment key.

The built-in cost ceiling. Because the AI never makes changes itself and every turn is capped at four tool steps and a fixed output budget, there is no way for it to run away into an unbounded bill. The most expensive single turn is knowable in advance: four trips to the model over the tool-result set, plus a size-limited reply. That predictability is the point.

6 The stub path: AI off, zero spend

When the paid AI is off, the platform does not go dark. The default `stub` provider (`ai/StubLlmClient.java`) is a fixed renderer: it runs the same read tools, then formats the results into plain markdown without ever calling a model. No model call means no tokens and no spend.

Aspect	Stub (default)	Model provider
Per-turn cost	Zero, no outside call	Input + output tokens at your rate
Selected by	`CROSSCONNECT_AI_PROVIDER=stub` (default, also what it falls back to when no key is set)	`CROSSCONNECT_AI_PROVIDER=langchain4j` + a key
Answer style	Plain, mechanical formatting of tool results (lists trimmed to 12)	AI-written natural-language prose with citations
Grounding	Citations come from the tool-result data; it cannot make facts up	Citations are checked against the tool results before the answer is shown

The default is the free tier. Out of the box, `crossconnect.ai.provider` resolves to `stub`, and the paid model client only switches on when the provider is set to `langchain4j` *and* an API key is present. So a misconfiguration fails safe toward zero spend, not toward a surprise bill. Turning the paid AI on is an explicit, logged decision.

7 Audit, attribution & chargeback

Every turn that runs is recorded, which is what lets you manage AI spend rather than treat it as an opaque line item. The `AiAuditEntry` row (table `ai_audit_entry`) is written for every turn in `AssistantService` and is scoped to one customer (tenant).

Field	What it captures	Why FinOps cares
`tenant_id`	Which customer the turn ran for	Charges spend to a customer or cost center
`prompt`	The user's question	Shows which questions are expensive
`tool_calls_json`	The tools used and what they returned	Shows what pulled data into the prompt, the main driver of input tokens
`citations_json` · `citations_ok`	The citations, and whether they passed validation	A rejected answer is spend that was wasted, worth tracking
`answer`	The text the AI returned	The output-token side of the turn
`latency_ms`	How long the turn took, in milliseconds	A stand-in for how heavy the turn was; tracks with token count
`created_at`	When it ran (timestamp)	Spend over time, and per-customer usage trends

One gap, stated plainly. The audit entry records the prompt, tool calls, answer, and latency, but the current database schema does **not** store a per-turn token count or dollar figure. So today, putting a dollar amount on a single turn is an estimate: count tokens from the saved prompt and answer, or match it up against the provider's own usage console using the API key in play. The clean fix would be a built-in token/cost column, and that is not in the current build.

8 The wider REST surface that feeds the model

The AI does not run on its own. Every tool it uses is a real call into the same web service the operators use, and those calls carry their own cost. That cost is infrastructure (servers you already own), not a per-use vendor charge. This surface is what makes one AI question cheap and another heavy: the difference is which endpoints the tools hit underneath. The surface is roughly **640 endpoints across 115 controllers**. By method, that is about 242 GET (reads), 165 POST, 68 DELETE, 34 PATCH, and 18 PUT. Reads make up most real traffic.

Tier	Typical latency	Shape	Share
Cheap	< 100 ms	1–2 tables, a fixed-size lookup or single-row write (cost does not grow with fleet size)	~210 endpoints
Moderate	100–500 ms	2–5 tables, a list with a join; cost grows with the number of devices	~270 endpoints
Heavy	0.5–5 s	5–15 tables, a fleet-wide pass or graph layout (usually cached)	~145 endpoints

Tier	Typical latency	Shape	Share
Very heavy	5 s – minutes	Batfish, the AI model, or live SNMP/SSH to devices; limited by outside input/output, not the database	~15 endpoints

Four things set an endpoint's infrastructure cost: how many trips it makes to the database, how fast that work grows as the fleet grows, whether it reaches a heavy subsystem (Batfish, the model, live SNMP/SSH, or flow processing), and whether the result is cached (saved and reused). The first three set the raw cost; the cache divides it down. The very-heavy tier reaches out to an outside process, and the AI model is one of those.

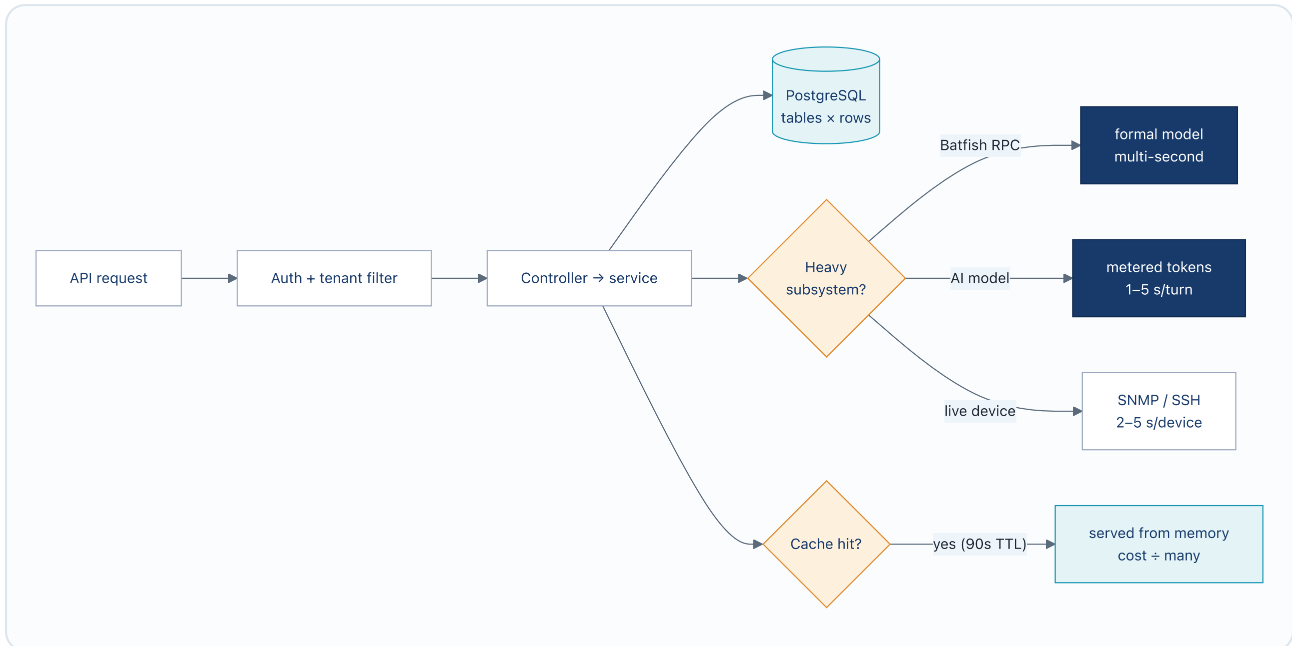

Figure 3. Where a call spends. An endpoint that stays inside the Postgres database ranges from cheap to heavy, but it is all infrastructure you already own. The moment it crosses over to the AI model it becomes the one per-token, billed path. Crossing to Batfish or a live device makes it slow and input/output-bound, but it is still not billed per call. A 90-second cache divides the cost of the repeatable fleet-wide reads.

9 The heaviest endpoints behind a turn

When the AI answers a fleet-wide question, the tools it runs can land on the endpoints below. The top of the list is the AI call itself; the rest are the heavy reads it may pull data from. These are the ones to measure and protect first.

#	Endpoint	Why it is expensive	Tier
1	`POST /assistant/ask`	A round-trip to the model (1–5 s, billed by token). Each tool it calls runs its own query, and agentic mode can loop up to four times. The only billed path. Fully logged.	very heavy
2	`GET /change-safety`	Two multi-second Batfish calls on the shared session, plus a live-traffic flow scan, combined into one verdict.	very heavy

#	Endpoint	Why it is expensive	Tier
3	`POST /discovery/run/{id}/stage`	A live SNMP walk of each device (2–5 s each), limited by the network; optionally pulls config over SSH.	very heavy
4	`POST /imports/{id}/stage`	Pulls every object from an outside source-of-truth system, maps the data, and writes it to staging. 10 s–minutes.	very heavy
5	`GET /topology/diagram.svg`	Builds the network graph, lays it out, and renders the SVG image. 1–3 s for large fleets; not cached.	heavy
6	`GET /atlas/overview`	Reads the whole fleet and tallies it up by role, vendor, and site. Grows with the number of devices; a 90s cache that computes once absorbs repeats.	heavy
7	`GET /data-quality`	Walks devices × interfaces × IPs × configs to check for phantom records, shadow IT, drift, and contradictions. Cached.	heavy
8	`GET /compliance/frameworks/{key}`	Takes a per-device snapshot of signals, then runs every compliance check over it. Cached 90s, computed once.	heavy
9	`GET /hotspots`	Rolls up 20+ signal services into one ranked work queue. Cached.	heavy
10	`GET /maturity`	Builds an operational-maturity snapshot from 8–12 data sources. Grows with the number of devices; cached.	heavy

> **The pattern.** Only one endpoint here, the AI call, carries a per-token bill. The rest are limited by input/output or CPU on infrastructure you already own. The cost lesson for AI is a knock-on one: an AI question that sets off a heavy fleet-wide read both pays its own token cost *and* packs that large result into the prompt, which is exactly the input-token inflation from §3. Narrow the question, narrow the bill.

10 Caching, the non-AI cost multiplier

The fleet-wide reads all work the same way: a short-lived result cache that **computes once and shares the result**. The first caller does the work; any other callers that arrive during the window wait on that one computation instead of each launching their own. When the cache already has the answer, a heavy endpoint serves it from memory in single-digit milliseconds.

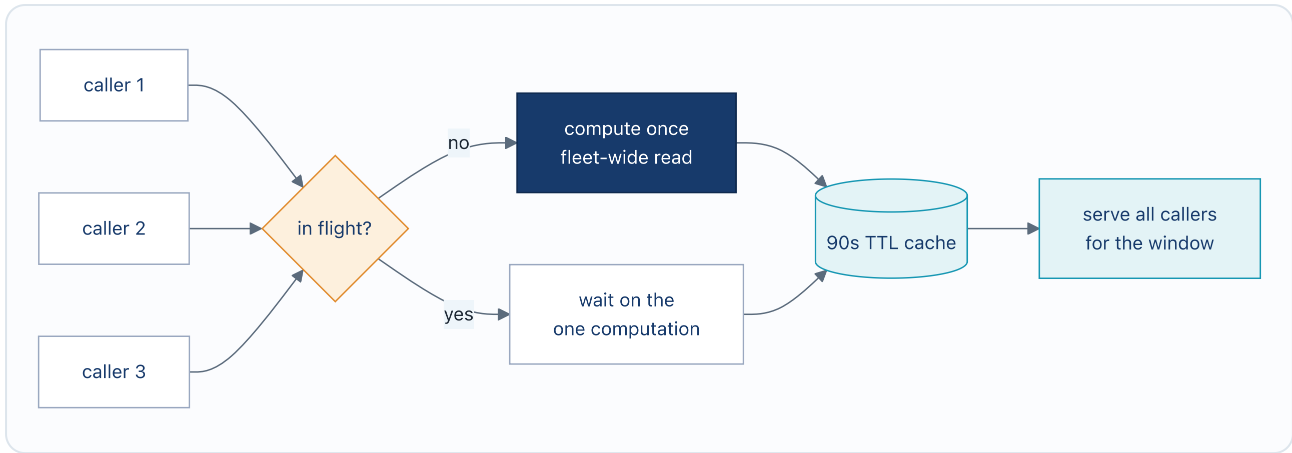

Figure 4. Compute-once caching. A 90-second cache plus the compute-once rule turns a worst case of "every page load re-runs the whole fleet audit" into "one computation serves every caller for the window." It caps the infrastructure cost of the heavy reads, including the ones an AI tool call might hit.

Where caching is **missing** matters too: the topology graph/diagram, imports, and committing observations to the source of truth are not cached, because each result is specific to one request and cannot be reused. Those are protected instead by per-endpoint rate limits and a separate pool for heavy work, so they cannot swamp the rest of the system. None of this is metered spend; it is the infrastructure cost behind a free (stub) or paid (model) AI answer alike.

A Appendix, AI cost configuration reference

Environment variables drive the AI cost settings. The defaults favor zero spend, and turning AI on and choosing a model are both explicit, deliberate steps. Every key and default below is read from `application.yml` and the client code.

Control	Property / env	Default	Cost role
AI provider	`CROSSCONNECT_AI_PROVIDER`	`stub`	The master on/off switch; `stub` = no spend, `langchain4j` = paid
Model id	`CROSSCONNECT_AI_MODEL`	`claude-opus-4-20250514`	Picks the price list
Temperature	`CROSSCONNECT_AI_TEMPERATURE`	`0.0`	Makes answers repeatable; no direct cost
Max output tokens	`CROSSCONNECT_AI_MAX_TOKENS`	`1024`	Caps the output half of a turn
Agentic mode	`CROSSCONNECT_AI_AGENTIC`	`true`	`false` forces single-trip keyword mode
Provider API key	`ANTHROPIC_API_KEY`	(unset)	An empty key keeps the stub active, so empty = no spend
Write-intent TTL	`CROSSCONNECT_AI_INTENT_TTL_SECONDS`	`900`	How long a queued change proposal lives; proposals never call the model

Control	Property / env	Default	Cost role
Per-tenant settings	`ai_settings` (DB, encrypted key)	per tenant	Provider, model, base URL, temperature, and max-tokens per customer; the default fallback model is `claude-sonnet-4-20250514` , with max-tokens set no lower than 2048

B Appendix, endpoint cost tiers

A few example endpoints in each tier, to show the non-billed cost that sits behind any AI turn. The rest of the surface follows these same patterns.

Tier	Cost driver	What protects it	Example endpoints
Cheap	One quick lookup or write	Database indexes	`GET /devices/{slug}` , `GET /vlans` , `GET /ip-addresses/by-address`
Moderate	A full-list read that grows with the fleet	Paging the results	`GET /devices` , `GET /cables` , `GET /flows/top`
Heavy	A fleet-wide pass or graph layout	90s compute-once cache	`GET /atlas/overview` , `GET /data-quality` , `GET /hotspots`
Very heavy	Batfish / AI model / live SNMP	Rate limits + partial answer if a subsystem is down	`POST /assistant/ask` , `GET /change-safety` , `POST /discovery/run/{id}/stage`

> **Failing gracefully caps cost.** The Batfish-backed endpoints return a partial answer (`available=false`) when the Batfish helper process is not running, so a missing or busy helper caps the cost instead of blocking the whole request. The AI layer works the same way: it bounds each turn (the four-step cap and the output ceiling) and falls back to the zero-spend stub when no model is configured.

CrossConnect by CybrIQ · AI API Cost Analysis · Engineering reference · 21 June 2026 · AI defaults, model id, token caps, and property names are read from the shipping build. The cost math (latency bands, endpoint tiers, and fleet-scaling) is a directional estimate to confirm under load. Model list prices are not quoted and change with your provider's price list. · contact_us@cybriq.io