AI Economy Playbook

2026

The hidden costs of LLMs, image, video, audio, and agents — and how to read your AI bill before it ships.

Mehmet Karakose
AI Economy Lab · aieconomylab.com

Contents

AI Economy Playbook 2026

Draft v1 — Introduction ~580 words, fits on one page in PDF layout.

The bill is bigger than the pricing page

You're paying more for AI than you think. The vendor pricing page told you `$2.50 per million input tokens` and `$10 per million output`. You did the math. Then the bill arrived, and it was 2× your forecast.

The pricing page wasn't lying. It just wasn't telling you the whole story.

This playbook is the rest of the story.

What gets buried

Four cost dimensions disappear from vendor pricing pages because they're inconvenient to advertise:

1. **Input ≠ output.** Output costs 3–5× input across every major provider. Generative tasks (chat, agents, structured generation) are dominated by output cost. Most budgets ignore this and end up 2–4× over.
2. **Caching changes the math.** Repeated system prompts drop to ~10% of normal cost with prompt caching — but the savings only land if you architect for it. Most teams pay full price because no one measured.
3. **Context burns silently.** "200K context window" doesn't mean "use 200K tokens cheaply." Long context is billed twice — once for the input, again for the harder reasoning the model has to do. Quality degrades too.
4. **Tokens are not language-neutral.** A Turkish prompt costs ~35% more than English on `o200k_base`. Thai, Persian, Arabic are 50–70% more expensive. If you're building for non-English markets, your unit economics are not what your spreadsheet says.

Add the multi-modal layer — image, video, audio, embeddings, fine-tuning, agent loops — and the gap between "what you paid" and "what you should have paid" gets wider every quarter.

Who this is for

You build with LLMs. Or you write the check that pays for someone who does. Either way, you've felt one of these:

- The bill is bigger than you projected, and you can't tell why.
- A teammate is choosing models, and you don't know if their reasoning is right.
- You're building for users in 5+ languages and your costs don't add up.
- You're past prototype, into production, and the unit economics matter.

If none of that lands — this playbook isn't for you. Read the calculator at aieconomylab.com instead; it's a 5-minute version.

How to read this

Each chapter ends with a **decision rule**: one or two sentences you can apply without re-reading the chapter. The body explains why. If you trust the rule and want to skim, skim — the rules are the whole product.

The chapters compound. By the end, you should be able to look at any AI feature and answer two questions:

- *What will this cost at scale?*
- *Where is the biggest saving I'm leaving on the table?*

Not perfectly. Within 30%, which is good enough to plan a roadmap or push back on a budget.

What's not here

- A tutorial on building with LLMs. Plenty of those exist.
- A case for or against any specific provider. The prices change every quarter; the principles don't.
- A guide to enterprise AI procurement. That's coming as a separate piece.

It's a working manual for the team building the feature, the founder writing the budget, and the engineer reviewing the pull request.

Mehmet Karakose · AI Economy Lab · 2026

Chapter 1.1 — Input ≠ Output

Draft v1 ~520 words. The first cost lie pricing pages tell.

The 3–5× rule

Across every major provider in 2026, output tokens cost 3–5× input tokens.

Model	Input ($/M)	Output ($/M)	Ratio
GPT-5	$1.25	$10.00	8×
GPT-5.4	$2.50	$15.00	6×
GPT-5.5	$5.00	$30.00	6×
GPT-5 mini	$0.25	$2.00	8×
Claude Opus 4.7	$15.00	$75.00	5×
Claude Sonnet 4.6	$3.00	$15.00	5×
Gemini 3.1 Pro	$2.00	$12.00	6×
Gemini 3 Flash	$0.50	$3.00	6×
Gemini 2.5 Pro	$1.25	$10.00	8×
o3	$2.00	$8.00	4×
o4-mini	$1.10	$4.40	4×
DeepSeek V4 Pro (regular)	$1.74	$3.48	2×
DeepSeek V4 Flash	$0.14	$0.28	2×
DeepSeek V3	$0.27	$1.10	4×

The exact numbers shift every quarter. The ratio doesn't.

Why output is expensive

Reading is parallel; writing is sequential. When you send 1M input tokens, the model processes them in one forward pass — heavy, but parallelizable across the GPU. When it generates output, every single token requires its own forward pass through the entire context. The compute is asymmetric. The price reflects it.

Worked example: a chatbot

Customer support bot. Per turn:

- System prompt (cached): 5K tokens
- Conversation history: 2K tokens
- User message: 200 tokens
- Bot response: 400 tokens

Looks input-heavy, right? 7.2K in, 400 out. Now run the math at typical mid-tier rates ($2.50 input / $10 output):

```
input:  7,200 × $2.50/M = $0.018
output:   400 × $10.00/M = $0.004
total:                   $0.022/turn
```

Output is **22% of cost on 5% of tokens.**

Scale to 100K turns/day: **~$2,200/day**, of which **$400 is the bot's mouth and $1,800 is the bot's ears.**

Now imagine you trim the system prompt by 50% (5K → 2.5K). You save $0.006/turn → **$600/day**. Compress the output by 50% (400 → 200)? You save $0.002/turn → $200/day.

Which optimization wins? Input. But which optimization gets shipped? In most teams: neither, because no one ran the per-component numbers.

The trap

Most cost forecasts use a single $/M-token number — sometimes input, sometimes output, sometimes a midpoint. All three are wrong. The right forecast splits input and output, applies them to your actual ratio, and only then sums.

The pattern that holds:

- **Output:input ratio < 0.1** (RAG, classification, summarization) → input dominates the bill. Optimize for input compression and prompt caching.
- **Output:input ratio > 0.3** (chat, agents, long-form generation, code synthesis) → output dominates. Optimize for tighter responses (`max_tokens`, structured output, smaller models on the generation side).
- **Output:input ratio > 1.0** (reasoning models, multi-step agent loops) → output crushes the bill. We'll get to that in Chapter 1.6.

Most builders look at their workload and *feel* it's input-heavy because the prompt is long. Tokens lie. Run the actual ratio.

Decision rule

> **Estimate the output:input ratio per use case before you pick a model. If >0.3, design for output reduction first — `max_tokens` caps, structured output, smaller models on the generation side. The input prompt is what you see; the output is where the money goes.**

Next: Chapter 1.2 — Cached Input. The 90% discount most teams pay full price for.

Chapter 1.2 — Cached Input

Draft v1 ~580 words. The 90% discount most teams pay full price for.

Two providers, two flavors

Provider	Cached read rate	Cache write rate	Min cacheable block
Anthropic	~10% of input	1.25× input	1,024 tokens
OpenAI	50% of input	(no extra)	None — auto
Google Gemini	~25% of input	(with explicit cache API)	32,768 tokens

Anthropic gives you the deepest discount but charges more per write. OpenAI is automatic and saves less. Gemini sits in the middle but requires a long minimum block.

The discount looks free. It isn't — you're paying upfront in cache write cost (Anthropic) or in API complexity (OpenAI).

Why caching works

Modern LLM inference does two things: process the input, then generate output. Processing the input requires computing key-value pairs through every layer of the model. If you sent the same prompt yesterday, those KV pairs are identical — the provider can store them and skip the computation. You pay storage rent (cache write fee) instead of compute rent (full input tokens).

It only pays off when the same prompt is reused enough times to amortize the write cost.

Worked example: support bot system prompt

Same chatbot from Chapter 1.1. System prompt is 5K tokens, used on every turn. 10K turns/day, Anthropic Sonnet 4.6.

Without caching:

```
5,000 × 10,000 × $3.00/M = $150/day on system prompt alone
```

With caching (write 1.25× = $3.75/M, read 10% = $0.30/M):

```
1× write:    5,000 × 1     × $3.75/M = $0.019
9,999× read: 5,000 × 9,999 × $0.30/M = $14.99
total:                                  $15/day
```

$135/day saved on a single component. $49K/year by adding a `cache_control` flag to the system message.

What to cache

The asymmetry: cache stable things, hit them often.

Cache: System prompts, role definitions, few-shot examples, tool definitions, structured output schemas, RAG context that doesn't change per query.

Don’t cache: User messages, agent step inputs, anything per-request, anything below the minimum block size.

The trap

Three common mistakes:

1. **Caching but not measuring.** Most teams turn caching on and never check cache hit rates. If your TTL is 5 minutes and traffic spreads across hours, half your “cached” reads pay the write cost again. Watch the `cache_read_input_tokens` field in API responses.
2. **Caching short prompts.** Anthropic’s minimum cacheable block is 1,024 tokens. Caching a 500-token system prompt does nothing. Gemini’s minimum is 32K — most apps never trigger it. Check the floor before assuming the discount.
3. **Cache invalidation drift.** Add one whitespace to the system prompt, the cache key changes, every customer pays a fresh write cost simultaneously. Lock the prompt or version it.

Cache vs context budget

Tempted to inflate the system prompt because “it’s cached anyway”? Don’t. Cached input is cheap, but it still passes through the model on every call. Quality degrades on long context (Chapter 1.3) regardless of price. Cache to save money on what you actually need, not as an excuse to keep more context.

Decision rule

> **Cache anything stable that’s referenced ≥5× per cache lifetime. Lock the cached block — every whitespace edit costs another full write across all clients. Measure cache hit rate weekly; if it’s below 80%, your TTL or your traffic shape is wrong.**

Next: Chapter 1.3 — Context Window Trap. Why “1M context” doesn’t mean “use 1M cheaply.”

Chapter 1.3 — Context Window Trap

Draft v1 ~620 words. “1M context” is insurance, not budget.

The pricing reality

Most providers charge a flat input rate up to a threshold, then a higher rate beyond it. As of 2026:

Model	Standard input	Long-context tier
Claude Sonnet 4.6	$3/M	$6/M (>200K)
Claude Opus 4.7	$15/M	$30/M (>200K)
Gemini 2.5 Pro	$1.25/M	$2.50/M (>128K)
GPT-5	$2.50/M	(single tier across full window)

The doubling at the long-context threshold isn’t an accident — it reflects the quadratic compute cost of long-range attention. Some providers price it linearly; some hide it in the standard rate. Either way, the cost is real.

The quality cost

Even when you can afford it, longer context degrades answer quality.

The “lost in the middle” effect (replicated across labs through 2024–2025): models attend strongly to the start and end of long inputs, weakly to the middle. A 100K-token input means anything in the 30K–70K range gets partial attention.

In practice: stuff 50 documents into context, ask the model to find the one mentioning X, and it’ll often miss documents in the middle even when the information is plainly there.

Worked example: docs Q&A bot

Internal documentation Q&A. Knowledge base: 500K tokens of company wiki. Two architectures.

A — stuff the docs in:

- Per query: 500K input + ~1K output on Claude Sonnet 4.6 long-context
- Cost: `500K × $6/M + 1K × $15/M = $3.015/query`
- 1K queries/day: **$3,015/day**, ~30s latency, mediocre accuracy.

B — RAG with retrieval:

- One-time embed: `500K × $0.10/M = $0.05`
- Per query: ~5K context (top-5 chunks) + 1K output = `5K × $3/M + 1K × $15/M = $0.030/query`
- 1K queries/day: **$30/day**, <2s latency, better accuracy.

100× cost reduction. Better quality. The “long context replaces RAG” claim only makes sense for one-off analyses where retrieval setup isn’t worth the engineering time.

When long context wins

It wins when:

- You can't predict what's needed. Single-shot complex reasoning where the whole document genuinely matters (legal review, academic paper analysis).
- Latency doesn't matter. Batch analysis, offline reports.
- You don't have a retrieval system yet. Cost of building RAG > cost of paying long-context premium for a few weeks.

It loses when:

- The data is structured, queryable, or has natural retrieval keys.
- You need real-time responses.
- The same context is reused across many queries (cache + RAG beats long context every time).

The trap

Two patterns to watch:

1. **Treating context size as budget.** "1M context" implies you should use 1M tokens. You shouldn't. Context windows are insurance — a high limit means you don't have to chunk perfectly. It doesn't mean stuffing more in helps.
2. **Caching the long context to "fix" the cost.** Caching does help on the price side (Chapter 1.2), but it doesn't fix the quality degradation. Long-context attention still suffers regardless of who paid for the input. Cache RAG context, not raw 100K-token doc dumps.

A cleaner mental model

Think of context like a desk: more space helps when you have lots to spread out, but a focused workspace gets more done. The tax on a cluttered desk is mistakes, not just rent.

Decision rule

> **Trim context to what's actually needed. Past 32K, use RAG instead of stuffing — the cost is 50–100× lower and quality is better. The big context window is for headroom, not for use. If you find yourself filling it, you're doing retrieval at the wrong layer.**

Next: Chapter 1.4 — Multi-Language Tokenization. Why a Turkish prompt costs 35% more.

Chapter 1.4 — Multi-Language Tokenization

Draft v1 ~600 words. The most ignored cost lens for non-English products.

The problem in one table

For 1,000 words of identical content, the input token count depends on the language **and** the tokenizer:

Language	o200k_base	claude	gemini
English	1.00×	1.00×	1.00×
Spanish	1.05×	1.05×	1.05×
German	1.15×	1.15×	1.15×
French	1.15×	1.15×	1.15×
Indonesian	1.25×	1.25×	1.20×
Japanese	1.25×	1.30×	1.25×
Russian	1.25×	1.30×	1.25×
Korean	1.30×	1.35×	1.30×
Polish	1.30×	1.30×	1.25×
Turkish	1.35×	1.40×	1.30×
Vietnamese	1.40×	1.40×	1.35×
Persian	1.50×	1.55×	1.50×
Arabic	1.45×	1.55×	1.45×
Hindi	1.55×	1.60×	1.55×
Thai	1.65×	1.70×	1.65×

Source: AI Economy Lab benchmarks on UDHR translations. Full 20-language table in Appendix B.

Why it happens

Tokenizers are trained on English-heavy corpora. They learn efficient encodings for English ("internationalization" → 2 tokens). Languages they've seen less get encoded character-by-character or in tiny fragments. Thai, Arabic, and Indic scripts also use complex Unicode that doesn't slot neatly into the tokenizer's vocabulary.

The differences across tokenizer families (`o200k_base` vs `claude` vs `gemini`) are small but real — Claude's tokenizer tends to be slightly worse on non-Latin scripts; Gemini's is slightly better in Eastern European languages.

The cost implication

If you build a multilingual product, your unit economics depend on the language mix.

Worked example. Same chatbot from Chapter 1.1, $0.022/turn baseline in English. Now run it in Turkey and Thailand:

Locale	Token multiplier	Cost/turn	$/100K turns/day
English	1.00×	$0.022	$2,200

Locale	Token multiplier	Cost/turn	$/100K turns/day
German	1.15×	$0.025	$2,500
Turkish	1.35×	$0.030	$3,000
Thai	1.65×	$0.036	$3,600

A Thai user costs you **64% more** than an English one for identical functionality. If your pricing is flat across locales, your margin in Thailand is 64% thinner. At meaningful volume, that's the difference between profitable expansion and silently subsidized growth.

Three responses

1. **Pre-translate to English.** Run inference in English, translate output back. Trade-off: latency goes up, translation quality matters, semantic content can drift.
2. **Use a tokenizer-aware model.** Some open-source models (e.g. fine-tuned Llama variants for specific languages) tokenize their target language more efficiently. Trade-off: you operate the model.
3. **Price tier by locale.** Charge more for high-overhead languages, or absorb the cost as part of localization. The least technical option, the most product-decision-heavy one.

Output overhead

The ratios above are for input. Output is similar but slightly different — generated text in non-English languages is also more tokens per word, so a "200-token Turkish answer" is actually fewer words than "200-token English answer." Most teams accidentally generate longer outputs in low-English-overlap languages because users expect similar verbosity. Cap `max_tokens` per locale.

The trap

Most pricing forecasts use English as the base case and assume "the model handles other languages too." It does — at 30–70% more cost. Audit your token mix per locale before you launch. The forecast spreadsheet that says "$0.02/user" turns into "$0.034/user" in Thailand the first week, without warning.

Decision rule

> **Multiply your input estimate by the tokenizer overhead for each target locale before budgeting. If you operate non-English at scale, either pre-translate to English for inference or make sure your pricing reflects the cost gap. Don't assume parity.**

Next: Chapter 1.5 — Batch API. Half the bill if you can wait 24 hours.

Chapter 1.5 — Batch API

Draft v1 ~480 words. Half off the bill, with a 24-hour catch.

The discount

Most major providers offer batch processing at half the synchronous rate:

Provider	Sync input	Batch input	SLA
OpenAI	$2.50/M	$1.25/M	24h
Anthropic	$3.00/M	$1.50/M	24h
Google	$1.25/M	$0.625/M	24h

Output rates are halved too. Same model, same quality. The provider runs your jobs when capacity is free.

What fits

Batch is built for jobs where latency doesn't matter:

- **Bulk classification.** 500K customer emails → topic tags.
- **Evaluation runs.** New model release, run your eval suite against the prompt set.
- **Offline generation.** Generate a million product descriptions once.
- **Backfills.** Re-process the last 6 months of transcripts because you added a new field.

What doesn't fit

- **Real-time chat.** User-facing latency is the constraint.
- **Agents.** Each step depends on the previous one's response.
- **RAG queries.** Users don't wait 24h for a search.
- **Anything that needs streaming output.**

Worked example

E-commerce migration: 500K SKUs need product descriptions. Each prompt ~300 input + ~150 output tokens, GPT-5 mini.

Sync:

```
input:  500K × 300 × $0.25/M = $37.50
output: 500K × 150 × $2.00/M = $150.00
total:  $187.50, runs in real-time
```

Batch:

```
input:  500K × 300 × $0.125/M = $18.75
output: 500K × 150 × $1.00/M  = $75.00
total:  $93.75, runs over up to 24h
```

$94 saved on one job. The bigger the migration, the bigger the saving. Backfills, evals, and offline generations all compound.

The friction

Batch APIs are a separate endpoint with a separate response format. You upload a JSONL file of requests, poll for completion, download results. Your existing prompt code doesn't directly port — you write batch-specific glue.

For most teams, this is a 1-day engineering investment that pays back within a month. For one-off analyses, it's not worth it; just pay sync rates.

The trap

Two common mistakes:

1. **Treating batch as "just run it overnight."** Batch SLA is 24h, but average completion is much less (often 1–3h). Plan for 24h, be pleasantly surprised by 3h. Don't promise users a 4h turnaround on a batch job.
2. **Mixing sync and batch in the same code path.** They have different rate limits, different error semantics, different response shapes. Build a clean abstraction layer or you'll regret it during the first batch failure at 3 AM.

Decision rule

> **Audit your AI calls quarterly. Anything that doesn't need real-time response moves to batch. The 50% saving compounds across every backfill, eval run, and offline generation — it's the cheapest optimization in this playbook.**

Next: Chapter 1.6 — Reasoning Token Tax. The cost dimension that didn't exist 18 months ago.

Chapter 1.6 — Reasoning Token Tax

Draft v1 ~620 words. The cost dimension that didn't exist 18 months ago.

What changed in 2024–2025

Reasoning models — OpenAI's o1/o3/o4 series, Gemini 2.5 Thinking, Claude Extended Thinking, DeepSeek V4 (thinking mode) — generate hidden reasoning tokens before producing the visible answer. You don't see them. You pay for them.

Model	Visible answer	Reasoning tokens (typical)	Output ratio
GPT-5	200	0	1×
o3 (medium effort)	200	1,500–5,000	8–25×
Gemini 2.5 Pro Thinking	200	800–3,000	4–15×
Claude Opus 4.7 (extended)	200	1,000–4,000	5–20×

All reasoning tokens are billed as **output**. So a 200-token visible response from o3 doesn't cost 200 output tokens — it costs 1,700–5,200.

The bill in practice

You build a support agent. Average visible answer: 400 tokens. Average daily volume: 100K turns.

Non-reasoning model (GPT-5, $10/M output):

```
400 × 100,000 × $10/M = $400/day on output
```

Reasoning model (o3-pro, $80/M output) with average 3,000 reasoning tokens:

```
(400 + 3,000) × 100,000 × $80/M = $27,200/day on output
```

68× the cost. Same visible response. The model is "smarter," but for a support task that mostly looks up FAQs, you're paying $26,800/day for reasoning that didn't change the answer.

When reasoning earns its cost

Reasoning models genuinely outperform on:

- Multi-step math / logic problems.
- Code synthesis with nontrivial constraints.
- Diagnostic / analytical tasks that need backtracking.
- Hard planning problems (agent task decomposition).

They don't earn their cost on:

- Customer support where most queries are FAQ-class.
- Summarization, classification, extraction.
- Conversational chat.
- Anything where the answer is mostly retrieval, not reasoning.

The benchmark wins are real. They just don't show up on every task you run.

Reasoning effort knobs

Major providers expose a way to cap depth:

- **OpenAI:** `reasoning_effort: "minimal" | "low" | "medium" | "high"`. Low cuts reasoning by ~5×; minimal effectively disables it.
- **Anthropic:** `thinking: { type: "enabled", budget_tokens: 2000 }`. Hard cap.
- **Gemini:** `thinking_config: { thinking_budget: 2048 }`. Hard cap.

Default settings often produce maximal reasoning. Cap aggressively, expand only when output quality measurably degrades.

The trap

Three patterns to watch:

1. **"o3 wins benchmarks, let's switch the whole product."** The wins are real on hard problems, but most of your traffic is easy problems. Run a per-task evaluation, not a global swap. The path that actually works: keep the cheaper model for the bulk, use a reasoning model only on the 5–15% of queries that benefit.
2. **No reasoning cap at all.** Default behavior on most reasoning models is "think as long as needed." For an agent loop with 5 reasoning steps, that's 5 × full reasoning trees — bills explode by 50×, not 5×. Always set a cap, even if it's high.
3. **Counting only visible output tokens in forecasts.** The most common bill-shock cause for teams running reasoning models in 2026. Usage dashboards combine reasoning + visible output; budget spreadsheets often miss the reasoning component entirely.

A pattern that works

Tier-1 model handles 85% of queries. Tier-2 reasoning model is invoked only on flagged queries (low confidence from tier-1, or specific task types where reasoning is known to help).

Example: a code-writing agent uses GPT-5 mini for boilerplate, escalates to o3 only when the user explicitly requests architecture-level design or when the initial response fails compile/test. Average cost stays close to GPT-5 mini's; the reasoning cost is paid only when it pays back.

Decision rule

> **Default to non-reasoning models. Switch to reasoning only for tasks where evals show meaningful quality lift, and always cap `reasoning_effort` or `thinking_budget`. The model that wins benchmarks may be 10–30× more expensive without making your product better.**

Next: Chapter 1.7 — Model selection in one page. The decision tree that ties Part 1 together.

Chapter 1.7 — Model Selection in One Page

Draft v1 ~620 words. The cheat sheet that ties Part 1 together.

The first six chapters were principles. This is the cheat sheet.

The selection axes

Five factors determine model class:

1. **Latency requirement** (real-time / batchable / async)
2. **Output type** (chat, structured, code, reasoning)
3. **Quality threshold** (good-enough / production / state-of-the-art)
4. **Volume** (low / medium / high)
5. **Language mix** (English-heavy / multilingual / non-English-dominant)

The decision tree

```
What's the dominant cost driver in this use case?

├─ INPUT-HEAVY (RAG, classification, summarization · output:input < 0.1)
│   ├─ High volume → cheapest model that hits quality bar
│   │   • GPT-5 mini, Claude Haiku 4.5, Gemini 3.1 Flash-Lite
│   ├─ Latency tolerable → batch API for 50% off
│   └─ Multi-language → check Chapter 1.4 ratios; tokenizer matters
│
├─ OUTPUT-HEAVY (chat, agents, code synthesis · output:input > 0.3)
│   ├─ Quality critical → top tier
│   │   • GPT-5, Claude Opus 4.7, Gemini 3.1 Pro
│   ├─ Mid quality OK → workhorse tier
│   │   • Claude Sonnet 4.6, Gemini 3 Flash, Gemini 2.5 Pro
│   └─ Cap max_tokens aggressively; structured output saves bytes
│
└─ REASONING-HEAVY (math, planning, multi-step logic)
    ├─ Genuinely hard problem → o3, Gemini 3.1 Pro, Claude Extended
    │   • Always set reasoning_effort cap (Chapter 1.6)
    └─ Looks hard but isn't → upgrade the prompt before upgrading the model
```

Quick comparison: where each provider currently wins

Use case	Recommended (May 2026)	Why
Customer support FAQ	GPT-5 mini, Gemini 3.1 Flash-Lite	Cheapest; good multilingual; clean structured output
Code generation (production)	Claude Sonnet 4.6, Gemini 3.1 Pro	Best instruction following; strong on long-codebase context
Bulk doc summarization	Gemini 3.1 Flash-Lite batch	Cheapest output rate; batch tier halves it again

Use case	Recommended (May 2026)	Why
Multi-step agent	GPT-5 (reasoning_effort: low)	Tool calls work cleanly; predictable cost
RAG over long docs	Claude Sonnet 4.6 + caching	Best long-context attention; cache the stable retrieval prompt
Hard math / diagnostics	o3, Gemini 3.1 Pro (effort: medium)	Reasoning is the actual product
Non-English chat (TR, ID, BR)	Claude Sonnet 4.6	Slightly better tokenizer for these locales

These recommendations age fast — re-validate quarterly. Prices and benchmark winners shift every few model releases; the principles in Part 1 don't.

What to do this week

1. **List the AI features in your product.** Be specific — "the chatbot" is too coarse; break it into "FAQ flow," "complaint flow," "agent flow."
2. **For each, mark its dominant cost driver** (input / output / reasoning).
3. **Apply the decision tree** above.
4. **Note which features are on the wrong tier.** Usually 1–2 per product. Common pattern: a flow that should be on a small fast model is running on a top-tier model "because we picked it for the hard cases."
5. **Run a small evaluation on the proposed swap.** 50–100 representative queries, blind comparison if possible. If quality holds, ship the swap.

A typical 5-feature product saves 30–50% by re-tiering 1–2 features. Full payback inside week 1.

What this doesn't tell you

The decision tree picks a class. The eval picks the model. **There is no model recommendation system that beats running your own task-specific evaluation** — partly because your data is unique, and partly because what "quality" means in your product can't be captured in a generic benchmark.

If you're picking a model based purely on benchmark leaderboards, you're picking a model optimized for the benchmark, not for you.

Decision rule

> **Pick the cheapest model that passes the use-case-specific eval — not the one that wins benchmarks. Eval per use case, not globally. Re-tier quarterly because the price-to-quality frontier shifts every model release.**

End of Part 1. Part 2 covers the multi-modal stack: image, video, audio, embeddings, fine-tuning, and agent loops.

Chapter 2.1 — Image Generation

Draft v1 ~580 words. Three pricing models, none of them comparable head-to-head.

Three pricing schemes, one bill

Image gen providers price differently, which makes apples-to-apples comparison painful:

Provider	Pricing unit	Standard	High quality
OpenAI DALL-E 3	per image	~$0.04 (1024×1024)	~$0.08 (HD)
OpenAI gpt-image-1	per token output	$0.01–0.07/image	$0.17/image
Google Imagen 3	per image	~$0.04 (1024×1024)	$0.06 (high)
Stable Diffusion 3.5 (Replicate)	per second of GPU	~$0.005–0.02/image	(longer steps)
Midjourney v7	per subscription seat	flat $30–120/mo	(with usage caps)
Flux Pro 1.1	per image	$0.04	$0.05

The variance between cheapest and most expensive for a single 1024×1024 image is ~40×. The variance in quality is much smaller. Pick the cheapest tier that passes your eval.

What drives cost inside one provider

Three knobs:

1. **Resolution.** 1024^2 is the default. 2048^2 costs ~4× because compute scales with pixel count. Most consumer use cases don't need 2048^2.
2. **Quality / steps.** "Standard" vs "HD" or "few-step" vs "many-step" inference is typically 2–4× different. The eye can tell on the right output (faces, text, fine detail), can't tell on most.
3. **Number of variations.** Some APIs charge per variation generated; some bake N variations into a single price. Read the per-call billing carefully.

Worked example: e-commerce product photos

You need to generate placeholder product images for 10,000 SKUs in an e-commerce migration.

Naive: high quality from scratch on DALL-E 3 HD:

```
10,000 × $0.08 = $800
```

Smarter: draft + upscale workflow:

```
Step 1: Generate at 512×512 standard quality (Flux 1.1 Pro):
  10,000 × $0.04 = $400
Step 2: Human review, mark ~30% as final-quality candidates:
  3,000 × $0.05 (HD upscale) = $150
Total: $550, ~30% saving, much better hit rate on the final 30%
```

The saving isn't just in dollars — it's in not wasting compute on rejects.

When quality genuinely matters

For these, pay for the best quality:

- Marketing hero images (single-use, brand-defining).
- User-facing illustrations in production.
- Anything with text or hands (these are the failure modes most providers still don't nail).
- Brand-consistent iterations (Flux and Midjourney have the strongest style transfer).

For these, default to medium:

- Internal mockups.
- Placeholder content during dev.
- Bulk catalog generation.
- Anything that's getting human-touched anyway.

The trap

Two patterns that quietly inflate the bill:

1. **Generating at production quality during exploration.** The first 50 prompts are exploration; you'll throw 80% of outputs away. Generate at standard. Only crank up quality when the prompt and concept are locked.
2. **Per-image pricing assumes one image per call.** Most APIs let you request `n=4` variations in a single call — but you pay for all 4 even if you only use one. If your workflow always picks the best of N, that's fine. If it doesn't, you're paying 4× for one usable output.

What to budget

Image gen for a typical SaaS feature (think: AI-generated thumbnails, in-product illustration):

Use case	Per-user/month estimate
Casual feature (1–5 images/user/month)	$0.10–0.40
Active feature (20–50 images/user/month)	$1.00–4.00
Bulk creator tool (200+ images/user/month)	$8.00+

If your subscription is $10/mo and your user generates 50 images, image gen alone is half your COGS. Price accordingly.

Decision rule

> **Default to medium quality. Reserve high quality for human-final outputs. Use a draft → review → upscale workflow on bulk jobs — the rejects don't deserve full-quality compute. Single-source one provider during exploration; lock in the cheapest that passes the eval before you scale.**

Next: Chapter 2.2 — Video Generation. The cost dimension that didn't exist 12 months ago.

Chapter 2.2 — Video Generation

Draft v1 ~560 words. The cost dimension that barely existed a year ago.

The pricing reality

Video models price per second of generated output. As of 2026:

Provider	Standard	High res	Notes
OpenAI Sora 2	$0.50–1.50/sec	$2–4/sec	1080p / 4K tiers
Google Veo 3	$0.50/sec	$0.75/sec	8s default clips
Runway Gen-4	$0.05–0.10/sec	$0.20/sec	Subscription credits
Pika 2.5	$0.05–0.15/sec	$0.30/sec	Flat per-clip on app
Luma Ray 2	$0.10/sec	$0.25/sec	

A 5-second clip ranges from **~$0.25 (cheapest tier) to ~$20 (top-tier 4K)**. An order of magnitude. The output of cheaper providers is usually identifiable on careful inspection but invisible in a feed.

What drives cost

Four knobs, in order of impact:

1. **Duration.** Linear. 10 seconds = 2× the cost of 5 seconds.
2. **Resolution.** 4K is typically 3–5× the cost of 1080p. Most distribution platforms compress to 1080p anyway.
3. **Model tier.** Top-tier (Sora 2, Veo 3) vs. mid-tier (Runway, Pika) is 5–10×. Top-tier wins on physics, character coherence, and complex prompts.
4. **Frames per second.** 24 fps default, 30/60 fps premium. Most viewers can't tell.

Worked example: marketing video

Marketing campaign: 30 different 4-second product clips. Internal review picks the 5 best for ad rotation.

Naive: 30 × Sora 2 high-res from scratch:

```
30 × 4s × $4/sec = $480
```

Smart: drafts on cheaper tier → upscale winners:

```
Step 1: 30 × 4s on Pika 2.5 ($0.10/sec) = $12
Step 2: Review, pick top 5
Step 3: 5 × 4s on Sora 2 high-res ($4/sec) = $80
Total: $92, ~80% saving
```

The 25 clips you didn't upscale are perfectly fine for internal review — most don't survive selection anyway, so they don't deserve top-tier compute.

What top-tier is actually for

The top-tier video models earn their cost on:

- **Physics** (water, fabric, hair, faces in motion).
- **Character consistency across cuts.**
- **Complex camera motion** (dolly, orbit, focus pull).
- **Multi-second narrative coherence.**

Mid-tier is fine for:

- Stock B-roll (sky, traffic, ocean, product spins).
- Abstract / motion graphics-adjacent visuals.
- Internal previz / storyboarding.
- Any clip <2 seconds.

The friction nobody talks about

Most video gen takes **30 seconds to 5 minutes** per clip. That's not just a UX problem — it's a billing risk. If you build a feature that auto-regenerates on user edits, a single user can burn $50 in 10 minutes by clicking "regenerate" four times.

Cap it at the application layer. Per-user-per-day video generation budgets are non-negotiable for any consumer product.

The trap

Three patterns:

1. **Treating video like image.** "Just generate 100 of these." A hundred 5-second clips on Sora 2 is **$2,000**. Image gen at scale is annoying; video gen at scale is bankrupt.
2. **Default high-resolution everywhere.** 4K only matters on cinema and high-end displays. For social, web, mobile — 1080p indistinguishable, 30–60% cheaper.
3. **No usage caps in consumer features.** Users will mash "regenerate" until they get the clip they want. Without server-side caps, the bill is unbounded.

What to budget

Use case	Per-month estimate
Internal team using video gen ad-hoc	$50–200/mo
Per-customer feature (low usage)	$0.20–1.00/user-month
Per-customer feature (active usage)	$5–20/user-month
Bulk pipeline (catalog automation)	$1,000–10,000/mo

If your subscription is $20/mo and average user generates 10 video clips, video gen is your entire margin.

Decision rule

> **Draft on a cheap tier, upscale only winners. Cap resolution at the resolution your users actually consume (1080p for everything except cinema). Always set per-user-per-day budgets in app code — the regenerate button is a bill amplifier.**

Next: Chapter 2.3 — Audio. TTS, STT, and the cost asymmetry between them.

Chapter 2.3 — Audio (TTS, STT, Music)

Draft v1 ~580 words. Three audio modalities, three different cost models.

Three modalities, three pricing schemes

Modality	Provider	Pricing
TTS (Text → Speech)	ElevenLabs	$0.18/1K chars (Pro), $0.30/1K chars (Eleven v3)
	OpenAI gpt-4o-mini-tts	$0.015/1K input chars + $0.012/1K output
	OpenAI gpt-realtime	$32/M output audio tokens
	Google Chirp 3 HD	$0.013/1K chars
STT (Speech → Text)	Whisper API	$0.006/min
	Deepgram Nova-3	$0.0043/min
	OpenAI gpt-realtime	$5/M input audio tokens
	AssemblyAI	$0.012/min
Music gen	Suno v5	~$0.05–0.10/sec (subscription credits)
	Udio	similar tiered

The biggest cost gap: **TTS is roughly 20–50× more expensive than STT** for equivalent durations. TTS at scale is a real budget line; STT usually rounds to nothing.

TTS — where the bill explodes

A 5-minute audio clip is roughly 3,500–5,000 characters. Worked example for a podcast generator that produces 100 episodes/day, each 30 minutes:

ElevenLabs Pro:

```
30 min × ~6,000 chars × 100 = 18M chars/day
18M / 1K × $0.18 = $3,240/day
```

OpenAI gpt-4o-mini-tts:

```
18M / 1K × $0.012 (output) ≈ $216/day
```

15× cheaper. The trade-off is voice quality — ElevenLabs is meaningfully better on emotion, accents, and naturalness for human-facing content. OpenAI's TTS is fine for explanatory voiceover, accessibility narration, internal training material.

STT — almost free at any scale

Same podcast generator, but transcribing user-uploaded recordings:

```
30 min × 100 = 3,000 minutes/day
Whisper API: 3,000 × $0.006 = $18/day
Deepgram Nova-3: 3,000 × $0.0043 = $13/day
```

For most products, STT cost is so small you don't budget for it explicitly. The exception: real-time STT in agent products (gpt-realtime) where input audio is billed in tokens — costs jump 5–10× because audio token rates are far higher than minute-based.

Caching TTS

TTS is one of the few modalities where output caching is straightforward: identical text → identical audio. If your product generates the same phrase repeatedly ("welcome to AI Economy Lab," meeting reminders, navigation prompts), cache audio outputs at the application layer.

Most TTS providers don't offer native caching the way LLM providers do. **You implement the cache yourself** — hash the text, store the audio file in S3/R2, serve from cache on hit. For high-traffic products with phrase reuse, hit rates of 60–90% are realistic.

Music — the costliest per-second modality

Music generation pricing is the highest per-second of any audio modality, often $0.05–0.10/sec generated. A 3-minute track is $9–18. Most products use music gen sparingly — background loops, intros — not as the core feature. If music is core, factor in subscription tiers (Suno Pro, Udio Pro) which cap per-month generation at flat rates.

The trap

Three patterns:

1. **Premium TTS by default.** Most products use ElevenLabs because it's the household name. Half the time, OpenAI gpt-4o-mini-tts at 1/15th the cost is fine — the difference is audible only in side-by-side comparison.
2. **No caching for repeated phrases.** Notification, reminder, and onboarding prompts repeat across users. A naive implementation re-generates audio for every user. Cache by text hash; hit rates are typically high.
3. **Real-time audio agents without a budget cap.** `gpt-realtime` is convenient but the audio-token rate is steep. A 10-minute conversation is $1–3. Cap per-session duration in app code.

Decision rule

> **Use STT generously — it's nearly free. Cache TTS aggressively for any text that repeats across users. Default to mid-tier TTS (OpenAI, Google Chirp) and reserve premium (ElevenLabs) for human-facing brand-quality content. Real-time audio agents need per-session budget caps in app code, not just provider rate limits.**

Next: Chapter 2.4 — Embeddings. Cheap per call, expensive per migration.

Chapter 2.4 — Embeddings

Draft v1 ~480 words. Cheap per call. Expensive per migration. Decided rarely.

The pricing reality

Embedding models are the cheapest billable AI primitive:

Provider	Model	Price
OpenAI	text-embedding-3-small	$0.020/M tokens
OpenAI	text-embedding-3-large	$0.130/M tokens
Cohere	embed-v4	$0.10/M tokens
Voyage	voyage-3	$0.06/M tokens
Google	gemini-embedding-001	$0.025/M tokens
Open-source (Together, Replicate)	nomic-embed-v2	$0.005–0.01/M tokens

Most products spend more on coffee than on embeddings. The cost is real only at very large indexes (hundreds of millions of documents) or in re-embedding scenarios.

What actually costs money

The embedding call itself is cheap. Three things blow up:

1. **Re-indexing.** When you change your embedding model, you re-embed everything. A 100M-token corpus on text-embedding-3-large is `100M × $0.130/M = $13`. A 10B-token enterprise corpus is $1,300. Annoying but not catastrophic.
2. **Vector database costs.** Storing the vectors costs more than generating them, especially at scale. A 10M-document index in Pinecone is ~$70–200/month. Storage and query costs dwarf the one-time embedding cost.
3. **Query-time embedding.** Every search query gets embedded. If you serve 1M queries/day at 50 tokens average, that's `1M × 50 × $0.020/M = $1/day` on small embeddings — trivial. With large embeddings (3-large, voyage-3) it's still under $10/day for 1M queries. Don't over-optimize this.

The real decision

Picking an embedding model is a long-term commitment. You can't easily switch without re-indexing. So the trade-offs:

- **Quality on your domain.** Run an eval on your actual retrieval task, not a benchmark. Domain-tuned embeddings often beat general-purpose ones, especially for specialized content (legal, medical, code).
- **Vector dimension.** 256-dim, 768-dim, 1536-dim, 3072-dim. Smaller dimensions = cheaper storage in vector DB, faster queries, slightly lower recall.
- **Multilingual.** If you index non-English content, check the model's training corpus. text-embedding-3-large and Cohere embed-v4 both support multilingual; smaller models can be English-only in practice.
- **Vendor lock-in.** Switching means re-indexing the whole corpus. That's the dominant migration cost, not the dollar amount.

Worked example: knowledge base

Internal company wiki. 500K documents, ~500 tokens each = 250M tokens.

One-time embedding (text-embedding-3-large):

```
250M × $0.130/M = $32.50
```

Storage (Pinecone serverless, ~3KB/vector):

```
500K × 3KB = 1.5 GB → ~$30/month
```

Query embedding (10K queries/day):

```
10K × 50 tokens × $0.130/M = $0.07/day → $2/month
```

Total: $32 setup + $32/month ongoing. The recurring cost is dominated by storage, not embedding generation.

The trap

Re-embedding without thinking. Every time a new “better” embedding model launches, the temptation is to swap. Don’t, unless:

- The new model has measurably better recall on your eval set.
- The migration cost (engineering + dollars) is amortized within 12 months.
- You’re not chasing benchmark gains that don’t show up in your product.

Most retrieval quality improvements come from better chunking, hybrid search, and reranking — not better embeddings.

Decision rule

> **Pin one embedding model per index — migration is a project, not a button click. Don’t switch unless your eval shows measurable retrieval quality lift on your actual data. Spend optimization energy on chunking, hybrid search, and reranking before changing the embedding model.**

Next: Chapter 2.5 — Fine-Tuning. The amortization curve nobody runs.

Chapter 2.5 — Fine-Tuning

Draft v1 ~580 words. The amortization curve nobody runs before deciding.

Two costs, both real

Fine-tuning has a one-time training cost and a recurring inference premium. Most teams price the first and forget the second.

Training (one-time)

Provider	Model class	Training rate
OpenAI	gpt-5-mini fine-tune	$25/M tokens
OpenAI	gpt-5 fine-tune	$90/M tokens
Anthropic	Claude Haiku 4.5 fine-tune (custom)	enterprise pricing
Together	Llama-class (LoRA)	$1.20/M tokens
Fireworks	Llama, Mistral fine-tune	$0.50/M tokens

Open-source-base fine-tuning (Together, Fireworks) is 20–50× cheaper for training than closed-model fine-tuning, but inference economics differ — see below.

Inference (recurring)

This is where the math gets interesting:

Tier	Base inference	Fine-tuned inference	Premium
OpenAI gpt-5-mini	$0.25/M in, $2/M out	$0.50/M in, $4/M out	2×
Together Llama-class	$0.20/M in, $0.20/M out	same as base	none
Fireworks fine-tune	$0.20/M in, $0.20/M out	small premium	minimal

OpenAI fine-tuned models cost **2× the base model in inference**. Forever. This is where most fine-tune decisions go wrong — the calculation only includes training cost.

The amortization curve

When does fine-tuning beat just-engineering-the-prompt?

Worked example: classification task. You have a 5K-token prompt with few-shot examples that gets 92% accuracy. Fine-tuning gives 95% with a 200-token prompt.

Without fine-tuning (gpt-5-mini base):

```
Per call: 5K × $0.25/M + 200 × $2/M = $0.0017
At 1M calls/year: $1,700
```

With fine-tuning:

```
Training: 10M tokens × $25/M = $250 (one-time)
Per call: 200 × $0.50/M + 200 × $4/M = $0.0009
At 1M calls/year: $900 + $250 = $1,150
```

Fine-tuning saves $550/year. **Break-even is around 350K calls.** Below that, just optimize the prompt.

If quality lift were smaller (92% → 93%), the calculation flips — the prompt-engineering route stays cheaper because the dropped accuracy doesn't justify the inference premium.

When fine-tuning earns its cost

Three patterns where it pays back:

1. **Volume × prompt-length savings.** High-volume product where fine-tuning lets you drop a long few-shot prompt to a short instruction. The shorter prompt compounds across millions of calls.
2. **Style / tone consistency.** Replicating a brand voice or specific domain register that's hard to describe in a prompt.
3. **Latency.** Fine-tuned models often respond faster because the prompt is shorter; lower TTFT.

It usually doesn't pay back for:

- Knowledge tasks. The knowledge can sit in RAG more cheaply.
- Reasoning tasks. Reasoning models are often better than fine-tuned non-reasoning models for hard logic.
- Tasks under 100K calls/year. Train cost > savings.

What about open-source-base fine-tuning?

Together and Fireworks fine-tuning of Llama / Mistral / Qwen is dramatically cheaper for training and has no inference premium. The trade-off:

- **Quality.** Llama-3.x and Qwen-3 base quality is good but typically below GPT-5 / Claude Sonnet on hard tasks.
- **Operational lift.** You're now responsible for evals, prompt versioning, and model version drift.

For commodity tasks (extraction, classification, formatting), open-source fine-tuning is a great fit. For anything user-facing where quality matters at the margin, the closed-model premium is usually worth paying.

The trap

The biggest mistake: jumping to fine-tuning before exhausting cheaper options.

Order of operations for cost optimization:

1. Better prompt (free).
2. Few-shot examples (free).
3. Cached input (Chapter 1.2 — 90% off).
4. RAG instead of stuffing context.
5. Smaller model with better prompt (often beats bigger model with worse prompt).
6. **Then** fine-tuning.

Most teams jump to step 6 because it sounds technical. Steps 1–5 usually solve the problem.

Decision rule

> **Don't fine-tune until you've exhausted prompt engineering, caching, and RAG. When you do, run the amortization math: training cost + inference premium × annual calls vs. base model cost. If break-even is more than 12 months out, the answer is "not yet." For commodity tasks**

at high volume, open-source fine-tunes (Together, Fireworks) often beat closed-model fine-tunes once total cost is computed.

Next: Chapter 2.6 — Agent Loops. The token explosion no one sees coming.

Chapter 2.6 — Agent Loops

Draft v1 ~600 words. The token explosion no one sees coming.

The compounding bill

A single LLM call has a known cost. An agent loop — where the model calls a tool, gets a result, decides what to do next, calls another tool — does **not** have a known cost. It has a worst-case bound and an average that's 10–30× the single-call equivalent.

Why: every loop iteration adds the previous step's input, output, tool result, and reasoning to the next call's context. Context grows linearly with steps; cost grows roughly quadratically because each step pays for the accumulated context.

Worked example

Customer support agent that can call: `search_kb`, `lookup_order`, `escalate_to_human`. Average task takes 3 tool calls.

Single-shot (no agent loop, RAG-style):

```
Input:  ~3K (system + user query + retrieved context)
Output: ~300 tokens
Cost on Claude Sonnet 4.6: 3K × $3/M + 300 × $15/M = $0.0135
```

Agent loop (3 tool calls):

```
Step 1: Read user message, decide to search_kb
  Input: 3K, Output: 200 (reasoning + tool call)

Step 2: Get search result, decide to lookup_order
  Input: 3K + 200 + 1K (search result) = 4.2K
  Output: 200

Step 3: Get order data, formulate response
  Input: 4.2K + 200 + 800 (order data) = 5.2K
  Output: 400 (final answer)

Total: input 12.4K, output 800
Cost: 12.4K × $3/M + 800 × $15/M = $0.0492
```

3.6× the cost of single-shot for the same outcome. And this is a "small" agent. A 5-step agent typically costs 8–15× single-shot. With reasoning models in the loop (Chapter 1.6), it's 30–80×.

The reasoning + tools amplifier

Combine an agent loop with reasoning tokens and the math gets ugly fast.

5-step agent on o3 with average 2,000 reasoning tokens per step:

```
Each step: input grows ~1-2K, output ~2,200 (reasoning + answer + tool call)
After 5 steps: input cumulative ~25K, output cumulative ~11K
Cost: 25K × $10/M (input) + 11K × $40/M (output) = $0.69 per task
```

Compared to a single-shot non-reasoning model at $0.013, that's **53× the cost.** If the agent quality lift is 5–10% (typical for genuinely hard tasks), is it worth it? Maybe. For most tasks, it isn't.

Caps that work

Three layers of protection, none optional:

1. **Hard step cap.** `max_iterations: 5` on the agent. If the agent hasn't finished in 5 steps, it failed; return the partial answer or escalate to a human. Without this cap, the agent is your bill's only limit.
2. **Token budget cap.** Track cumulative tokens across the loop. Abort if total > N (e.g. 50K). This catches the "agent stuck thinking" failure mode that step caps miss.
3. **Per-task dollar cap.** Multiply tokens × pricing. If a single user request crosses, say, $0.50 in agent loop cost, abort. This is the only cap that protects against truly pathological loops.

What runs as agents and shouldn't

Three patterns that look like agent tasks but don't need the loop:

1. **Sequential pipelines with deterministic structure.** Step 1 always calls tool A, step 2 always calls tool B, etc. This is workflow code, not an agent. Hardcode it; it's faster, cheaper, more reliable.
2. **Single-tool tasks.** Agent that "decides" whether to search_kb, but always does. Cut the agent. Just call the tool.
3. **Decision trees with 2-3 branches.** Often cheaper to have a small classifier model pick the branch and route deterministically.

When agents are worth it

Real agent territory:

- **Open-ended exploratory tasks.** Multi-source research, complex troubleshooting, "figure out why X is failing."
- **Tool composition that depends on intermediate results.** "Look up the customer, find the relevant order based on what they said, suggest a fix from the KB based on order status."
- **Tasks where the right next step genuinely depends on the previous result.**

For these, agents are right. For everything else — pipeline or workflow.

Decision rule

> **Cap step count, total token budget, AND per-task dollar cost. Profile every agent flow's actual average and p99 cost; expect p99 to be 5–10× the average. If a task can be expressed as a deterministic pipeline, write it as one — agents are for branching exploration, not for control flow you could have written.**

End of Part 2. Part 3 is strategic: how to forecast, how to track, how to evaluate vendors.

Chapter 3.1 — Forecasting AI Costs

Draft v1 ~620 words. Three numbers, three scenarios, one budget.

The three numbers

Every AI feature's cost forecast reduces to three numbers:

1. **Tokens per event.** What does one call to this feature cost in tokens? Input + output, separately.
2. **Events per user per day.** How many calls per active user, per day, on average?
3. **Growth rate.** How is the user base growing month over month?

Multiply: `cost = tokens × users × events/user/day × 30 × $/M`. Project forward with growth.

It looks simple. It usually goes wrong because the numbers are guessed, not measured.

Three scenarios, not one

The single biggest forecasting mistake is building a single point estimate. **Always forecast three scenarios:**

- **Best case.** Light usage. Token-efficient prompts. High cache hit rate. Power users behave like average users.
- **Expected case.** Today's measured numbers, projected forward at current growth.
- **Worst case.** Heavy users dominate. Cache hit rate is low. Output runs at the upper end of the distribution. Output:input ratio is 30% above measured.

The worst case is usually 2–3× the best case. **Budget the worst case.** Plan around the expected case. Be pleasantly surprised if you land on the best.

Worked example: AI chat feature

You're launching an AI chat feature. Beta data:

- 300 input + 250 output tokens per turn
- Active users: 500
- Average 10 turns per user per day
- GPT-5 mini: $0.25/M input, $2/M output

Expected case:

```
Per turn: 300 × $0.25/M + 250 × $2/M = $0.000575
Per user/day: 10 × $0.000575 = $0.00575
Monthly: 500 × 30 × $0.00575 = $86
```

Worst case (heavy users 3× the average, cache hit drops, output 1.5× expected):

```
Per turn: ~$0.0011 (output growth dominant)
Per user/day: ~$0.022
Monthly: $330
```

Six months out at 50% MoM growth (3,800 users in expected case):

```
Expected: $660/mo
Worst: $2,500/mo
```

That's the budget conversation. "We'll spend $660/month on AI" is wrong; "we'll spend $660–2,500/month, depending on usage shape" is the truth.

The forecasting mistakes

Five patterns to watch:

1. **Treating tokens as fixed.** Token counts vary by user. Heavy users have longer conversations and more turns. Forecast on p75 or p90 usage, not the mean.
2. **Ignoring cache TTL behavior.** Cached costs only land if traffic shape supports the cache. A burst at 9 AM might cache; spread traffic across 12 hours might not.
3. **Skipping output:input ratio.** Forecasting input only, then "rough output multiplier." Output cost dominates output-heavy features (Chapter 1.1) — forecast both axes.
4. **Forgetting reasoning tokens.** If the feature uses reasoning models, expected output is 5–20× visible output (Chapter 1.6). Most spreadsheets miss this entirely.
5. **Linear growth assumption.** Real growth is exponential or hockey-stick. A linear forecast wildly under-predicts the budget at month 6 if growth is 30%+/month.

A useful pattern: cost-per-feature tags

Most forecasting tools let you label calls with metadata. Tag every API call with `feature_id` at the application layer. At month-end, aggregate by feature: which features are growing in cost, which are flat, which are declining? Outliers get re-tiered (Chapter 1.7).

This is the foundation for everything in Chapter 3.2 (tracking). If you don't tag now, you can't track later.

Re-forecast cadence

Monthly. Forecasts go stale fast: usage patterns change, model prices shift, growth rates surprise.

Set a recurring 30-minute review:

- Pull last month's actual spend by feature.
- Compare to last month's forecast.
- Note where you were off by >20% and why.
- Update next 3 months' forecasts.

The team that re-forecasts monthly is never surprised by the bill. The team that forecasts once and walks away is.

Decision rule

> **Forecast in three scenarios (best / expected / worst), budget the worst, plan the expected. Tag every API call with feature_id from day one. Re-forecast monthly and track variance per feature. The bill that surprises you is the bill you didn't measure.**

Next: Chapter 3.2 — Tracking actual vs estimated. The dashboards most teams don't build.

Chapter 3.2 — Tracking Actual vs Estimated

Draft v1 ~580 words. The dashboard most teams haven't built and need.

What vendor dashboards don't tell you

OpenAI, Anthropic, and Google all give you a usage dashboard. They show:

- Total spend per day / week / month
- Spend by API key
- Spend by model

They don't show:

- Spend per **product feature** (your customer support bot vs. your code-gen feature)
- Spend per **customer** (which 5% of users are 80% of the bill?)
- Spend per **task type** (RAG queries vs. agent runs vs. simple Q&A)
- **Cache hit rates** in a way you can drill into

Without this, you can't optimize. The bill is one number; your product has 5–20 distinct AI features. You need to attribute that number to features.

The minimum viable cost dashboard

You need three columns:

1. **Date / hour** (time bucket)
2. **Feature ID** (what part of your product made this call)
3. **Cost** (input tokens × input rate + output tokens × output rate)

Tag every API call with a `feature_id` at the application layer. Capture the response usage object (`prompt_tokens`, `completion_tokens`, plus `cached_input_tokens` and `reasoning_tokens` where applicable). Compute cost on the way out. Log to a database, BigQuery, or even a flat file with daily aggregation.

This is **half a day's engineering**. The first month of data will surprise you.

What you'll find

Common patterns when teams first build this:

- One feature is 60% of the bill. You expected it to be 30%.
- One customer (or one cron job) is 20% of the bill. You didn't know.
- Cache hit rate is 40% on a feature that should be 95%. Something broke quietly.
- Reasoning token count is 10× what you forecasted. You're paying for thinking nobody asked for.
- Output:input ratio per feature differs by 5×. Some features should be on smaller models.

Each of these is a specific optimization with a specific dollar saving. None of them is visible in the vendor dashboard.

The first audit

Run this once you have a month of data:

1. **Top 5 features by cost.** Are they the top 5 by user value? If not, something is mis-tiered.
2. **Cost per task** (by feature). Calculate average and p99. p99 / average tells you how much variance the worst-case tasks contribute.
3. **Cache hit rate** (where applicable). Below 80%? Investigate.
4. **Output:input ratio** per feature. >0.3? You're on the wrong side of Chapter 1.1.
5. **Reasoning token share** (if using reasoning models). Plot reasoning tokens / visible output tokens. Anomalies stand out.

A typical SaaS finds 30–50% savings in this first audit. The team that's never run it is, on average, 30–50% over-spending.

Tooling options

Three paths, depending on team size:

- **Self-built** (recommended for small teams). Logging middleware in your AI calls, Postgres or BigQuery for storage, Metabase / Grafana / built-in Astro page for visualization. ~1 day of work.
- **Off-the-shelf observability.** Helicone, Langfuse, OpenLLMetry, Portkey. Cost: $50–500/month. Faster to set up; tags depend on what they support.
- **AI Economy Lab** (coming): hosted dashboard that combines actual spend with the calculator's optimal-cost model — flagging the gap automatically. Sign up at aieconomylab.com to be notified on launch.

For a 1-engineer-on-AI startup, self-built is fine. For 5+ engineers shipping AI features in parallel, off-the-shelf saves coordination overhead.

The discipline

Three habits separate teams that stay on budget from teams that don't:

1. **Tag at write time.** Add the `feature_id` when you write the AI call. Backfilling tags is painful.
2. **Review weekly.** 15 minutes. Look at the dashboard. Anything spike? Anything drop?
3. **Compare to forecast monthly.** Variance > 20% means either the feature changed, the user behavior changed, or your forecast was wrong. Investigate.

Decision rule

> **Tag every AI call with feature_id and capture token usage from the response. Build a minimum dashboard (date × feature × cost) within the first month of any AI feature. Run the cost audit at month 2 — most teams find 30–50% savings on the first pass. The team that doesn't track is the team that gets surprised.**

Next: Chapter 3.3 — Vendor evaluation. Not just price; lock-in, volatility, stability.

Chapter 3.3 — Vendor Evaluation Framework

Draft v1 ~620 words. Five dimensions, no leaderboards.

Why benchmarks aren't enough

LLM leaderboards rank models on standardized evals. Useful for research; misleading for production decisions. The model that scores 0.5 points higher on MMLU may cost 3× more, miss your specific use case, or come from a vendor that drops support next quarter.

Production vendor evaluation is five-dimensional, not one.

The five dimensions

1. $/quality (on your task)

Run your own eval. 50–100 prompts representative of real production traffic. Score outputs blind, ideally by 2 reviewers. Compute average cost per output that passes the bar.

Cheapest model that passes is the right choice — not the model with the highest absolute score.

2. $/latency

For user-facing features, latency is a feature. A 4-second response feels worse than a 1-second response, even if the 4-second response is "smarter." Measure p50 and p95 latency on your typical input length.

The fastest model that passes the quality bar wins for user-facing real-time tasks. For batch / async, latency doesn't matter.

3. Vendor stability

Some vendors are 18 months old. Some are 8 years old. Some are funded $50M+; some are bootstrapped. None of this is a price you pay this month, but **the cost of switching when a vendor folds or pivots is enormous** — re-architecting prompts, re-running evals, possibly re-fine-tuning.

Score: how confident are you the vendor will exist in 24 months? Adjust accordingly.

4. Pricing volatility

How often has this model's price changed in the last 12 months? Some vendors have re-priced 3× in a year (always downward, but with feature gating). Others have been flat. Volatility matters for budgeting — a 30% price drop is good, but it can also signal aggressive market positioning that might reverse.

OpenAI, Anthropic, Google: relatively stable, with quarterly model updates. Newer entrants: volatile. Open-source-base providers (Together, Fireworks): stable per model, but model SKUs come and go.

5. Lock-in cost

If you needed to switch from this vendor tomorrow, what would it cost?

- **Prompts.** Models respond differently to the same prompt. A switch means a re-prompt-engineering project.
- **Fine-tunes.** Lost on switch unless you're using LoRA / open-source-base fine-tunes you control.
- **Caching.** Cache state is per-vendor.
- **Custom features.** Function-calling syntax, structured output schemas, tool definitions all differ.

A “low lock-in” vendor is one where switching costs you a week. “High lock-in” is one where switching costs you a quarter.

Worked comparison

Use case: customer support bot, English + Turkish + Indonesian, 200K turns/day.

Vendor / Model	$/quality	$/latency	Stability	Volatility	Lock-in	Net
OpenAI GPT-5 mini	High	High	High	Low	Low	**Strong default**
Anthropic Claude Sonnet 4.6	High	Mid	High	Low	Mid	Strong, slight FT lock
Google Gemini 2.5 Flash	High	High	High	Mid	Mid	Strong, schema risk
Together fine-tuned Llama	Mid	High	Mid	Low	Low	Cheap, ops cost
Smaller frontier vendor	TBD	TBD	Low	High	Mid	Skip unless niche edge

For this use case, the top 3 are interchangeable. Pick two for a primary + secondary; route by feature based on which one passes the per-feature eval better.

The two-vendor rule

For any AI use case spending >$5K/month, run two vendors in production:

1. Failover for outages (real ones happen multiple times per quarter, even with frontier providers).
2. Price negotiation leverage.
3. Resilience when a model deprecates or changes behavior.
4. Real-world A/B comparison data, ongoing.

The operational cost of a second vendor is real (extra API key management, response parsing, eval) but it’s the difference between “we have options” and “we’re hostage to one provider’s pricing.”

Re-evaluation cadence

Quarterly. Major model releases (~2–3 per year per vendor) shift the competitive picture. Re-run the eval. Check the dashboard. Decide whether to swap.

Decision rule

> **Score vendors on five dimensions, not one. Use 2 vendors per critical use case above $5K/month — failover, leverage, resilience. Re-evaluate quarterly when model lineups change. Lock-in cost is the dimension everyone underestimates and the one that determines whether the contract you sign today still serves you in 18 months.**

End of Part 3. Closing in the next file.

Closing

Draft v1 ~440 words. The playbook in 13 sentences, plus what to do next.

The 13 decision rules

1. **Estimate output:input ratio per use case.** If >0.3, design for output reduction first.
2. **Cache anything stable referenced ≥5× per cache lifetime.** Lock the cached block.
3. **Trim context to what's needed.** Past 32K, use RAG over stuffing.
4. **Multiply input estimates by tokenizer overhead per locale.** Don't assume parity.
5. **Audit for batch.** Anything not real-time moves to batch for 50% off.
6. **Default to non-reasoning models.** Cap effort when you do use them.
7. **Pick the cheapest model that passes the eval.** Re-tier quarterly.
8. **Image gen: draft on cheap, upscale only winners.** Cap resolution to what users actually consume.
9. **Video gen: hard caps on duration, resolution, and per-user-per-day spend.**
10. **Audio: STT generously, TTS with caching, real-time agents with session caps.**
11. **Don't fine-tune until you've exhausted prompts, caching, and RAG.** Run the amortization math.
12. **Agent loops: cap step count, total tokens, AND per-task dollar cost.**
13. **Forecast in three scenarios. Tag every call. Re-evaluate vendors quarterly.**

If you remember nothing else, the through-line is: **measure the actual numbers per feature**, then apply the rule that fits.

What to do this week

Pick one. Just one.

- **Run the cost audit.** If you don't have feature-tagged AI cost data yet, build the tagging this week. The numbers you don't have are the ones costing you the most.
- **Re-tier one feature.** Look at your top-spend feature. Is it on the right tier per Chapter 1.7? Often the answer is no.
- **Check cache hit rate.** If you use prompt caching, what's the actual hit rate? If <80%, find out why.
- **Run the multi-language audit.** If your product serves non-English users, what's the actual per-locale cost? Compare to English baseline.

Each of these is a half-day of work. Each typically returns 10–30% on the AI bill.

What's next

This playbook is a snapshot. Prices change every quarter. New cost dimensions emerge as model architectures evolve. Multi-modal pricing in 2027 will look different than it does today.

I publish the **Token Economy Index** monthly: model price changes, vendor moves, real cost benchmarks. It's the operating manual that stays current — and it's the way the principles in this playbook get re-tested against new data.

Subscribe at aieconomylab.com.

Stay in touch

- 🌐 aieconomylab.com — calculator + monthly index
- 📦 github.com/aieconomylab — open-source pricing data and tools
- 📪 hello@aieconomylab.com — feedback, corrections, sponsor inquiries

If a chapter saved you money, tell me which one. If a chapter was wrong about your specific use case, tell me that too. The next edition is better because of both.

— Mehmet Karakose *AI Economy Lab April 2026*

Appendix A — Pricing Reference Tables

Snapshot: April 2026. Prices change quarterly. Live data: github.com/aieconomylab/ai-pricing. Calculator: aieconomylab.com.

A.1 LLMs

USD per million tokens.

Model	Input	Cached input	Output
OpenAI			
GPT-5.5	$5.00	$0.50	$30.00
GPT-5.5 Pro	$30.00	–	$180.00
GPT-5.4	$2.50	$0.25	$15.00
GPT-5.4 mini	$0.75	$0.075	$4.50
GPT-5.4 nano	$0.20	$0.02	$1.25
GPT-5.4 Pro	$30.00	–	$180.00
GPT-5	$1.25	$0.125	$10.00
GPT-5 mini	$0.25	$0.025	$2.00
GPT-5 nano	$0.05	$0.005	$0.40
GPT-5 Pro	$15.00	–	$120.00
o3	$2.00	$0.50	$8.00
o3-pro	$20.00	–	$80.00
o4-mini	$1.10	$0.275	$4.40
o3-mini	$1.10	$0.55	$4.40
Anthropic			
Claude Opus 4.7	$15.00	$1.50 (read)	$75.00
Claude Sonnet 4.6	$3.00	$0.30 (read)	$15.00
Claude Haiku 4.5	$0.80	$0.08 (read)	$4.00
Google			
Gemini 3.1 Pro	$2.00	$0.20	$12.00
Gemini 3 Flash	$0.50	$0.05	$3.00
Gemini 3.1 Flash-Lite	$0.25	$0.025	$1.50
Gemini 2.5 Pro	$1.25	$0.125	$10.00
Gemini 2.5 Flash	$0.30	$0.075	$2.50
xAI			
Grok 4	$3.00	–	$15.00
DeepSeek			
DeepSeek V4 Pro (75% launch promo, ends 2026-05-31)	$0.435	$0.003625	$0.87
DeepSeek V4 Pro (regular)	$1.74	$0.0145	$3.48
DeepSeek V4 Flash	$0.14	$0.0028	$0.28

Model	Input	Cached input	Output
DeepSeek V3	$0.27	$0.027	$1.10
DeepSeek R1	$0.55	$0.14	$2.19
Together (open-source)			
Llama 3.3 70B	$0.20	–	$0.20
Qwen 3 72B	$0.20	–	$0.20

Long-context tiers: Anthropic models charge 2× over 200K input. Gemini 3.1 Pro and Gemini 2.5 Pro charge ~2× over 200K input (input rises to $4 / output to $18 for 3.1 Pro past 200K). GPT-5 family is single-tier. Batch tier is 50% off across all providers.

A.2 Image generation

Provider	Standard	High quality
OpenAI DALL-E 3 (1024^2)	$0.040	$0.080 (HD)
OpenAI gpt-image-1	$0.011–0.07	$0.17
Google Imagen 3	$0.040	$0.060
Flux 1.1 Pro (Replicate)	$0.040	$0.055
Stable Diffusion 3.5 (Replicate)	$0.005–0.020	(varies by steps)
Midjourney v7	flat $30–120/mo	(with caps)

A.3 Video generation

Per second of generated output.

Provider	Standard tier	High res tier
OpenAI Sora 2 (1080p / 4K)	$0.50–1.50	$2.00–4.00
Google Veo 3	$0.50	$0.75
Runway Gen-4	$0.05–0.10	$0.20
Pika 2.5	$0.05–0.15	$0.30
Luma Ray 2	$0.10	$0.25

A.4 Audio

TTS (per 1,000 characters)

Provider	Standard	Premium
ElevenLabs	$0.18 (Pro)	$0.30 (Eleven v3)
OpenAI gpt-4o-mini-tts	$0.012 (output)	–
OpenAI gpt-realtime	$32/M output audio tokens	–
Google Chirp 3 HD	$0.013	–

STT (per minute)

Provider	Rate
OpenAI Whisper API	$0.006

Provider	Rate
Deepgram Nova-3	$0.0043
OpenAI gpt-realtime	$5/M input audio tokens
AssemblyAI	$0.012

Music generation (per second of audio)

Provider	Approximate range
Suno v5	$0.05–0.10 (subscription credits)
Udio	$0.05–0.10 (similar tiered)

A.5 Embeddings (per million tokens)

Provider	Model	Rate
OpenAI	text-embedding-3-small	$0.020
OpenAI	text-embedding-3-large	$0.130
Cohere	embed-v4	$0.10
Voyage	voyage-3	$0.06
Google	gemini-embedding-001	$0.025
Open-source (Together)	nomic-embed-v2	$0.005–0.01

A.6 Fine-tuning

Training (one-time)

Provider	Model	Rate
OpenAI	gpt-5-mini fine-tune	$25/M tokens
OpenAI	gpt-5 fine-tune	$90/M tokens
Together	Llama-class (LoRA)	$1.20/M tokens
Fireworks	Llama / Mistral fine-tune	$0.50/M tokens

Inference premium

Provider	Premium over base
OpenAI fine-tuned	2×
Together fine-tuned	none
Fireworks fine-tuned	minimal

These tables go stale. The auto-updated source is `github.com/aieconomylab/ai-pricing` — also available as an npm package and via jsDelivr CDN.

Appendix B — Tokenizer Ratio Table

Per-language token-overhead multipliers relative to English (1.00). Same content; different languages cost different numbers of tokens.

Measured on Universal Declaration of Human Rights translations across the major tokenizer families. Live data: github.com/aieconomylab/ai-pricing/blob/main/data/tokenizer-ratios.json.

All 20 locales × 3 tokenizer families

Language	Code	o200k_base	claude	gemini
English	en	1.00×	1.00×	1.00×
Spanish	es	1.05×	1.05×	1.05×
Chinese (Simplified)	zh-CN	1.08×	1.10×	1.05×
German	de	1.15×	1.15×	1.15×
French	fr	1.15×	1.15×	1.15×
Italian	it	1.15×	1.15×	1.15×
Dutch	nl	1.15×	1.15×	1.15×
Portuguese (BR)	pt-BR	1.15×	1.15×	1.10×
Indonesian	id	1.25×	1.25×	1.20×
Japanese	ja	1.25×	1.30×	1.25×
Russian	ru	1.25×	1.30×	1.25×
Polish	pl	1.30×	1.30×	1.25×
Korean	ko	1.30×	1.35×	1.30×
Ukrainian	uk	1.35×	1.35×	1.30×
Turkish	tr	1.35×	1.40×	1.30×
Vietnamese	vi	1.40×	1.40×	1.35×
Arabic	ar	1.45×	1.55×	1.45×
Persian / Farsi	fa	1.50×	1.55×	1.50×
Hindi	hi	1.55×	1.60×	1.55×
Thai	th	1.65×	1.70×	1.65×

How to use this table

For a budget estimate in language X:

```
expected_cost = base_cost(English) × ratio(X, tokenizer)
```

Example: a feature costs $0.022 per turn in English on a model using `o200k_base`. The Turkish version of the same feature on the same model costs:

```
$0.022 × 1.35 = $0.030 per turn (~36% higher)
```

For Thai:

```
$0.022 × 1.65 = $0.036 per turn (~64% higher)
```

When the ratio matters most

- **Multi-locale subscription products.** If pricing is flat across locales, your margin shrinks in high-overhead languages by exactly this ratio.
- **Long-context features.** A 50K-token RAG context in Thai is effectively a 82K-token context in English equivalent. Past context-window thresholds, costs jump.
- **Reasoning models in non-English.** Reasoning happens in the model's "internal" representation, which may not align with the input language. The output ratios above apply to visible output; reasoning tokens are often closer to English ratios regardless of input language.

Methodology

Ratios computed by:

1. Translating ~500 words from the Universal Declaration of Human Rights into each target language.
2. Running each translation through `gpt-tokenizer` for `o200k_base`, the Claude tokenizer, and the Gemini tokenizer.
3. Recording `tokens_in_target_language / tokens_in_english`.
4. Repeating across 5 sample passages and averaging.

This benchmark is approximate. Real production text (technical, conversational, structured) tokenizes slightly differently than UDHR-style content. For mission-critical budgets, run the same measurement on a sample of your actual corpus.

Source data

The full dataset, including per-tokenizer breakdowns, raw token counts, and reproducible benchmark code, lives in the open-source repo:

- **Repo:** github.com/aieconomylab/ai-pricing
- **File:** `data/tokenizer-ratios.json`
- **License:** MIT — use, modify, redistribute freely

Contributions for additional languages, additional tokenizer families, or refined benchmarks are welcome via PR.

Appendix C — Glossary

Concepts referenced in the playbook, defined briefly. For depth, the relevant chapter is noted.

Agent loop — A pattern where the model calls a tool, observes the result, decides what to do next, and repeats. Costs grow non-linearly with steps because context accumulates. (Chapter 2.6)

Batch API — Provider-offered async processing tier. ~50% off sync rates with up to 24-hour SLA. Same model, same quality, just async. (Chapter 1.5)

Cached input — Input tokens that the provider has already processed and cached. Charged at a fraction of normal input rate (10% on Anthropic, 50% on OpenAI). Requires architectural commitment to a stable prompt prefix. (Chapter 1.2)

Context window — The maximum input + output tokens a model can process in one call. "1M context" is the upper limit, not the recommended amount. Long context is billed twice — once for the input, again in degraded answer quality. (Chapter 1.3)

Custom field (Beehiiv terminology) — A user-defined field on subscriber records. Used here as the recommended way to segment subscribers by locale (vs. tags or utm fields).

Embedding — A numerical vector representation of text, used for semantic search and RAG. Cheapest billable AI primitive. Migrating to a new embedding model requires re-indexing the entire corpus — that's the dominant cost, not the embedding generation. (Chapter 2.4)

Few-shot examples — Sample input/output pairs included in the prompt to teach the model the desired format. Adds tokens but improves output. Cache-friendly. (Chapter 1.2)

Fine-tuning — Training a custom version of a base model on your data. One-time training cost plus a recurring inference premium (often 2× base model on closed providers). Doesn't pay back until ~350K calls/year for typical use cases. (Chapter 2.5)

Honeypot field — An invisible form field that bots fill out and humans don't (because they can't see it). Spam protection without CAPTCHA friction. Used in the AI Economy Lab subscribe form.

KV cache — The internal key-value pairs the model computes when processing input. Cached input works by storing these pre-computed KV pairs and reusing them. (Chapter 1.2)

Lost in the middle — Empirical pattern where models attend less to information in the middle of long inputs than to the start or end. Quality cost of long context, regardless of price. (Chapter 1.3)

LoRA (Low-Rank Adaptation) — A lightweight fine-tuning technique that trains a small adapter layer rather than the full model. Used by Together and similar open-source-base providers. Cheap, portable, no inference premium. (Chapter 2.5)

Output:input ratio — Tokens of output / tokens of input for a given task. <0.1 means input-heavy (RAG, classification); >0.3 means output-heavy (chat, agents); >1.0 typical for reasoning models and agent loops. The single most useful number for picking a model. (Chapter 1.1)

p50 / p95 / p99 — Percentile metrics. p50 = median; p95 = 95th percentile; p99 = 99th percentile. Useful for cost forecasting because AI workloads have long tails — average cost is misleading; p95 or p99 is what surprises the bill. (Chapter 3.1)

Prompt caching — Provider-side caching of repeated prompt prefixes. See “cached input.” (Chapter 1.2)

RAG (Retrieval-Augmented Generation) — Pattern where you retrieve relevant context (via embedding similarity) and pass only that to the model, instead of stuffing the whole knowledge base into context. Usually 50–100× cheaper than long-context stuffing. (Chapter 1.3)

Reasoning tokens — Hidden internal tokens generated by reasoning models (o3, Gemini Thinking, Claude Extended Thinking) before producing the visible answer. Billed as output, often 5–20× the visible output count. The single biggest source of bill shock in 2026. (Chapter 1.6)

reasoning_effort / thinking_budget — Provider parameters that cap how much reasoning a reasoning model performs. Without a cap, reasoning models can think indefinitely on a hard prompt. Always set a cap. (Chapter 1.6)

Reranker — A model used to re-score retrieved candidates after embedding-based retrieval. Often improves quality more than swapping embedding models. (Chapter 2.4)

Structured output — Constraining the model’s output to a specific schema (JSON Schema, function call, regex). Reduces output length, improves parsing, lowers cost on output-heavy tasks. (Chapter 1.1)

Tier (model tier) — Loose grouping of models by quality and price. Top tier (GPT-5, Claude Opus 4.7), workhorse (Sonnet 4.6, Gemini 2.5 Pro), small/fast (GPT-5 mini, Haiku 4.5, Flash). Re-tiering features is usually the cheapest optimization. (Chapter 1.7)

Token — The unit of text the model sees. Roughly ¾ of a word in English; often 1.0–1.7× a word in other languages. The cost unit for nearly all LLM, embedding, and (newly) audio billing. (Chapter 1.4)

Tokenizer — The algorithm that converts text into tokens. Three major families today: `o200k_base` (OpenAI), Claude tokenizer (Anthropic), Gemini tokenizer (Google). Token counts vary across families even for the same text. (Chapter 1.4)

TTS / STT — Text-to-Speech / Speech-to-Text. TTS is typically 20–50× more expensive than STT for equivalent durations. (Chapter 2.3)

utm fields (utm_source, utm_medium, utm_campaign) — Standard URL-tracking parameters used to attribute conversions to traffic sources. Beehiiv’s API auto-classifies API-created subscribers as “direct” regardless of utm fields sent — use tags for locale segmentation instead.

Have a term we should add? Open an issue on github.com/aieconomylab or e-mail hello@aieconomylab.com.