# AI Cost Playbook

FinOps for LLM systems, token economics, caching, routing, and cost governance

Caio Incau

# AI Cost Playbook

FinOps for LLM systems, token economics, caching, routing, and cost governance

**Caio Incau**

# Table of Contents



Code snapshots for every chapter can be found in this book's GitHub repository: `https://github.com/PacktPublishing/The-AI-Cost-Playbook`.

# Preface

In early 2024, a team I worked with at a large fintech deployed a customer support agent powered by Claude Sonnet. The proof-of-concept was brilliant — it resolved tickets faster, customers loved it, and leadership greenlit a production rollout. Nobody checked the cost model. Within six weeks the monthly API bill crossed six figures. The agent was calling tools in recursive loops, stuffing entire conversation histories into every request, and nobody had instrumented a single token counter. We killed the project on a Thursday afternoon.

That experience radicalized me about AI cost engineering. Not because the technology was wrong — it was working beautifully — but because nobody on the team had a mental model for how LLM costs actually behave. We were treating a usage-based, token-level billing system like a fixed SaaS subscription. The bill was the inevitable result.

Since then I've spent two years building cost observability, optimization layers, and governance frameworks for LLM workloads across multiple production systems. I've watched teams reduce their API spend by 60-80% without degrading quality, and I've watched other teams burn through six-figure budgets on workloads that could have been served by a model one-tenth the price. The difference was never the technology. It was the engineering discipline.

## Who this book is for

This book is for engineers, tech leads, and platform teams running LLM workloads in production — or about to. You should be comfortable with Python and have at least a passing familiarity with LLM APIs (sending messages, receiving completions). You don't need to be a machine learning engineer. You don't need a GPU cluster. You need to care about what your AI systems cost and why.

If you're an engineering leader accountable for AI spend, the governance chapters (12-14) are written with you in mind, but I'd encourage you to read the technical chapters too. Understanding token economics at a mechanical level changes how you make architectural decisions.

## What you'll build

Throughout this book you'll build **tokenwatch** — a Python toolkit for LLM cost observability and optimization. It starts as a simple token-counting middleware and grows into a complete system with:

- **Token metering** — intercept every LLM call and record input, output, cached, and thinking tokens
- **Cost attribution** — map costs to features, users, and teams with a unit-economics dashboard

- **Prompt caching** — a cache layer that exploits provider-native caching (Anthropic prompt caching, OpenAI cached context)
- **Model routing** — a quality/cost cascade that sends cheap queries to small models and escalates to powerful models only when needed
- **Budget governance** — alerts, quotas, and circuit breakers that prevent runaway spend

Each chapter adds a module to tokenwatch. By the end, you'll have a production-grade cost engineering toolkit — and more importantly, a mental model for thinking about LLM costs that will outlast any specific API or pricing change.

## How to read this book

The book is designed to be read sequentially. Each chapter builds on the previous one, both conceptually and in code. If you skip ahead to model routing (Chapter 8) without understanding token economics (Chapter 2) and metering (Chapter 3), the optimization decisions won't make sense.

That said, if you're already running LLM workloads and just need to solve a specific problem, each chapter's introduction tells you what it assumes and what it builds. Jump to what you need, but be ready to backtrack if the foundations feel shaky.

Code examples use Python 3.12+ with the Anthropic and OpenAI SDKs. The project is framework-agnostic — no LangChain, no LlamaIndex, no heavy abstractions. You'll understand every line because you wrote it.

## A note on pricing

LLM pricing changes constantly. By the time you read this, the specific per-token prices I could cite will be wrong. Instead of hardcoding prices, tokenwatch uses a `pricing.yaml` configuration file that you update as prices change. The book teaches you the *mechanics* — input tokens vs. output tokens vs. cached tokens vs. thinking tokens, how batch discounts work, how caching changes your effective rate — rather than asserting volatile numbers as facts. The principles are durable even when the prices are not.

## Acknowledgments

Thanks to the engineering teams who let me experiment with increasingly aggressive cost optimization on their production workloads. Thanks to the FinOps community for pushing cloud cost engineering into the AI era. And thanks to everyone who reviewed early drafts of these chapters and told me when I was being too theoretical and not enough hands-on-keyboard.

Let's start cutting your AI bill.

*Caio Incau — June 2026*

{mainmatter}

# Chapter 1 — The AI Bill Shock

Your cloud bill is predictable. You provision instances, you know the hourly rate, you set autoscaling bounds. Even if usage spikes, the cost model is linear and well-understood. Then you add an LLM-powered feature and the bill becomes a mystery. What happened?

LLM costs are fundamentally different from traditional compute costs. They're usage-based at the token level, non-linear with respect to user value, and driven by decisions that individual developers make in prompt templates — not by infrastructure capacity planning. A single poorly designed prompt can cost 100x more than a well-designed one serving the same purpose. A recursive agent loop can burn through your monthly budget in an afternoon.

This chapter is about understanding *why* AI costs are different, *how* they behave, and *what* you need to measure before you can optimize anything. By the end, you'll have the first piece of tokenwatch: a configuration system and the mental model for everything that follows.

---

## Why LLM costs surprise everyone

Traditional software costs scale with infrastructure. You pay for compute, storage, and network — resources that are provisioned ahead of time and priced per hour or per GB. The relationship between usage and cost is straightforward: more users means more servers means a higher bill, but the cost-per-user is roughly constant.

LLM costs break this model in three ways.

### 1. Token-level billing

You don't pay for "an API call." You pay for the number of tokens processed — both the tokens you send in (the prompt) and the tokens the model generates (the completion). A simple "yes or no" classification might consume 500 input tokens and 5 output tokens. A detailed analysis of a document might consume 50,000 input tokens and 2,000 output tokens. Both are "one API call," but the second costs 50-100x more.

This means cost is driven by *what you ask* and *how you ask it*, not by how many times you ask. A prompt template that includes unnecessary context — "here's the entire conversation history, now answer this question" — costs more every single time it runs. Multiply by thousands of requests per day and you have a real problem.

### 2. Non-linear cost scaling

In traditional systems, serving 10x more users costs roughly 10x more. With LLMs, cost scaling depends on how the product is designed. Consider an agent that answers customer questions:

- **Simple question** (no tool calls): 1 LLM call, ~500 tokens total. Cost: ~$0.003.
- **Complex question** (3 tool calls): 4 LLM calls (initial + 3 tool results), ~8,000 tokens total. Cost: ~$0.05.
- **Ambiguous question** (agent loop, 8 iterations): 9 LLM calls, ~40,000 tokens total. Cost: ~$0.25.

The "ambiguous question" costs 80x more than the "simple question," but both look like "one user interaction" in your product metrics. If 10% of your queries fall into the ambiguous category, they might drive 60% of your total cost. This non-linearity makes forecasting extremely hard.

**3. Developer-driven cost**

In traditional infrastructure, costs are driven by capacity decisions made by ops teams. In LLM systems, costs are driven by prompt design decisions made by individual developers. The developer who writes `You are a helpful assistant. Here is the full document: {document}` versus `Classify the following excerpt: {first_500_chars}` has just made a 10x cost decision — often without realizing it.

This is a governance problem, not a technology problem. Without visibility into per-prompt costs, developers optimize for quality and correctness (which is reasonable) without any feedback on the cost implications of their design choices.

I> **Note:** This is not an argument against quality. It's an argument for *informed trade-offs*. Often, you can get the same quality at a fraction of the cost — but only if you measure first.

---

## The anatomy of an LLM API bill

Let's look at what you're actually paying for when you call an LLM API. Every provider — Anthropic, OpenAI, Google — uses roughly the same billing model with minor variations.

**Token classes**

Modern LLM APIs bill for several distinct classes of tokens:

**Input tokens** — the tokens in your prompt: the system message, user message, conversation history, tool definitions, and any injected context. You control this directly through prompt design.

**Output tokens** — the tokens the model generates in response. You control this indirectly through instructions ("be concise", `max_tokens`) and directly through the task design (classification vs. open-ended generation).

**Cached input tokens** — tokens that hit a provider-side cache (like Anthropic's prompt caching). These are billed at a significant discount — typically 90% cheaper than regular input tokens. We'll cover caching mechanics in detail in Chapter 6.

**Thinking tokens** — tokens the model generates during extended thinking (Claude's "thinking" feature, OpenAI's o-series reasoning tokens). These are billed at output token rates but can be substantial — a complex reasoning task might generate 5,000-10,000 thinking tokens before producing a 200-token answer.

The relative cost of these token classes varies by provider and model, but the pattern is consistent: output tokens cost more than input tokens, and cached tokens cost less than uncached tokens. This asymmetry is the foundation of most cost optimization strategies.

**A concrete example**

Let's trace the cost of a single request through the billing model. Suppose you're using Claude Sonnet to analyze a customer support ticket:

```
System prompt:           200 tokens  (input)
Conversation history:  3,000 tokens  (input)
Current user message:    100 tokens  (input)
Tool definitions:        800 tokens  (input)
──────────────────────────────────────────
Total input:           4,100 tokens

Model response:          500 tokens  (output)
```

Now imagine this request runs 10,000 times per day. That's 41 million input tokens and 5 million output tokens per day. At typical managed-API pricing, you're looking at a material daily cost — and that's for a single feature.

The cost levers are immediately visible: the conversation history (3,000 tokens) dominates the input. If you can cache the system prompt and tool definitions (1,000 tokens), truncate the history to the most recent 3 messages (reducing from 3,000 to 1,000 tokens), and use a cheaper model for simple tickets, you could reduce costs by 50-70% on this single endpoint.

But you can't optimize what you can't measure. That's where tokenwatch begins.

---

## Setting up the tokenwatch project

Let's create the project structure. tokenwatch is a Python package that will grow throughout the book. We start with the minimal scaffolding.

```
"""tokenwatch/__init__.py — Cost observability toolkit for LLM systems."""

__version__ = "0.1.0"
```

```
"""tokenwatch/config.py — Configuration management for tokenwatch."""

from dataclasses import dataclass, field
from pathlib import Path
from typing import Any

import yaml


@dataclass
class ModelPricing:
    """Pricing for a single model. Rates are per million tokens."""
    model_id: str
    provider: str
    input_rate: float       # $ per 1M input tokens
    output_rate: float      # $ per 1M output tokens
```

```
    cached_rate: float = 0.0     # $ per 1M cached input tokens
    thinking_rate: float = 0.0   # $ per 1M thinking tokens (if applicable)
    batch_input_rate: float = 0.0    # $ per 1M input tokens in batch mode
    batch_output_rate: float = 0.0   # $ per 1M output tokens in batch mode

    def cost_for_tokens(
        self,
        input_tokens: int = 0,
        output_tokens: int = 0,
        cached_tokens: int = 0,
        thinking_tokens: int = 0,
        batch: bool = False,
    ) -> float:
        """Calculate the cost for a given token breakdown."""
        if batch and self.batch_input_rate > 0:
            input_cost = (input_tokens / 1_000_000) * self.batch_input_rate
            output_cost = (output_tokens / 1_000_000) * self.batch_output_rate
        else:
            input_cost = (input_tokens / 1_000_000) * self.input_rate
            output_cost = (output_tokens / 1_000_000) * self.output_rate

        cached_cost = (cached_tokens / 1_000_000) * self.cached_rate
        thinking_cost = (thinking_tokens / 1_000_000) * self.thinking_rate

        return input_cost + output_cost + cached_cost + thinking_cost


@dataclass
class TokenwatchConfig:
    """Global configuration loaded from pricing.yaml."""
    models: dict[str, ModelPricing] = field(default_factory=dict)
    default_model: str = ""
    budget_alert_threshold_usd: float = 100.0
    budget_hard_limit_usd: float = 1000.0

    @classmethod
    def from_yaml(cls, path: str | Path) -> "TokenwatchConfig":
        """Load configuration from a YAML file."""
        path = Path(path)
        if not path.exists():
            raise FileNotFoundError(
                f"Pricing config not found: {path}. "
                "Copy pricing.example.yaml and update with current rates."
            )

        with open(path) as f:
            raw = yaml.safe_load(f)

        models = {}
```

```
        for model_data in raw.get("models", []):
            model = ModelPricing(
                model_id=model_data["model_id"],
                provider=model_data["provider"],
                input_rate=model_data["input_rate"],
                output_rate=model_data["output_rate"],
                cached_rate=model_data.get("cached_rate", 0.0),
                thinking_rate=model_data.get("thinking_rate", 0.0),
                batch_input_rate=model_data.get("batch_input_rate", 0.0),
                batch_output_rate=model_data.get("batch_output_rate", 0.0),
            )
            models[model.model_id] = model

        return cls(
            models=models,
            default_model=raw.get("default_model", ""),
            budget_alert_threshold_usd=raw.get("budget_alert_threshold_usd",
100.0),
            budget_hard_limit_usd=raw.get("budget_hard_limit_usd", 1000.0),
        )

    def get_model(self, model_id: str) -> ModelPricing:
        """Get pricing for a model, raising if not configured."""
        if model_id not in self.models:
            raise KeyError(
                f"Model '{model_id}' not found in pricing config. "
                f"Available: {list(self.models.keys())}"
            )
        return self.models[model_id]
```

Now create the pricing configuration file. This is the file the reader updates as prices change — we never hardcode prices in the application code.

```
# pricing.yaml — Update these rates from provider pricing pages.
# Rates are in USD per 1 million tokens.
# Last verified: 2026-06-01 (UPDATE THIS DATE when you refresh prices)

default_model: "claude-sonnet-4-20250514"

budget_alert_threshold_usd: 100.0
budget_hard_limit_usd: 1000.0

models:
  - model_id: "claude-sonnet-4-20250514"
    provider: "anthropic"
    input_rate: 3.0      # Check: https://docs.anthropic.com/en/docs/about-
claude/pricing
    output_rate: 15.0
    cached_rate: 0.30
    thinking_rate: 15.0
```

```
    batch_input_rate: 1.5
    batch_output_rate: 7.5

  - model_id: "claude-haiku-4-5-20250514"
    provider: "anthropic"
    input_rate: 0.80
    output_rate: 4.0
    cached_rate: 0.08
    thinking_rate: 4.0
    batch_input_rate: 0.40
    batch_output_rate: 2.0

  - model_id: "claude-opus-4-20250514"
    provider: "anthropic"
    input_rate: 15.0
    output_rate: 75.0
    cached_rate: 1.50
    thinking_rate: 75.0
    batch_input_rate: 7.5
    batch_output_rate: 37.5

  - model_id: "gpt-4o"
    provider: "openai"
    input_rate: 2.50
    output_rate: 10.0
    cached_rate: 1.25
    batch_input_rate: 1.25
    batch_output_rate: 5.0

  - model_id: "gpt-4o-mini"
    provider: "openai"
    input_rate: 0.15
    output_rate: 0.60
    cached_rate: 0.075
    batch_input_rate: 0.075
    batch_output_rate: 0.30
```

W> **Warning:** These prices are examples based on publicly available pricing at the time of writing. LLM pricing changes frequently. Always verify current rates on the provider's official pricing page before using these values for budgeting or forecasting.

Let's verify the configuration loads correctly:

```
"""test_config.py — Verify pricing configuration."""

from tokenwatch.config import TokenwatchConfig

config = TokenwatchConfig.from_yaml("pricing.yaml")

# Check a model is loaded
```

```
sonnet = config.get_model("claude-sonnet-4-20250514")
print(f"Model: {sonnet.model_id}")
print(f"Provider: {sonnet.provider}")
print(f"Input rate: ${sonnet.input_rate}/1M tokens")
print(f"Output rate: ${sonnet.output_rate}/1M tokens")

# Calculate cost for our example request
cost = sonnet.cost_for_tokens(
    input_tokens=4_100,
    output_tokens=500,
)
print(f"\nSingle request cost: ${cost:.6f}")
print(f"Daily cost (10k requests): ${cost * 10_000:.2f}")
print(f"Monthly cost (10k req/day): ${cost * 10_000 * 30:.2f}")
```

Running this shows you the cost breakdown for the customer support example we walked through earlier. The numbers will match your `pricing.yaml` — when prices change, you update the YAML and the calculations automatically reflect reality.

---

## The cost visibility problem

Here's the uncomfortable truth about most teams running LLM workloads: they have no idea what individual features cost. They see a monthly API bill — a single number — and they might know the total token count. But they can't answer basic questions like:

- Which feature is the most expensive?
- What's the cost per user interaction?
- Which prompt template drives the most spend?
- Are costs growing linearly with usage, or super-linearly?

Without answers to these questions, optimization is guesswork. You might spend a week optimizing a prompt that accounts for 2% of your spend while ignoring one that accounts for 60%.

This is the same problem that cloud FinOps solved for infrastructure costs a decade ago. The FinOps Foundation defines a "crawl, walk, run" maturity model:

**Crawl** — You can see your total AI spend broken down by provider and model. You know the trend.

**Walk** — You can attribute costs to features, teams, and users. You have unit economics (cost per request, cost per user).

**Run** — You have automated optimization (caching, routing, batching), budget governance, and forecasting.

Most teams are stuck at crawl. This book takes you to run.

T> **Tip:** If you're wondering whether cost optimization is worth the engineering investment, here's a quick heuristic: if your monthly LLM API spend exceeds $5,000, you almost certainly

have low-hanging optimizations worth 30-50% savings. If it exceeds $50,000, the savings from this book will likely exceed an engineer's annual salary.

---

## The tokenwatch roadmap

Here's what we'll build across the book, with each chapter adding a module:

| Chapter | Module | What it does |
|---|---|---|
| 1 (this) | `config` | Pricing configuration, model registry |
| 2 | `tokens` | Token counting, context window mechanics |
| 3 | `meter` | Metering middleware, request logging |
| 4 | `economics` | Unit cost calculations, margin analysis |
| 5 | `prompt` | Prompt compression, structured output optimization |
| 6 | `cache` | Prompt caching layer, hit rate tracking |
| 7 | `batch` | Batch processing queue, async workloads |
| 8 | `router` | Model routing, quality/cost cascades |
| 9 | `rag` | RAG cost optimization, chunking strategies |
| 10 | `agents` | Agent cost controls, loop budgets |
| 11 | `selfhost` | Break-even calculator, GPU cost modeling |
| 12 | `governance` | Budgets, quotas, showback |
| 13 | `forecast` | Forecasting, provider mix optimization |
| 14 | Full integration | Dashboard, runbook, production deployment |

By the end, `tokenwatch` will be a toolkit you can drop into any Python LLM application to get immediate cost visibility and automated optimization.

---

## Common mistakes

**1. Treating LLM costs like infrastructure costs.** Teams apply traditional capacity planning — "we need X instances for Y users" — to LLM workloads. But LLM costs are driven by prompt design, not capacity. Two features serving the same number of users can differ in cost by 100x based on how they use the model.

**2. Optimizing before measuring.** The instinct is to jump straight to "let's use a cheaper model." But without per-feature cost attribution, you might optimize the wrong thing. Always instrument first, optimize second.

**3. Ignoring output tokens.** Output tokens typically cost 3-5x more than input tokens. A feature that generates long responses (code generation, detailed analysis) can be dramatically more

expensive than one that generates short responses (classification, extraction) — even with the same input.

**4. Hardcoding prices in application code.** Prices change. If your cost calculations are scattered across the codebase with magic numbers, you'll never keep them accurate. Centralize pricing in a configuration file that's easy to update.

---

## Exercises

**Easy:** Add a new model to `pricing.yaml` (pick any model from a provider's current pricing page). Load the config and verify the model appears in `config.models`.

**Medium:** Write a function `estimate_monthly_cost(config, model_id, daily_requests, avg_input_tokens, avg_output_tokens)` that calculates the projected monthly cost for a workload. Test it with realistic numbers for a feature you're building or considering.

**Hard:** Create a `PricingComparator` class that takes a `TokenwatchConfig` and a workload description (daily requests, average input/output tokens) and generates a comparison table showing the monthly cost across all configured models. Include a column showing the percentage savings compared to the most expensive option.

---

## Summary

In this chapter you learned:

- **LLM costs are fundamentally different** from traditional infrastructure costs — they're token-level, non-linear, and developer-driven
- **Token classes matter** — input, output, cached, and thinking tokens are billed at different rates, and understanding these classes is the foundation of optimization
- **Cost visibility is the prerequisite** for optimization — you can't improve what you can't measure
- **Pricing belongs in configuration** — the `pricing.yaml` pattern keeps your cost calculations accurate as provider prices change
- **tokenwatch starts with config** — a clean data model for pricing that every subsequent module will build on

### Next chapter

Now that you understand why LLM costs behave differently and have a pricing configuration system, we need to go deeper into the unit of cost: the token. Chapter 2 covers token economics in detail — how tokenizers work, why context windows matter for cost, what thinking tokens are, and how to count tokens before you send a request. This knowledge is essential for everything that follows.

---

A> **Code for this chapter:** `project/cap01/tokenwatch/`

# Chapter 2 — Token Economics 101

Every dollar on your LLM bill traces back to tokens. Not characters, not words, not API calls — tokens. A token is the fundamental unit of both processing and billing in language models. If you don't understand how tokenization works, how context windows affect cost, and how different token classes are priced, every optimization decision you make will be based on intuition rather than mechanics.

This chapter goes deep on token economics. By the end, you'll be able to look at any prompt and estimate its cost before sending it to the API. You'll understand why a 200-word prompt can cost more than a 2,000-word prompt in certain configurations, and why context window size is both a capability feature and a cost trap. We'll also add a token counting module to tokenwatch that lets you pre-compute costs locally.

---

## How tokenization works

LLMs don't process raw text. They process sequences of integers, where each integer represents a token — a chunk of text that the model's tokenizer has learned to recognize. The mapping from text to tokens is determined during training and varies by model family.

Most modern models use a variant of Byte Pair Encoding (BPE). The tokenizer starts with individual bytes and iteratively merges the most frequent pairs into single tokens. The result is a vocabulary of subword units: common words like "the" become single tokens, while rare words like "defenestration" are split into multiple tokens ("def", "en", "est", "ration" or similar).

This has direct cost implications:

- **English text** averages roughly 1 token per 4 characters, or about 1.3 tokens per word
- **Code** tends to be more token-dense — whitespace, brackets, and variable names can each be individual tokens
- **Non-Latin scripts** (Chinese, Japanese, Korean, Arabic) use significantly more tokens per character
- **Structured data** (JSON, XML) has high token overhead from delimiters and keys

**Worked example: format matters**

To see the cost impact concretely, consider encoding the same information — five customer records — in three different formats, priced at the configured rate from `pricing.yaml` ($3.00 per million input tokens in our example configuration):

| Format | Token count (approx.) | Cost per 100K requests |
|---|---|---|
| Prose ("Customer Alice, age 34, plan premium...") | 120 tokens | $0.036 |
| JSON (`{"name":"Alice","age":34,"plan":"premium"}`) | 165 tokens | $0.050 |
| XML (`<customer><name>Alice</name>...`) | 210 tokens | $0.063 |

The same information in XML costs 75% more than prose. At 100,000 requests per day, that format choice alone adds roughly $14/month of input cost per payload field. This is why the tokenwatch project's `estimate_tokens` function exists: it lets you compare representations before committing to one in production.

```
"""tokenwatch/tokens.py — Token counting and estimation."""

import tiktoken
from anthropic import Anthropic


# tiktoken is the open-source tokenizer library that matches
# OpenAI's tokenization. For Anthropic models, we use their
# SDK's built-in token counting.
_openai_encoding = tiktoken.get_encoding("cl100k_base")
_anthropic_client: Anthropic | None = None


def _get_anthropic_client() -> Anthropic:
    """Lazy-initialize the Anthropic client for token counting."""
    global _anthropic_client
    if _anthropic_client is None:
        _anthropic_client = Anthropic()
    return _anthropic_client


def count_tokens_openai(text: str) -> int:
    """Count tokens using OpenAI's tokenizer (cl100k_base).

    Works for GPT-4o, GPT-4o-mini, and most current OpenAI models.
    """
    return len(_openai_encoding.encode(text))


def count_tokens_anthropic(
    text: str,
    model: str = "claude-sonnet-4-20250514",
) -> int:
```

```
    """Count tokens using Anthropic's token counting API.

    This uses the official /v1/messages/count_tokens endpoint,
    which gives exact counts for Claude models.
    """
    client = _get_anthropic_client()
    result = client.messages.count_tokens(
        model=model,
        messages=[{"role": "user", "content": text}],
    )
    return result.input_tokens


def estimate_tokens(text: str) -> int:
    """Quick local estimate without API calls.

    Uses the heuristic of ~4 characters per token for English text.
    Not exact, but useful for fast cost estimation before committing
    to an API call.
    """
    return max(1, len(text) // 4)


def count_message_tokens(
    messages: list[dict],
    system: str = "",
    model: str = "claude-sonnet-4-20250514",
) -> dict[str, int]:
    """Count tokens for a complete message payload.

    Returns a breakdown of token counts by component.
    """
    breakdown = {
        "system_tokens": estimate_tokens(system) if system else 0,
        "message_tokens": 0,
        "overhead_tokens": 0,
    }

    for msg in messages:
        content = msg.get("content", "")
        if isinstance(content, str):
            breakdown["message_tokens"] += estimate_tokens(content)
        elif isinstance(content, list):
            # Handle multi-part content blocks
            for block in content:
                if isinstance(block, dict) and "text" in block:
                    breakdown["message_tokens"] += estimate_tokens(block["text"])

    # API overhead: message framing, role tokens, etc.
```

```
    # Each message adds ~4 tokens of overhead
    breakdown["overhead_tokens"] = len(messages) * 4

    breakdown["total_estimated"] = sum(breakdown.values())
    return breakdown
```

T> **Tip:** The `estimate_tokens` function is intentionally rough. Use it for quick cost estimates and sanity checks. For exact counts, use the provider-specific functions. The Anthropic SDK's `count_tokens` method gives exact results for Claude models; `tiktoken` gives exact results for OpenAI models.

---

## Context windows: capability or cost trap?

Every model has a maximum context window — the total number of tokens (input + output) it can process in a single request. Claude Sonnet supports a 200K-token context window. GPT-4o supports 128K tokens. These numbers are impressive, but they're also dangerous.

A larger context window means you *can* send more data in a single request. It doesn't mean you *should*. Here's the cost reality:

```
Scenario A: Send 2,000 input tokens, get 500 output tokens
Scenario B: Send 100,000 input tokens, get 500 output tokens
```

Both produce the same 500-token answer. Scenario B costs 50x more on the input side. If you're stuffing an entire document into the context "just in case the model needs it," you're paying for 98,000 tokens of context that might be irrelevant to the question.

This is the context window trap: the model handles large contexts gracefully, so developers stop thinking about what they're sending. The right approach is surgical context — send only what the model needs to answer the specific question.

**Real-world scenario: the 200K context trap**

At a large fintech, an engineering team built a document analysis feature that loaded entire PDF contracts (40,000-80,000 tokens) into the context window for every user question. The feature answered questions accurately, so nobody investigated the cost. After instrumenting with tokenwatch, the team discovered that the feature consumed 60% of the company's total LLM budget — despite serving only 5% of total requests. The average question was "what is the renewal date?" and only needed a 500-token excerpt from the document's final section.

The fix was straightforward: use RAG (Chapter 9) to retrieve only the relevant section. Input tokens per request dropped from 60,000 to 3,000 — a 95% reduction. The cost of that single feature fell from $45,000/month to $2,200/month.

The lesson: context windows are a capability ceiling, not a cost target. The `ContextAnalysis` class in the tokenwatch code below embodies this principle — it flags requests that are using disproportionate context.

```python
"""tokenwatch/context.py — Context window analysis and optimization."""

from dataclasses import dataclass
from tokenwatch.tokens import estimate_tokens


@dataclass
class ContextAnalysis:
    """Analysis of context window utilization."""
    model: str
    max_context: int
    input_tokens: int
    reserved_output: int
    utilization_pct: float
    estimated_cost_usd: float
    wasted_context_tokens: int  # tokens sent but likely unused

    @property
    def efficiency_rating(self) -> str:
        if self.utilization_pct < 10:
            return "excellent"  # small, focused context
        elif self.utilization_pct < 50:
            return "good"
        elif self.utilization_pct < 80:
            return "watch"  # getting expensive
        else:
            return "danger"  # near limit, very expensive


# Context window sizes for common models (tokens)
MODEL_CONTEXT_WINDOWS = {
    "claude-sonnet-4-20250514": 200_000,
    "claude-haiku-4-5-20250514": 200_000,
    "claude-opus-4-20250514": 200_000,
    "gpt-4o": 128_000,
    "gpt-4o-mini": 128_000,
}


def analyze_context(
    messages: list[dict],
    system: str = "",
    model: str = "claude-sonnet-4-20250514",
    reserved_output: int = 4096,
) -> ContextAnalysis:
    """Analyze context window utilization for a request.

    This helps identify requests that are using disproportionate
    amounts of context, which directly translates to cost.
```

```
    """
    from tokenwatch.tokens import count_message_tokens
    from tokenwatch.config import TokenwatchConfig

    token_breakdown = count_message_tokens(messages, system, model)
    input_tokens = token_breakdown["total_estimated"]

    max_context = MODEL_CONTEXT_WINDOWS.get(model, 128_000)
    utilization = (input_tokens + reserved_output) / max_context * 100

    # Heuristic: if system prompt > 50% of input, that's potentially
    # wasted context that could be cached
    system_tokens = token_breakdown["system_tokens"]
    wasted = system_tokens if system_tokens > input_tokens * 0.5 else 0

    # Load cost from pricing.yaml (we'll refine this in Chapter 3)
    config = TokenwatchConfig.from_yaml("pricing.yaml")
    cost = input_tokens * config.get_rate(model, "input") / 1_000_000  # from
pricing.yaml

    return ContextAnalysis(
        model=model,
        max_context=max_context,
        input_tokens=input_tokens,
        reserved_output=reserved_output,
        utilization_pct=utilization,
        estimated_cost_usd=cost,
        wasted_context_tokens=wasted,
    )
```

---

## Input tokens vs. output tokens vs. the rest

The cost asymmetry between token classes is the single most important thing to understand for optimization. Let's break down each class and its cost implications.

### Input tokens

These are the tokens you send to the model: system prompt, user messages, conversation history, tool definitions, injected context (RAG results, documents). You have full control over input tokens.

Key insight: input tokens are paid *every time* the model is called. If your system prompt is 500 tokens and you make 100,000 requests per day, you're paying for 50 million system-prompt tokens per day — the same 500 tokens, over and over. This is exactly what caching is designed to solve (Chapter 6).

### Output tokens

These are the tokens the model generates. Output tokens are typically 3-5x more expensive than input tokens. This asymmetry exists because generation is computationally more expensive than processing input — each output token requires a full forward pass through the model.

Key insight: you can constrain output tokens with `max_tokens`, but be careful. Setting `max_tokens=100` on a task that naturally requires 500 tokens will produce truncated, useless responses. Better approaches include structured output (forcing JSON with a schema) and clear instructions about response length.

### Cached input tokens

When you use provider-native caching (Anthropic's prompt caching, OpenAI's automatic caching), tokens that hit the cache are billed at a steep discount — often 90% less than regular input tokens. The catch: there's typically a minimum prefix length (Anthropic requires the cached prefix to be at least 1,024 tokens for Sonnet-class models), and caches have TTL (time-to-live) limits.

The economics are compelling. If your system prompt + tool definitions are 2,000 tokens and you make 100,000 requests/day, caching those tokens saves you roughly 90% on the first 2,000 tokens of every request. At scale, this is significant.

### Thinking tokens

Claude's extended thinking and OpenAI's o-series reasoning models generate "thinking" tokens — internal reasoning that is visible to you but precedes the actual response. These tokens are billed at output token rates.

Thinking tokens can be substantial. A complex reasoning task might generate 5,000-10,000 thinking tokens to produce a 200-token answer. If you don't need the model to "show its work" or reason through complex logic, you're paying output rates for tokens you might not need. Disabling extended thinking when it's not required is a straightforward cost optimization.

```
"""Example: Token class breakdown for a request."""

from tokenwatch.config import TokenwatchConfig, ModelPricing

config = TokenwatchConfig.from_yaml("pricing.yaml")
sonnet = config.get_model("claude-sonnet-4-20250514")

# Scenario: customer support with thinking enabled
breakdown = {
    "input_tokens": 4_100,
    "output_tokens": 500,
    "cached_tokens": 0,
    "thinking_tokens": 3_000,
}

cost_no_cache = sonnet.cost_for_tokens(**breakdown)
print(f"Without caching: ${cost_no_cache:.6f}")
```

```
# Same request with system prompt cached (1,000 tokens cached)
breakdown_cached = {
    "input_tokens": 3_100,   # 4100 - 1000 cached
    "output_tokens": 500,
    "cached_tokens": 1_000,
    "thinking_tokens": 3_000,
}
cost_cached = sonnet.cost_for_tokens(**breakdown_cached)
print(f"With caching: ${cost_cached:.6f}")
print(f"Savings: {(1 - cost_cached/cost_no_cache)*100:.1f}%")

# Same request without thinking
breakdown_no_think = {
    "input_tokens": 4_100,
    "output_tokens": 500,
    "cached_tokens": 0,
    "thinking_tokens": 0,
}
cost_no_think = sonnet.cost_for_tokens(**breakdown_no_think)
print(f"Without thinking: ${cost_no_think:.6f}")
print(f"Savings vs thinking: {(1 - cost_no_think/cost_no_cache)*100:.1f}%")
```

I> **Note:** The savings percentages here depend entirely on your `pricing.yaml` values. The *pattern* is durable: caching reduces input cost, eliminating unnecessary thinking reduces output cost. The specific percentages change with pricing updates.

---

## Counting tokens before you spend

One of the most valuable practices in LLM cost engineering is counting tokens *before* you send the request. This lets you:

1. Estimate cost and decide whether to proceed
2. Choose the right model based on complexity
3. Truncate or compress context if it exceeds a budget
4. Log expected vs. actual costs for anomaly detection

Let's build a pre-flight cost estimator:

```
"""tokenwatch/preflight.py — Pre-request cost estimation."""

from dataclasses import dataclass
from tokenwatch.config import TokenwatchConfig
from tokenwatch.tokens import estimate_tokens


@dataclass
class PreflightEstimate:
    """Cost estimate before sending a request."""
    model: str
```

```
    estimated_input_tokens: int
    max_output_tokens: int
    estimated_cost_min: float  # assuming short output
    estimated_cost_max: float  # assuming max_tokens output
    recommendation: str


def preflight_check(
    config: TokenwatchConfig,
    model: str,
    system: str,
    messages: list[dict],
    max_tokens: int = 4096,
    thinking_budget: int = 0,
) -> PreflightEstimate:
    """Estimate cost before sending an API request.

    Returns an estimate with min/max cost range and a recommendation.
    """
    pricing = config.get_model(model)

    # Count input tokens
    input_tokens = estimate_tokens(system)
    for msg in messages:
        content = msg.get("content", "")
        if isinstance(content, str):
            input_tokens += estimate_tokens(content)
        elif isinstance(content, list):
            for block in content:
                if isinstance(block, dict) and "text" in block:
                    input_tokens += estimate_tokens(block["text"])
    input_tokens += len(messages) * 4  # overhead

    # Min cost: short response (10% of max_tokens)
    min_output = max(10, max_tokens // 10)
    cost_min = pricing.cost_for_tokens(
        input_tokens=input_tokens,
        output_tokens=min_output,
        thinking_tokens=thinking_budget // 2,
    )

    # Max cost: full max_tokens response
    cost_max = pricing.cost_for_tokens(
        input_tokens=input_tokens,
        output_tokens=max_tokens,
        thinking_tokens=thinking_budget,
    )

    # Generate recommendation
```

```
    if cost_max > config.budget_hard_limit_usd:
        recommendation = "BLOCK: estimated cost exceeds hard limit"
    elif cost_max > config.budget_alert_threshold_usd:
        recommendation = "WARN: estimated cost exceeds alert threshold"
    elif input_tokens > 50_000:
        recommendation = "REVIEW: large context — consider truncation or RAG"
    else:
        recommendation = "OK"

    return PreflightEstimate(
        model=model,
        estimated_input_tokens=input_tokens,
        max_output_tokens=max_tokens,
        estimated_cost_min=cost_min,
        estimated_cost_max=cost_max,
        recommendation=recommendation,
    )
```

This preflight check is lightweight — it runs locally, no API calls — and gives you a cost estimate before you commit to the request. In Chapter 3, we'll wrap this into the metering middleware so every request gets a pre-flight check automatically.

---

## The hidden cost of conversation history

One pattern that catches teams off guard is the growing cost of conversation history. In a multi-turn conversation, each new message includes the entire history. The cost of each turn grows linearly:

```
Turn 1:  system(200) + user(100)                          = 300 input tokens
Turn 2:  system(200) + user(100) + assistant(400) + user(100) = 800 input tokens
Turn 3:  system(200) + user(100) + assistant(400) + user(100) + assistant(500) +
user(150) = 1,450 input tokens
Turn 10: system(200) + accumulated history                 = ~5,000+ input tokens
```

Over a 10-turn conversation, you've paid for the system prompt 10 times, the first user message 9 times, and so on. The total cost of the conversation is not the sum of individual messages — it's the sum of *all prefixes* of the conversation.

A 20-turn conversation with an average of 300 tokens per message costs as much as sending the *entire* conversation history 10 times over. This is why conversation history management — truncation, summarization, and caching — has an outsized impact on cost.

```
"""Example: Cost growth in multi-turn conversations."""

from tokenwatch.config import TokenwatchConfig

config = TokenwatchConfig.from_yaml("pricing.yaml")
sonnet = config.get_model("claude-sonnet-4-20250514")
```

```
system_tokens = 200
avg_tokens_per_message = 300
num_turns = 20

total_input_tokens = 0
total_output_tokens = 0

for turn in range(1, num_turns + 1):
    # Input: system + all previous messages + current user message
    history_tokens = system_tokens + (turn * 2 - 1) * avg_tokens_per_message
    total_input_tokens += history_tokens
    total_output_tokens += avg_tokens_per_message  # assistant response

cost = sonnet.cost_for_tokens(
    input_tokens=total_input_tokens,
    output_tokens=total_output_tokens,
)
print(f"20-turn conversation cost: ${cost:.4f}")
print(f"Total input tokens processed: {total_input_tokens:,}")
print(f"Average input tokens per turn: {total_input_tokens // num_turns:,}")
```

---

## Common mistakes

**1. Confusing characters with tokens.** A 1,000-character prompt is not 1,000 tokens. For English text it's roughly 250 tokens; for JSON it might be 400+; for Chinese text it could be 500+. Always count tokens, not characters.

**2. Ignoring output token cost.** Output tokens are 3-5x more expensive than input tokens. A feature that generates verbose responses (2,000-token outputs) costs dramatically more than one that generates concise responses (100-token outputs), even with identical input.

**3. Using large context windows by default.** "The model supports 200K tokens, so let's send everything" is a costly mindset. Context window size is a ceiling, not a target. Send the minimum context needed for the task.

**4. Forgetting conversation history growth.** Each turn in a multi-turn conversation re-sends the entire history. A 20-turn conversation can cost 10x what you'd expect if you only count individual messages. The failure mode is subtle: your per-request cost dashboard looks fine because early turns are cheap. But by turn 15, a single request can cost as much as the first 10 turns combined. The fix is conversation-level cost tracking (which the `ConversationCostTracker` exercise below asks you to build) and history truncation strategies (Chapter 5).

**5. Treating token counts as static.** A prompt that costs $0.001 today will cost a different amount when you switch models, when the provider changes pricing, or when your input data changes shape. Always compute costs dynamically from `pricing.yaml` rather than hardcoding dollar amounts in your codebase.

---

### Exercises

**Easy:** Use `estimate_tokens` to compare the token count of the same information expressed as prose vs. JSON vs. XML. Which format is the most token-efficient? Which is the least?

**Medium:** Build a `ConversationCostTracker` that tracks the cumulative cost of a multi-turn conversation. It should accept messages one at a time, count the tokens for each turn (including the growing history), and report the total cost, the cost of the most recent turn, and the projected cost if the conversation continues for N more turns at the current average.

**Hard:** Write a `ContextBudget` class that takes a maximum input token budget and a list of context sources (system prompt, conversation history, RAG results, user message), each with a priority. It should allocate the token budget across sources by priority, truncating lower-priority sources first. If the budget is tight, it should warn the caller about what was truncated.

---

### Summary

In this chapter you learned:

- **Tokenization mechanics** — how BPE tokenizers convert text to tokens, and why token counts vary by language, format, and content type
- **Token class economics** — input, output, cached, and thinking tokens are billed at different rates, creating asymmetries you can exploit for optimization
- **Context window costs** — larger contexts cost more per request, and conversation history causes quadratic cost growth across turns
- **Pre-flight estimation** — counting tokens before sending a request lets you estimate cost, choose models, and enforce budgets proactively
- **The tokenwatch/tokens module** — local token counting and estimation for both Anthropic and OpenAI models

#### Next chapter

You can count tokens and estimate costs — but estimates are not actuals. Chapter 3 builds the metering middleware: a decorator and context manager that wraps every LLM call, records actual token usage from the API response, calculates real costs, and stores them for analysis. This is where tokenwatch becomes a real observability tool.

---

A> **Code for this chapter:** `project/cap02/tokenwatch/`

## Chapter 3 — Measuring First: Token Metering and Cost Attribution

You can't optimize what you can't measure. That sentence has become a cliche in software engineering, but it's especially true for LLM costs because the metrics you need don't exist by default. Your cloud provider gives you a monthly bill. Your LLM API gives you per-request token counts buried in response headers. Nobody gives you "cost per feature" or "tokens per user" out of the box. You have to build that yourself.

This chapter builds the core of tokenwatch: a metering middleware that wraps every LLM call, extracts token usage from the API response, calculates real costs using your pricing configuration, tags the usage with metadata (feature, user, team), and stores everything for analysis. By the end, every LLM call in your system will be instrumented, and you'll have the data foundation for every optimization in the rest of the book.

**Where this fits in your cost journey**
Metering is the first chapter in the optimization sequence for a reason: every technique in the rest of this book depends on having accurate, per-request cost data. Prompt compression (Chapter 5) needs baseline token counts to measure improvement. Caching (Chapter 6) needs hit-rate metrics to prove value. Model routing (Chapter 8) needs per-feature cost distributions to calibrate complexity thresholds. Governance (Chapter 12) needs real-time spend aggregates to enforce budgets. And forecasting (Chapter 13) needs months of historical data to project future spend with any confidence. Without metering, you are guessing. With metering, you are engineering. The investment you make in this chapter pays compound returns across every subsequent optimization.

---

## What to measure

Before writing code, let's define exactly what data we need from every LLM request:

**Request metadata:** - Timestamp - Model used - Feature or endpoint name (e.g., "ticket-classifier", "chat-agent") - User or tenant ID - Team or cost center - Request ID (for correlation)

**Token usage (from API response):** - Input tokens - Output tokens - Cache creation input tokens (Anthropic: tokens written to cache) - Cache read input tokens (Anthropic: tokens read from cache) - Thinking tokens (if extended thinking was used)

**Computed values:** - Cost in USD (calculated from token counts and pricing config) - Latency (time from request to response) - Whether the request was a cache hit/miss - Whether the request was batch or synchronous

This data, collected consistently across every request, gives you complete cost observability. Note that the list above includes token classes that many teams initially overlook: cache creation tokens, cache read tokens, and thinking tokens. Cache tokens become essential once you enable prompt caching in Chapter 6 — if you do not meter them from day one, you will have to retrofit your data model later and lose weeks of historical comparison data. Thinking tokens matter because extended thinking (used with claude-sonnet-4-6 and claude-opus-4-6) can consume thousands of output tokens that never appear in the visible response. If you only track visible output tokens, your cost calculations will undercount significantly on any request that uses extended thinking.

Let's build the system to collect it.

**Why not just use the provider dashboard?**
Both Anthropic and OpenAI provide usage dashboards. They show total tokens and total cost per day. So why build your own metering? Three reasons:

**1. No attribution.** Provider dashboards show aggregate spend across your entire API key. They cannot tell you which feature, user, or team drove the cost. If your bill doubles overnight, the dashboard tells you "you spent more tokens." It cannot tell you "the chat-agent feature had a prompt regression that tripled input tokens."

**2. No per-request granularity.** Provider dashboards aggregate by day. You cannot drill down to individual requests, correlate cost with latency, or identify outlier requests. The metering system we build here records every request individually, enabling percentile analysis (Chapter 4) and anomaly detection (Chapter 12).

**3. No real-time enforcement.** Provider dashboards are retrospective — you see yesterday's costs today. Budget enforcement requires real-time cost tracking that happens at request time, not after the billing cycle closes.

**Real-world scenario: the invisible regression**

At a large fintech, an engineer updated a system prompt for a ticket classifier, adding detailed category descriptions "for better accuracy." The prompt went from 200 tokens to 1,800 tokens. The quality improvement was marginal. The cost impact was a 9x increase in input tokens per request, multiplied across 150,000 daily requests. Without per-request metering, this regression was invisible for two weeks — hidden inside a monthly bill that nobody reviewed until the budget alert fired. With the metering middleware we build in this chapter, the regression would have been caught within the first hour: the `cost_per_request` metric for that feature would spike immediately.

---

## The usage record

First, we define the data structure for a single metered request:

```
"""tokenwatch/meter.py — Token metering and cost attribution."""

import time
import uuid
from dataclasses import dataclass, field
from datetime import datetime, timezone
from typing import Any

from tokenwatch.config import TokenwatchConfig


@dataclass
class UsageRecord:
    """A single metered LLM request with cost attribution."""

    # Identity
    request_id: str = field(default_factory=lambda: str(uuid.uuid4()))
    timestamp: str = field(
        default_factory=lambda: datetime.now(timezone.utc).isoformat()
    )
```

```
    # Attribution
    model: str = ""
    provider: str = ""
    feature: str = ""        # e.g., "ticket-classifier"
    user_id: str = ""        # who triggered the request
    team: str = ""           # cost center
    environment: str = ""    # "production", "staging", "development"

    # Token usage (from API response)
    input_tokens: int = 0
    output_tokens: int = 0
    cache_creation_tokens: int = 0
    cache_read_tokens: int = 0
    thinking_tokens: int = 0

    # Computed
    cost_usd: float = 0.0
    latency_ms: float = 0.0
    is_batch: bool = False
    is_cache_hit: bool = False

    # Raw response metadata (for debugging)
    stop_reason: str = ""
    raw_usage: dict = field(default_factory=dict)

    def to_dict(self) -> dict:
        """Convert to a dictionary for storage/serialization."""
        return {
            "request_id": self.request_id,
            "timestamp": self.timestamp,
            "model": self.model,
            "provider": self.provider,
            "feature": self.feature,
            "user_id": self.user_id,
            "team": self.team,
            "environment": self.environment,
            "input_tokens": self.input_tokens,
            "output_tokens": self.output_tokens,
            "cache_creation_tokens": self.cache_creation_tokens,
            "cache_read_tokens": self.cache_read_tokens,
            "thinking_tokens": self.thinking_tokens,
            "cost_usd": self.cost_usd,
            "latency_ms": self.latency_ms,
            "is_batch": self.is_batch,
            "is_cache_hit": self.is_cache_hit,
            "stop_reason": self.stop_reason,
        }
```

## The usage store

We need somewhere to store usage records. For development and small-scale production, a JSON-lines file works well. For larger systems, you'd swap this for a database or event stream — but the interface stays the same.

```
"""tokenwatch/store.py — Usage record storage."""

import json
from pathlib import Path
from datetime import datetime, timezone
from typing import Iterator

from tokenwatch.meter import UsageRecord


class UsageStore:
    """Append-only store for usage records.

    Uses JSON Lines format (.jsonl) — one JSON object per line.
    Easy to append, easy to process with standard tools.
    """

    def __init__(self, path: str | Path = "tokenwatch_usage.jsonl"):
        self.path = Path(path)
        self.path.parent.mkdir(parents=True, exist_ok=True)

    def record(self, usage: UsageRecord) -> None:
        """Append a usage record to the store."""
        with open(self.path, "a") as f:
            f.write(json.dumps(usage.to_dict()) + "\n")

    def query(
        self,
        feature: str | None = None,
        model: str | None = None,
        user_id: str | None = None,
        team: str | None = None,
        since: str | None = None,
    ) -> list[UsageRecord]:
        """Query usage records with optional filters.

        Args:
            feature: Filter by feature name
            model: Filter by model ID
            user_id: Filter by user ID
            team: Filter by team/cost center
            since: ISO timestamp — only records after this time
        """
        if not self.path.exists():
```

```
        return []

    records = []
    for line in self._read_lines():
        data = json.loads(line)

        if feature and data.get("feature") != feature:
            continue
        if model and data.get("model") != model:
            continue
        if user_id and data.get("user_id") != user_id:
            continue
        if team and data.get("team") != team:
            continue
        if since and data.get("timestamp", "") < since:
            continue

        record = UsageRecord(**{
            k: v for k, v in data.items()
            if k in UsageRecord.__dataclass_fields__
        })
        records.append(record)

    return records

def total_cost(self, **filters) -> float:
    """Get total cost matching the given filters."""
    records = self.query(**filters)
    return sum(r.cost_usd for r in records)

def _read_lines(self) -> Iterator[str]:
    """Read lines from the store file."""
    with open(self.path) as f:
        for line in f:
            line = line.strip()
            if line:
                yield line

def summary(self, **filters) -> dict:
    """Generate a summary of usage matching filters."""
    records = self.query(**filters)
    if not records:
        return {"total_records": 0, "total_cost_usd": 0.0}

    total_cost = sum(r.cost_usd for r in records)
    total_input = sum(r.input_tokens for r in records)
    total_output = sum(r.output_tokens for r in records)
    total_cached = sum(r.cache_read_tokens for r in records)
```

```
        by_feature: dict[str, float] = {}
        by_model: dict[str, float] = {}
        for r in records:
            by_feature[r.feature] = by_feature.get(r.feature, 0) + r.cost_usd
            by_model[r.model] = by_model.get(r.model, 0) + r.cost_usd

        return {
            "total_records": len(records),
            "total_cost_usd": round(total_cost, 6),
            "total_input_tokens": total_input,
            "total_output_tokens": total_output,
            "total_cached_tokens": total_cached,
            "avg_cost_per_request": round(total_cost / len(records), 6),
            "cache_hit_rate": sum(1 for r in records if r.is_cache_hit) /
len(records),
            "cost_by_feature": by_feature,
            "cost_by_model": by_model,
        }
```

**Design decisions in the UsageStore**

The `UsageStore` uses JSON Lines (`.jsonl`) format — one JSON object per line — rather than a database. This is a deliberate trade-off:

- **Append-only writes** are fast and never block the critical path. Your LLM API call latency is 200-2000ms; appending a line to a file takes microseconds.
- **Standard tooling** works immediately: `wc -l` counts requests, `jq` filters and aggregates, and any data pipeline can ingest JSONL.
- **No schema migrations.** When you add a new field to `UsageRecord`, old records simply lack that field. No migration scripts needed.

The trade-off is query performance: scanning a large JSONL file is O(n). For development and moderate production (under 1M records), this is fine. For high-volume production, you would swap the store backend to a database (PostgreSQL, ClickHouse) or an event stream (Kafka) while keeping the same `UsageStore` interface. The `query()` and `summary()` methods would become SQL queries, but the calling code would not change. This is why the store is a separate module from the meter — the boundary is designed for exactly this evolution.

A question that comes up frequently in production is whether the metering store should live on the critical path of the API call. The answer is: the write must be fast, but it does not need to be synchronous with the response. In tokenwatch, the `record()` method appends a single line to a file, which takes microseconds and will never block the response. In a more complex deployment, you might push the record to an in-memory queue (a Python `deque` or a Redis stream) and flush to durable storage asynchronously. The key design principle is that metering should never add measurable latency to the LLM call itself. If your metering layer adds even 10 milliseconds, engineers will start finding ways to bypass it, and you will lose the visibility that makes everything else in this book possible.

---

## The metering middleware

Now the centerpiece: a wrapper that intercepts LLM API calls, measures everything, and records the usage. We'll build it as both a context manager and a decorator, so you can use whichever pattern fits your code.

```
"""tokenwatch/middleware.py — Metering middleware for LLM API calls."""

import time
import functools
from typing import Any, Callable

import anthropic
import openai

from tokenwatch.config import TokenwatchConfig
from tokenwatch.meter import UsageRecord
from tokenwatch.store import UsageStore


class TokenMeter:
    """Metering middleware for LLM API calls.

    Wraps Anthropic and OpenAI clients to automatically record
    token usage and costs for every request.
    """

    def __init__(
        self,
        config: TokenwatchConfig,
        store: UsageStore | None = None,
        default_feature: str = "unknown",
        default_team: str = "unknown",
        environment: str = "development",
    ):
        self.config = config
        self.store = store or UsageStore()
        self.default_feature = default_feature
        self.default_team = default_team
        self.environment = environment

    def measure_anthropic(
        self,
        response: anthropic.types.Message,
        latency_ms: float,
        feature: str | None = None,
        user_id: str = "",
        team: str | None = None,
    ) -> UsageRecord:
        """Extract usage from an Anthropic API response and record it."""
```

```
usage = response.usage
model = response.model

# Anthropic reports cache tokens in the usage object
cache_creation = getattr(usage, "cache_creation_input_tokens", 0) or 0
cache_read = getattr(usage, "cache_read_input_tokens", 0) or 0

# Input tokens from usage minus cache tokens
input_tokens = usage.input_tokens - cache_read

# Check for thinking tokens (extended thinking)
thinking_tokens = 0
for block in response.content:
    if getattr(block, "type", "") == "thinking":
        # Thinking tokens are counted in output_tokens by the API
        # We separate them for cost analysis
        thinking_text = getattr(block, "thinking", "")
        thinking_tokens += len(thinking_text) // 4  # rough estimate

pricing = self.config.get_model(model)
cost = pricing.cost_for_tokens(
    input_tokens=input_tokens,
    output_tokens=usage.output_tokens,
    cached_tokens=cache_read,
    thinking_tokens=thinking_tokens,
)

record = UsageRecord(
    model=model,
    provider="anthropic",
    feature=feature or self.default_feature,
    user_id=user_id,
    team=team or self.default_team,
    environment=self.environment,
    input_tokens=input_tokens,
    output_tokens=usage.output_tokens,
    cache_creation_tokens=cache_creation,
    cache_read_tokens=cache_read,
    thinking_tokens=thinking_tokens,
    cost_usd=cost,
    latency_ms=latency_ms,
    is_cache_hit=cache_read > 0,
    stop_reason=response.stop_reason or "",
    raw_usage={
        "input_tokens": usage.input_tokens,
        "output_tokens": usage.output_tokens,
        "cache_creation_input_tokens": cache_creation,
        "cache_read_input_tokens": cache_read,
    },
```

```
        )

        self.store.record(record)
        return record

    def measure_openai(
        self,
        response: Any,  # openai.types.chat.ChatCompletion
        latency_ms: float,
        feature: str | None = None,
        user_id: str = "",
        team: str | None = None,
    ) -> UsageRecord:
        """Extract usage from an OpenAI API response and record it."""
        usage = response.usage
        model = response.model

        # OpenAI cached tokens
        cached = 0
        if hasattr(usage, "prompt_tokens_details") and
usage.prompt_tokens_details:
            cached = getattr(usage.prompt_tokens_details, "cached_tokens", 0) or
0

        input_tokens = usage.prompt_tokens - cached

        # Reasoning tokens (o-series models)
        thinking_tokens = 0
        if hasattr(usage, "completion_tokens_details") and
usage.completion_tokens_details:
            thinking_tokens = getattr(
                usage.completion_tokens_details, "reasoning_tokens", 0
            ) or 0

        try:
            pricing = self.config.get_model(model)
        except KeyError:
            # Model not in config — record with zero cost
            pricing = None

        cost = 0.0
        if pricing:
            cost = pricing.cost_for_tokens(
                input_tokens=input_tokens,
                output_tokens=usage.completion_tokens,
                cached_tokens=cached,
                thinking_tokens=thinking_tokens,
            )
```

```
        record = UsageRecord(
            model=model,
            provider="openai",
            feature=feature or self.default_feature,
            user_id=user_id,
            team=team or self.default_team,
            environment=self.environment,
            input_tokens=input_tokens,
            output_tokens=usage.completion_tokens,
            cache_read_tokens=cached,
            thinking_tokens=thinking_tokens,
            cost_usd=cost,
            latency_ms=latency_ms,
            is_cache_hit=cached > 0,
            stop_reason=response.choices[0].finish_reason if response.choices
else "",
            raw_usage={
                "prompt_tokens": usage.prompt_tokens,
                "completion_tokens": usage.completion_tokens,
                "total_tokens": usage.total_tokens,
                "cached_tokens": cached,
            },
        )

        self.store.record(record)
        return record
```

---

## Wrapping real API calls

Now let's create convenience functions that wrap the native API clients with metering. These are the functions you'll use throughout your application instead of calling the APIs directly:

```
"""tokenwatch/client.py — Metered LLM client wrappers."""

import time
from typing import Any

import anthropic
import openai

from tokenwatch.config import TokenwatchConfig
from tokenwatch.middleware import TokenMeter
from tokenwatch.store import UsageStore


class MeteredAnthropic:
    """Anthropic client with automatic token metering."""

    def __init__(self, meter: TokenMeter, client: anthropic.Anthropic | None =
```

```
None):
        self.client = client or anthropic.Anthropic()
        self.meter = meter

    def create_message(
        self,
        feature: str = "",
        user_id: str = "",
        team: str = "",
        **kwargs: Any,
    ) -> tuple[anthropic.types.Message, "UsageRecord"]:
        """Send a message and automatically record usage.

        Accepts all standard anthropic.messages.create() parameters
        plus metering metadata (feature, user_id, team).

        Returns (response, usage_record) tuple.
        """
        start = time.monotonic()
        response = self.client.messages.create(**kwargs)
        latency_ms = (time.monotonic() - start) * 1000

        record = self.meter.measure_anthropic(
            response=response,
            latency_ms=latency_ms,
            feature=feature,
            user_id=user_id,
            team=team,
        )

        return response, record


class MeteredOpenAI:
    """OpenAI client with automatic token metering."""

    def __init__(self, meter: TokenMeter, client: openai.OpenAI | None = None):
        self.client = client or openai.OpenAI()
        self.meter = meter

    def create_chat(
        self,
        feature: str = "",
        user_id: str = "",
        team: str = "",
        **kwargs: Any,
    ) -> tuple[Any, "UsageRecord"]:
        """Send a chat completion and automatically record usage.
```

```
        Accepts all standard openai.chat.completions.create() parameters
        plus metering metadata.

        Returns (response, usage_record) tuple.
        """
        start = time.monotonic()
        response = self.client.chat.completions.create(**kwargs)
        latency_ms = (time.monotonic() - start) * 1000

        record = self.meter.measure_openai(
            response=response,
            latency_ms=latency_ms,
            feature=feature,
            user_id=user_id,
            team=team,
        )

        return response, record
```

Here's how you use the metered clients in practice:

```
"""example_metering.py — Using metered clients."""

from tokenwatch.config import TokenwatchConfig
from tokenwatch.middleware import TokenMeter
from tokenwatch.store import UsageStore
from tokenwatch.client import MeteredAnthropic

# Initialize
config = TokenwatchConfig.from_yaml("pricing.yaml")
store = UsageStore("usage_data/usage.jsonl")
meter = TokenMeter(config, store, environment="production")
client = MeteredAnthropic(meter)

# Make a metered API call
response, record = client.create_message(
    feature="ticket-classifier",
    user_id="user-123",
    team="support-platform",
    model="claude-sonnet-4-20250514",
    max_tokens=100,
    messages=[{
        "role": "user",
        "content": "Classify this support ticket: 'My card was charged twice'"
    }],
)

# The response works exactly like a normal API response
print(f"Classification: {response.content[0].text}")
```

```
# But now you also have cost data
print(f"Cost: ${record.cost_usd:.6f}")
print(f"Input tokens: {record.input_tokens}")
print(f"Output tokens: {record.output_tokens}")
print(f"Latency: {record.latency_ms:.0f}ms")

# And it's stored for later analysis
summary = store.summary(feature="ticket-classifier")
print(f"\nTotal classifier cost today: ${summary['total_cost_usd']:.4f}")
print(f"Requests: {summary['total_records']}")
print(f"Avg cost/request: ${summary['avg_cost_per_request']:.6f}")
```

T> **Tip:** The metered client returns a tuple `(response, record)` instead of just the response. This is intentional — it makes the cost visible at the call site. If you find this ergonomically awkward, you can access the most recent record from the store, but I recommend keeping it explicit.

---

## Cost attribution: tagging every request

The power of the metering system comes from the attribution tags. By consistently tagging every request with `feature`, `user_id`, and `team`, you can answer questions that a raw API bill never could:

- "Which feature costs the most?" → `store.summary(feature="...")` for each feature
- "Which team drives the most spend?" → `store.summary(team="...")`
- "What's the cost per user?" → aggregate by `user_id`
- "Is cost growing faster than usage?" → trend analysis on `cost_usd` vs request count

The key discipline is **consistent tagging**. Every API call must include these tags. If you have a call with `feature="unknown"`, you've lost visibility into that cost. In Chapter 12 on governance, we'll build enforcement mechanisms that reject untagged requests.

```
"""tokenwatch/attribution.py — Cost attribution analysis."""

from collections import defaultdict
from tokenwatch.store import UsageStore


def cost_by_dimension(
    store: UsageStore,
    dimension: str,
    **filters,
) -> dict[str, dict]:
    """Break down costs by a given dimension.

    Args:
        dimension: "feature", "model", "user_id", or "team"
    """
    records = store.query(**filters)
    breakdown: dict[str, dict] = defaultdict(lambda: {
```

```
        "total_cost": 0.0,
        "request_count": 0,
        "total_input_tokens": 0,
        "total_output_tokens": 0,
        "avg_latency_ms": 0.0,
    })

    for r in records:
        key = getattr(r, dimension, "unknown")
        entry = breakdown[key]
        entry["total_cost"] += r.cost_usd
        entry["request_count"] += 1
        entry["total_input_tokens"] += r.input_tokens
        entry["total_output_tokens"] += r.output_tokens
        entry["avg_latency_ms"] += r.latency_ms

    # Compute averages
    for key, entry in breakdown.items():
        count = entry["request_count"]
        entry["avg_cost_per_request"] = entry["total_cost"] / count
        entry["avg_latency_ms"] = entry["avg_latency_ms"] / count

    return dict(sorted(
        breakdown.items(),
        key=lambda x: x[1]["total_cost"],
        reverse=True,
    ))
```

---

## Common mistakes

**1. Not tagging requests consistently.** If 20% of your requests have `feature="unknown"`, you've lost visibility into 20% of your spend. Enforce tagging at the middleware level — reject or warn on untagged requests. The failure mode: your cost dashboard shows $12,000/month under "unknown", and when leadership asks what it is, nobody knows. The fix is to make the `feature` parameter required in your metered client wrapper and fail loudly on empty strings during development.

**2. Calculating costs from estimates instead of actuals.** Token estimates are useful for pre-flight checks, but your cost records should always use the actual token counts from the API response. The API response is the ground truth. The failure mode: your estimates undercount by 15% (common with complex message structures), and over a month your reported spend diverges from the actual bill by thousands of dollars. The `TokenMeter.measure_anthropic` method reads tokens directly from `response.usage` for this reason.

**3. Storing only aggregates.** It's tempting to just increment counters — "total tokens today: 1.2M." But you lose the ability to drill down. Store individual records; aggregate at query time. Disk is cheap; lost visibility is expensive. The failure mode: you see a cost spike but cannot identify which requests caused it, which feature changed, or which user triggered the anomaly.

**4. Ignoring cache tokens in cost calculations.** If you count `input_tokens` from the API response without subtracting cached tokens, you'll double-count. Anthropic's `usage.input_tokens` includes cached tokens — you need to separate them for accurate cost calculation. Look at the `measure_anthropic` method: it computes `input_tokens = usage.input_tokens - cache_read` to avoid this trap.

**5. Not recording failures.** A request that fails after the model processes 10,000 input tokens still costs money — the provider bills for the processing, even if the response is an error. Track failed requests separately so you can account for their cost and identify retry storms that multiply spend.

**6. Deferring metering until "later."** Teams often plan to add metering "once we have real traffic." By the time real traffic arrives, the system has been in production for months, costs are climbing, and retrofitting metering requires touching every call site. Build metering into your first API call. The `MeteredAnthropic` and `MeteredOpenAI` wrappers are designed as drop-in replacements for the native clients — the migration cost at day one is near zero, while the migration cost at month six is significant refactoring.

---

## Exercises

**Easy:** Instrument a simple script that makes 5 API calls to Claude with different prompt lengths. Use the metered client and then call `store.summary()` to see the aggregate cost. Verify that the total matches the sum of individual record costs. Pay close attention to how input token counts change as you vary prompt length — you should see a roughly linear relationship between prompt character count and input tokens, at a ratio of approximately 4 characters per token for English text. This exercise builds intuition for how prompt length translates to cost, which you will need when optimizing prompts in Chapter 5.

**Medium:** Build a `CostReport` class that takes a `UsageStore` and generates a formatted text report showing: top 5 features by cost, top 5 models by cost, total cost for the period, and the average cost per request. Include a comparison to the previous period (yesterday vs. today, or this week vs. last week).

**Hard:** Create a real-time cost monitor that watches the usage store file for new records (using file polling or `watchdog`) and prints a live dashboard to the terminal showing: running total cost, requests per minute, average cost per request, and alerts when cost exceeds a threshold. Use the `rich` library for a formatted terminal display.

---

## Summary

In this chapter you learned:

- **What to measure** — every LLM request should record token counts by class, cost, latency, and attribution metadata (feature, user, team)
- **The UsageRecord data model** — a comprehensive record of a single metered request, designed for both storage and analysis

- **The UsageStore** — a simple but effective append-only store using JSON Lines format, with filtering and aggregation
- **The TokenMeter middleware** — extracts actual token usage from Anthropic and OpenAI API responses and calculates costs against your pricing config
- **Metered clients** — drop-in replacements for native API clients that automatically record every request
- **Cost attribution** — consistent tagging enables per-feature, per-user, and per-team cost analysis

**Next chapter**

You have cost data flowing. Now you need to make it meaningful. Chapter 4 covers unit economics: how to calculate cost per request, cost per user, cost per feature, and how to determine whether your AI features have healthy margins. This is where raw cost data becomes business intelligence.

---

A> **Code for this chapter:** `project/cap03/tokenwatch/`

# Chapter 4 — Unit Economics: Cost per Request, User, and Feature

Your metering system is recording every token, every dollar, every millisecond. But raw cost data is not useful by itself. Telling your VP of Engineering "we spent $47,000 on Claude last month" doesn't help anyone make decisions. What helps is: "our ticket classifier costs $0.003 per classification, our chat agent costs $0.12 per conversation, and the chat agent has negative margins on long conversations."

Unit economics transforms raw cost data into business metrics. It connects API spend to product value, reveals which features are profitable and which are subsidized, and gives engineering teams the language to discuss cost trade-offs with business stakeholders. This chapter builds the analytics layer on top of tokenwatch's metering data.

**Where this fits in your cost journey**

Chapter 3 gave you raw data: how many tokens each request consumed and what it cost. This chapter transforms that raw data into the metrics that drive business decisions. Without unit economics, cost conversations stay at the aggregate level ("we spent too much last month") and optimization effort is spread randomly. With unit economics, you can precisely identify which features need optimization (the ones with thin or negative margins), which users are expensive to serve (the power users whose conversations run to 50 turns), and which models are overqualified for their workloads (a topic that leads directly into model routing in Chapter 8). Unit economics is the analytical bridge between metering and optimization — it tells you where to focus the techniques from Chapters 5 through 11.

---

## Why unit economics matter

**Worked example: two features, same total cost, very different economics**
Consider two AI features that each cost $15,000 per month:

| Metric | Ticket classifier | Chat agent |
|---|---|---|
| Monthly cost | $15,000 | $15,000 |
| Monthly requests | 5,000,000 | 125,000 |
| Cost per request | $0.003 | $0.12 |
| Value per request | $0.50 (saved triage time) | $0.80 (saved agent time) |
| Gross margin | 99.4% | 85.0% |
| Monthly profit | $2,485,000 | $85,000 |

Both features cost the same in absolute terms, but their economics are completely different. The ticket classifier is incredibly efficient — it costs fractions of a cent per classification and displaces $0.50 of human labor each time. The chat agent is still profitable but has thinner margins and higher variance (some conversations run 20+ turns and cost $0.50+ each, eroding the margin).

Without unit economics, both features appear equally expensive. With unit economics, you can make informed decisions: the classifier needs no optimization — its margins are enormous. The chat agent needs attention — its P99 cost per conversation may be unprofitable, and that is where optimization effort should focus.

This is why unit economics matter more than aggregate cost tracking.

In traditional SaaS, the cost of serving a user is relatively fixed — a few database queries, some compute, network transfer. The marginal cost of one more user is tiny. AI features break this model. Each user interaction consumes tokens, and the cost can vary by 100x depending on the interaction complexity.

This creates three problems that unit economics solves:

**1. Pricing decisions.** If you're building an AI product, you need to know what each interaction costs to set prices that are sustainable. A chat agent that costs $0.15 per conversation needs to generate at least $0.15 in value (through subscriptions, reduced support costs, or revenue) to be viable.

**2. Feature prioritization.** Not all features are created equal. A feature that costs $50,000/month and saves $200,000/month in support costs has a 4x ROI. A feature that costs $30,000/month and produces "nice to have" summaries has uncertain ROI. Unit economics makes these comparisons concrete.

**3. Cost anomaly detection.** If your ticket classifier usually costs $0.003 per request and suddenly starts costing $0.03, something changed — a prompt got longer, a model was upgraded, or a bug is causing retries. Unit economics baselines make anomalies visible.

## The unit economics module

Let's build the analytics layer. It reads from the usage store and computes the metrics that matter:

```
"""tokenwatch/economics.py — Unit economics calculations."""

from dataclasses import dataclass
from collections import defaultdict
from datetime import datetime, timezone, timedelta

from tokenwatch.store import UsageStore
from tokenwatch.meter import UsageRecord


@dataclass
class FeatureEconomics:
    """Unit economics for a single feature."""
    feature: str
    total_cost_usd: float
    request_count: int
    unique_users: int
    cost_per_request: float
    cost_per_user: float
    avg_input_tokens: float
    avg_output_tokens: float
    avg_latency_ms: float
    cache_hit_rate: float
    p95_cost_per_request: float
    cost_trend_pct: float  # vs. previous period


@dataclass
class UserEconomics:
    """Unit economics for a single user."""
    user_id: str
    total_cost_usd: float
    request_count: int
    features_used: list[str]
    cost_per_request: float
    heaviest_feature: str
    heaviest_feature_cost: float


class EconomicsEngine:
    """Calculates unit economics from metered usage data."""

    def __init__(self, store: UsageStore):
        self.store = store
```

```
def feature_economics(
    self,
    since: str | None = None,
    compare_period: bool = True,
) -> list[FeatureEconomics]:
    """Calculate unit economics for each feature.

    Args:
        since: ISO timestamp for the start of the analysis period
        compare_period: If True, also calculate the previous
            period of the same length for trend comparison
    """
    records = self.store.query(since=since)
    if not records:
        return []

    # Group by feature
    by_feature: dict[str, list[UsageRecord]] = defaultdict(list)
    for r in records:
        by_feature[r.feature].append(r)

    # If comparing, get previous period records
    prev_costs: dict[str, float] = {}
    if compare_period and since:
        period_start = datetime.fromisoformat(since)
        period_length = datetime.now(timezone.utc) - period_start
        prev_start = (period_start - period_length).isoformat()
        prev_records = self.store.query(since=prev_start)
        prev_in_range = [
            r for r in prev_records
            if r.timestamp < since
        ]
        for r in prev_in_range:
            prev_costs[r.feature] = prev_costs.get(r.feature, 0) + r.cost_usd

    results = []
    for feature, feature_records in by_feature.items():
        costs = [r.cost_usd for r in feature_records]
        costs_sorted = sorted(costs)
        total_cost = sum(costs)
        count = len(feature_records)
        users = set(r.user_id for r in feature_records if r.user_id)

        # P95 cost
        p95_idx = int(count * 0.95)
        p95_cost = costs_sorted[min(p95_idx, count - 1)]

        # Cache hit rate
        cache_hits = sum(1 for r in feature_records if r.is_cache_hit)
```

```
            cache_rate = cache_hits / count if count > 0 else 0.0

            # Cost trend
            prev_cost = prev_costs.get(feature, 0.0)
            if prev_cost > 0:
                trend = ((total_cost - prev_cost) / prev_cost) * 100
            else:
                trend = 0.0

            results.append(FeatureEconomics(
                feature=feature,
                total_cost_usd=total_cost,
                request_count=count,
                unique_users=len(users),
                cost_per_request=total_cost / count,
                cost_per_user=total_cost / len(users) if users else total_cost,
                avg_input_tokens=sum(r.input_tokens for r in feature_records) /
count,
                avg_output_tokens=sum(r.output_tokens for r in feature_records) /
count,
                avg_latency_ms=sum(r.latency_ms for r in feature_records) /
count,
                cache_hit_rate=cache_rate,
                p95_cost_per_request=p95_cost,
                cost_trend_pct=trend,
            ))

        return sorted(results, key=lambda x: x.total_cost_usd, reverse=True)

    def user_economics(self, since: str | None = None) -> list[UserEconomics]:
        """Calculate unit economics per user."""
        records = self.store.query(since=since)
        if not records:
            return []

        by_user: dict[str, list[UsageRecord]] = defaultdict(list)
        for r in records:
            if r.user_id:
                by_user[r.user_id].append(r)

        results = []
        for user_id, user_records in by_user.items():
            total_cost = sum(r.cost_usd for r in user_records)
            count = len(user_records)

            # Find heaviest feature for this user
            feature_costs: dict[str, float] = defaultdict(float)
            for r in user_records:
                feature_costs[r.feature] += r.cost_usd
```

```
            heaviest = max(feature_costs.items(), key=lambda x: x[1])

            results.append(UserEconomics(
                user_id=user_id,
                total_cost_usd=total_cost,
                request_count=count,
                features_used=list(set(r.feature for r in user_records)),
                cost_per_request=total_cost / count,
                heaviest_feature=heaviest[0],
                heaviest_feature_cost=heaviest[1],
            ))

        return sorted(results, key=lambda x: x.total_cost_usd, reverse=True)
```

## Margin analysis: is your AI feature profitable?

The most important question in AI product economics is: does this feature generate more value than it costs? To answer it, you need to pair LLM cost data with revenue or value data from your business systems.

```
"""tokenwatch/margins.py — Margin analysis for AI features."""

from dataclasses import dataclass


@dataclass
class MarginAnalysis:
    """Margin analysis for a single feature."""
    feature: str
    llm_cost_per_unit: float    # cost per request/interaction
    revenue_per_unit: float      # revenue or value generated per unit
    other_costs_per_unit: float  # infra, labor, etc.
    gross_margin_pct: float
    contribution_margin_pct: float
    break_even_volume: int       # requests needed to break even on fixed costs
    monthly_volume: int
    monthly_profit_usd: float
    is_profitable: bool


def analyze_margins(
    feature: str,
    llm_cost_per_unit: float,
    revenue_per_unit: float,
    other_costs_per_unit: float = 0.0,
    fixed_costs_monthly: float = 0.0,
    monthly_volume: int = 0,
) -> MarginAnalysis:
```

```
    """Analyze the margins of an AI feature.

    Args:
        feature: Feature name
        llm_cost_per_unit: Average LLM cost per interaction (from economics
engine)
        revenue_per_unit: Revenue or value per interaction
            For revenue-generating features: actual revenue
            For cost-saving features: savings vs. human alternative
        other_costs_per_unit: Non-LLM variable costs (compute, storage, etc.)
        fixed_costs_monthly: Monthly fixed costs (engineering time, etc.)
        monthly_volume: Current monthly request volume
    """
    total_variable_cost = llm_cost_per_unit + other_costs_per_unit
    gross_margin = revenue_per_unit - llm_cost_per_unit
    contribution_margin = revenue_per_unit - total_variable_cost

    gross_margin_pct = (gross_margin / revenue_per_unit * 100) if
revenue_per_unit > 0 else -100
    contribution_margin_pct = (
        (contribution_margin / revenue_per_unit * 100)
        if revenue_per_unit > 0 else -100
    )

    # Break-even volume: fixed costs / contribution margin per unit
    if contribution_margin > 0:
        break_even = int(fixed_costs_monthly / contribution_margin)
    else:
        break_even = -1  # never breaks even

    monthly_profit = (contribution_margin * monthly_volume) - fixed_costs_monthly

    return MarginAnalysis(
        feature=feature,
        llm_cost_per_unit=llm_cost_per_unit,
        revenue_per_unit=revenue_per_unit,
        other_costs_per_unit=other_costs_per_unit,
        gross_margin_pct=gross_margin_pct,
        contribution_margin_pct=contribution_margin_pct,
        break_even_volume=break_even,
        monthly_volume=monthly_volume,
        monthly_profit_usd=monthly_profit,
        is_profitable=monthly_profit > 0,
    )
```

Here's how you'd use this in practice:

```
"""example_margins.py — Margin analysis for AI features."""

from tokenwatch.margins import analyze_margins
```

```python
# Feature 1: Ticket classifier — saves human triage time
classifier_margins = analyze_margins(
    feature="ticket-classifier",
    llm_cost_per_unit=0.003,          # $0.003 per classification
    revenue_per_unit=0.50,             # saves $0.50 in human triage time
    other_costs_per_unit=0.001,        # compute costs
    fixed_costs_monthly=2000.0,        # engineering maintenance
    monthly_volume=100_000,
)

print(f"Ticket Classifier:")
print(f"  Gross margin: {classifier_margins.gross_margin_pct:.1f}%")
print(f"  Monthly profit: ${classifier_margins.monthly_profit_usd:,.2f}")
print(f"  Break-even volume: {classifier_margins.break_even_volume:,} requests")
print(f"  Profitable: {classifier_margins.is_profitable}")

# Feature 2: Chat agent — complex, more expensive
chat_margins = analyze_margins(
    feature="chat-agent",
    llm_cost_per_unit=0.12,           # $0.12 per conversation
    revenue_per_unit=0.80,             # saves $0.80 vs. human agent
    other_costs_per_unit=0.02,         # compute, storage
    fixed_costs_monthly=5000.0,        # engineering maintenance
    monthly_volume=50_000,
)

print(f"\nChat Agent:")
print(f"  Gross margin: {chat_margins.gross_margin_pct:.1f}%")
print(f"  Monthly profit: ${chat_margins.monthly_profit_usd:,.2f}")
print(f"  Break-even volume: {chat_margins.break_even_volume:,} requests")
print(f"  Profitable: {chat_margins.is_profitable}")
```

T> **Tip:** For cost-saving features (replacing human work), the "revenue per unit" is the cost of the human alternative. If a human agent costs $0.80 per ticket resolution and your AI agent costs $0.12, the economic value is $0.80 per interaction. This framing is critical when presenting AI ROI to business stakeholders.

---

## Cost distribution analysis

Not all requests are equal. Understanding the distribution of costs — not just the average — is critical for optimization. If 5% of your requests account for 50% of your costs, you know exactly where to focus.

```python
"""tokenwatch/distribution.py — Cost distribution analysis."""

import math
from dataclasses import dataclass
from tokenwatch.store import UsageStore
```

```
@dataclass
class CostDistribution:
    """Statistical distribution of costs for a feature."""
    feature: str
    count: int
    mean: float
    median: float
    p50: float
    p90: float
    p95: float
    p99: float
    std_dev: float
    min_cost: float
    max_cost: float
    top_5pct_share: float  # % of total cost from top 5% of requests
    coefficient_of_variation: float  # std_dev / mean


def analyze_distribution(store: UsageStore, feature: str) -> CostDistribution:
    """Analyze the cost distribution for a feature."""
    records = store.query(feature=feature)
    if not records:
        raise ValueError(f"No records found for feature: {feature}")

    costs = sorted([r.cost_usd for r in records])
    n = len(costs)

    mean = sum(costs) / n
    median = costs[n // 2]
    variance = sum((c - mean) ** 2 for c in costs) / n
    std_dev = math.sqrt(variance)

    # Percentiles
    p50 = costs[int(n * 0.50)]
    p90 = costs[int(n * 0.90)]
    p95 = costs[int(n * 0.95)]
    p99 = costs[min(int(n * 0.99), n - 1)]

    # Top 5% share
    top_5pct_count = max(1, int(n * 0.05))
    top_5pct_cost = sum(costs[-top_5pct_count:])
    total_cost = sum(costs)
    top_5pct_share = (top_5pct_cost / total_cost * 100) if total_cost > 0 else 0

    cv = (std_dev / mean) if mean > 0 else 0

    return CostDistribution(
```

```
        feature=feature,
        count=n,
        mean=mean,
        median=median,
        p50=p50,
        p90=p90,
        p95=p95,
        p99=p99,
        std_dev=std_dev,
        min_cost=costs[0],
        max_cost=costs[-1],
        top_5pct_share=top_5pct_share,
        coefficient_of_variation=cv,
    )
```

The coefficient of variation (CV) is particularly telling. A CV close to 0 means all requests cost roughly the same — you're dealing with a predictable workload. A CV above 1.0 means high variance — some requests cost dramatically more than others. High-CV features are prime targets for model routing (Chapter 8), where you send cheap requests to cheap models and expensive requests to powerful models.

**How to use distributions in production**

In practice, the most actionable insight from distribution analysis is the top-5-percent share. If 5% of your requests account for 40% or more of your total cost, you have a long-tail problem. The right response depends on the root cause. If the expensive requests are long conversations (high input token counts from accumulated history), the fix is conversation truncation from Chapter 5. If they are complex reasoning tasks that genuinely require a powerful model, the fix is routing those requests to the appropriate model tier (Chapter 8) and routing the 95% of cheap requests to a cheaper model. If the expensive requests are retry storms from failed tool calls, the fix is better error handling and the agent budget controls from Chapter 10.

The `CostDistribution` dataclass captures the metrics you need to diagnose each of these patterns. Run `analyze_distribution` for every feature monthly and track how the P95, P99, and top-5-percent share evolve over time. A rising P99 signals a regression somewhere in your pipeline — either a prompt got longer, a model was changed, or user behavior shifted toward more complex queries. Catching these trends early, before they compound into a budget crisis, is the operational payoff of distribution analysis.

---

## Building the cost dashboard

Let's bring it all together into a report generator that produces a readable cost dashboard from your metering data:

```
"""tokenwatch/dashboard.py — Cost dashboard report generator."""

from tokenwatch.store import UsageStore
from tokenwatch.economics import EconomicsEngine
```

```
from tokenwatch.distribution import analyze_distribution


def generate_cost_report(
    store: UsageStore,
    since: str | None = None,
) -> str:
    """Generate a formatted cost report from usage data."""
    engine = EconomicsEngine(store)
    features = engine.feature_economics(since=since)

    if not features:
        return "No usage data found for the specified period."

    lines = []
    lines.append("=" * 70)
    lines.append("TOKENWATCH COST REPORT")
    lines.append("=" * 70)

    # Overall summary
    total_cost = sum(f.total_cost_usd for f in features)
    total_requests = sum(f.request_count for f in features)
    total_users = len(set(
        u for f in features
        for u in [f.feature]  # placeholder — real impl aggregates users
    ))

    lines.append(f"\nTotal cost:      ${total_cost:,.4f}")
    lines.append(f"Total requests:  {total_requests:,}")
    lines.append(f"Avg cost/request: ${total_cost/total_requests:.6f}")
    lines.append(f"Features tracked: {len(features)}")

    # Per-feature breakdown
    lines.append(f"\n{'Feature':<30} {'Cost':>10} {'Reqs':>8} {'$/Req':>10}
{'Cache%':>7} {'Trend':>8}")
    lines.append("-" * 73)

    for f in features:
        trend_str = f"+{f.cost_trend_pct:.0f}%" if f.cost_trend_pct > 0 else
f"{f.cost_trend_pct:.0f}%"
        lines.append(
            f"{f.feature:<30} "
            f"${f.total_cost_usd:>9.4f} "
            f"{f.request_count:>8,} "
            f"${f.cost_per_request:>9.6f} "
            f"{f.cache_hit_rate*100:>6.1f}% "
            f"{trend_str:>8}"
        )
```

```
    # Recommendations
    lines.append(f"\n{'RECOMMENDATIONS':=^70}")

    for f in features:
        if f.cache_hit_rate < 0.3 and f.request_count > 100:
            lines.append(
                f"  [CACHE] {f.feature}: cache hit rate is
{f.cache_hit_rate*100:.0f}% "
                f"— enable prompt caching (Chapter 6)"
            )
        if f.avg_input_tokens > 10_000:
            lines.append(
                f"  [CONTEXT] {f.feature}: avg {f.avg_input_tokens:.0f} input
tokens "
                f"— consider context compression (Chapter 5)"
            )
        if f.cost_trend_pct > 50:
            lines.append(
                f"  [TREND] {f.feature}: cost up {f.cost_trend_pct:.0f}% "
                f"— investigate for regressions"
            )

    lines.append("=" * 70)
    return "\n".join(lines)
```

W> **Warning:** The dashboard is a starting point for analysis, not a complete FinOps solution. For production use, you'll want to integrate with your existing dashboards (Grafana, Datadog, custom internal tools). The value of tokenwatch is in the data collection and the unit economics calculations — the visualization layer should match your team's existing tooling.

---

## Connecting cost to product metrics

The final step in unit economics is connecting LLM costs to product outcomes. This is where cost engineering meets product analytics. You need answers to questions like:

- "Do users who trigger the most LLM calls also have the highest retention?"
- "Do more expensive responses lead to better CSAT scores?"
- "Is the cost per resolution trending up or down as we improve the prompts?"

These questions require joining your tokenwatch data with your product analytics. The join key is usually `user_id` and `timestamp`. We won't build a full product analytics integration here — that's specific to your stack — but tokenwatch's consistent use of `user_id` and `feature` tags makes the join straightforward with any analytics tool.

The most common pattern is exporting the JSONL usage store to your data warehouse (BigQuery, Snowflake, Databricks) and joining it against your product events table. A simple query joining `tokenwatch.user_id = events.user_id` and aligning on date gives you cost-per-user alongside retention, conversion, and satisfaction metrics. Teams that perform this join regularly discover

surprising patterns: sometimes the most expensive users are also the most retained (the AI feature is genuinely valuable to them), and sometimes the most expensive users are bots or power users gaming a free tier. Both insights drive different actions, and neither is visible without connecting cost data to product data.

For teams that cannot easily export to a data warehouse, the `UserEconomics` dataclass provides a lightweight alternative. It surfaces the heaviest feature per user and cost-per-request by user, which often reveals the 80/20 pattern: a small fraction of users drives a disproportionate share of cost. This is enough to inform pricing decisions (should heavy users pay more?) and optimization targeting (which user segments benefit most from model routing?).

---

## Common mistakes

**1. Using averages without understanding distributions.** An average cost of $0.05 per request means nothing if the median is $0.01 and the 99th percentile is $2.50. Always look at the distribution, not just the mean. The failure mode: you set a per-request budget at 2x the average ($0.10), and it blocks 3% of legitimate requests that naturally fall in the long tail. Meanwhile, the P99 requests at $2.50 each account for 40% of total spend, and you have no visibility into them because you only track the average.

**2. Ignoring the cost of failed requests.** A request that fails after consuming 10,000 input tokens still costs money. If your error rate is 5% and failed requests are expensive (large context, retries), the cost of failure can be significant. Track failed request costs separately. At a large fintech, a retry loop on a RAG feature was silently re-sending 15,000-token requests up to 3 times on transient errors, tripling the effective cost of that feature.

**3. Treating LLM cost in isolation.** LLM API cost is often the majority of AI feature cost, but not all of it. There's also the embedding generation cost (for RAG), the infrastructure cost (servers running your middleware), and the engineering cost (maintaining prompts and models). Unit economics should include all variable costs, not just the API bill.

**4. Not segmenting by user behavior.** Power users who send 100 messages per day have different economics than casual users who send 2. If your pricing model doesn't account for this, you'll subsidize heavy users at the expense of light ones. The `UserEconomics` dataclass in our code exposes exactly this: `heaviest_feature` and `heaviest_feature_cost` per user reveal the power-user subsidy pattern.

**5. Treating margins as static.** Margins change when providers adjust pricing, when you optimize prompts, and when user behavior shifts. Recalculate margins monthly and after any prompt or model change. The `cost_trend_pct` field in `FeatureEconomics` exists to catch these shifts automatically.

---

## Exercises

**Easy:** Use the `EconomicsEngine` to analyze a sample usage store with at least 3 features. Print the top feature by total cost and the top feature by cost-per-request. Are they the same feature?

In most production systems, they are not — the feature with the highest total cost is typically a high-volume, low-cost-per-request feature, while the feature with the highest cost-per-request is a low-volume, complex feature. Understanding this distinction is the foundation of unit economics thinking.

**Medium:** Build a `CostAnomaly` detector that compares the current hour's average cost-per-request for each feature against the trailing 24-hour average. Flag any feature where the current hour is more than 2 standard deviations above the baseline. Test it by artificially inserting some high-cost records.

**Hard:** Create a `PricingSimulator` that takes the current usage data and simulates the impact of pricing changes. Given a set of proposed price changes (e.g., "input tokens go from $3 to $2 per million"), recalculate all costs and show the impact on total spend, per-feature margins, and break-even volumes. This is the tool you'd use before a provider price change goes into effect.

---

## Summary

In this chapter you learned:

- **Unit economics transforms raw data** — cost per request, cost per user, and cost per feature turn opaque API bills into actionable business metrics
- **Margin analysis** connects LLM costs to business value, revealing which features are profitable and which are subsidized
- **Cost distributions matter** — averages hide the long tail; P95/P99 costs and top-5% share reveal where optimization effort should focus
- **The tokenwatch economics engine** provides feature-level and user-level cost breakdowns with trend analysis
- **Cost dashboards** with recommendations automate the process of finding optimization opportunities

### Next chapter

You have visibility and business metrics. Now it's time to start optimizing. Chapter 5 tackles the first and often most impactful lever: prompt engineering for cost. You'll learn compression techniques, structured output optimization, and how to get the same quality from fewer tokens.

---

A> **Code for this chapter:** `project/cap04/tokenwatch/`

# Chapter 5 — Prompt Engineering for Cost

Most prompt engineering advice focuses on quality: how to get better, more accurate, more reliable responses from LLMs. This chapter focuses on a different dimension: how to get the *same quality* from fewer tokens. Prompt engineering for cost is not about making responses worse — it's about eliminating waste in your prompts that adds tokens without adding value.

The impact is often surprising. In production systems I've worked on, prompt compression and restructuring have reduced input token counts by 30-60% without measurable degradation in

output quality. When you multiply that by millions of daily requests, the savings are substantial. And unlike caching or model routing, prompt optimization has zero infrastructure complexity — it's pure engineering craft.

**Where this fits in your cost journey**

Prompt engineering for cost is the first optimization lever for a reason: it reduces the fundamental unit of cost (tokens per request) before any other technique is applied. Every token you eliminate from a prompt is a token that does not need to be cached (Chapter 6), does not need to be batched (Chapter 7), does not need to be routed (Chapter 8), and does not appear in your governance budget (Chapter 12). The savings compound through every subsequent layer of optimization. Moreover, prompt compression requires no infrastructure changes — no new services, no database migrations, no API key rotations. You edit text, test quality, and deploy. For teams under immediate cost pressure, this chapter delivers the fastest return on effort of any chapter in the book.

---

## The anatomy of a wasteful prompt

Let's look at a real-world prompt pattern that I've seen in multiple production systems. It's a support ticket classifier:

```
# BEFORE optimization — 847 tokens
WASTEFUL_PROMPT = """You are an expert customer support ticket classifier for our
company.
You have been trained on thousands of support tickets and you understand the
nuances
of customer complaints, questions, and requests. Your job is to carefully analyze
each
support ticket and classify it into exactly one of the following categories.
Please
read the ticket carefully and think step by step about which category best fits
the
ticket content. Here are the categories you can choose from:

Category 1: billing - This category is for any tickets related to billing,
payments,
charges, refunds, invoices, pricing, subscription fees, or any other financial
matters.

Category 2: technical - This category is for any tickets related to technical
issues,
bugs, errors, crashes, performance problems, integration issues, API problems, or
any
other technical difficulties.

Category 3: account - This category is for any tickets related to account
management,
login issues, password resets, profile updates, account settings, permissions, or
any
other account-related matters.
```

```
Category 4: feature_request - This category is for any tickets where the customer
is
requesting a new feature, suggesting an improvement, or asking for functionality
that
doesn't currently exist.

Category 5: general - This category is for any tickets that don't fit into any of
the
above categories, including general inquiries, feedback, or miscellaneous
requests.

Please classify the following support ticket and respond with ONLY the category
name
(billing, technical, account, feature_request, or general). Do not include any
explanation or additional text. Just the category name.

Support ticket to classify:
{ticket_text}

Your classification:"""
```

Now the optimized version:

```
# AFTER optimization — 168 tokens
OPTIMIZED_PROMPT = """Classify this support ticket into exactly one category.

Categories: billing, technical, account, feature_request, general

Respond with only the category name.

Ticket: {ticket_text}"""
```

Same task, same output quality (we tested this), 80% fewer input tokens. The wasteful prompt contains extensive preamble ("You are an expert…"), redundant category descriptions (the model already understands what "billing" means), and repeated instructions. Modern LLMs don't need this hand-holding. They need clear, concise instructions.

Why does the verbose version persist in so many codebases? Two reasons. First, early LLMs (GPT-3 era) genuinely performed better with elaborate instructions, and the habits formed during that period have not been updated. Second, engineers copy prompts from blog posts and tutorials that optimize for readability and explanation, not for production cost. A prompt that reads well in a tutorial ("You are a world-class expert in financial document analysis…") is actively wasteful at 200,000 requests per day. The discipline of prompt engineering for cost requires treating your system prompt the same way you treat production code: reviewed, tested, measured, and stripped of anything that does not contribute to correctness.

**Worked example: monthly savings from prompt compression**

Let's quantify the impact using `pricing.yaml` rates. Assume a ticket classifier running on Claude Sonnet (at the rate configured in `pricing.yaml`) at 100,000 requests per day:

```
Wasteful prompt: 847 tokens/request
  Daily input cost: 847 * 100,000 / 1,000,000 * $3.00 = $254.10
  Monthly input cost: $7,623

Optimized prompt: 168 tokens/request
  Daily input cost: 168 * 100,000 / 1,000,000 * $3.00 = $50.40
  Monthly input cost: $1,512

Monthly savings: $6,111 (80% reduction)
```

This is pure input-side savings — before any caching, routing, or batching. The `compare_prompts` function in `tokenwatch/prompt.py` automates exactly this calculation, and the tokenwatch `PromptComparison` dataclass tracks both token reduction and estimated monthly savings.

W> **Warning:** Prompt compression is not a universal "just remove words" exercise. Some instructions are genuinely necessary for quality — few-shot examples, specific formatting requirements, edge case handling. The goal is to remove *waste*, not *information*. Always test that output quality is maintained after compression.

---

## Compression techniques

Here are the techniques that consistently reduce token counts in production prompts, ranked by typical impact:

**1. Remove preamble and role-play**

The "You are an expert X who has been trained on Y and understands Z" pattern is the single biggest source of wasted tokens. Modern models perform the same whether you tell them they're an expert or not. In systematic testing across classification, extraction, and generation tasks, removing the preamble has no measurable impact on output quality.

Keep the system prompt functional: state what the model should do, not who it should pretend to be. Models like claude-sonnet-4-6 and claude-haiku-4-5 have been instruction-tuned to follow concise directives. Telling the model it is "an expert" does not activate any special capability — it simply consumes tokens. The only exceptions are role instructions that genuinely constrain the output style or domain (for example, "respond in formal legal language" is a legitimate stylistic constraint, not wasteful preamble).

```
# Before: 89 tokens
system_before = """You are a highly skilled data extraction specialist with years
of
experience in parsing financial documents. Your expertise includes identifying
key
financial metrics, extracting structured data from unstructured text, and
```

```
ensuring
accuracy in all your extractions."""

# After: 12 tokens
system_after = "Extract financial metrics from the provided document as JSON."
```

**2. Use structured output schemas**

Instead of describing the output format in prose, use JSON schema constraints. Both Anthropic and OpenAI support structured output — the model is constrained to produce valid JSON matching your schema, which eliminates the need for format instructions in the prompt.

```
"""tokenwatch/prompt.py — Prompt optimization utilities."""

import json
from typing import Any


def compress_system_prompt(
    original: str,
    max_tokens: int | None = None,
) -> str:
    """Apply basic compression heuristics to a system prompt.

    This is a semi-automated tool — it applies safe transformations
    and flags areas for manual review.
    """
    compressed = original

    # Remove common filler phrases
    filler_phrases = [
        "You are an expert ",
        "You are a highly skilled ",
        "You have been trained on ",
        "Your job is to carefully ",
        "Please read carefully and ",
        "Think step by step about ",
        "Here are the ",
        "Please note that ",
        "It is important to ",
        "Make sure to ",
        "Keep in mind that ",
    ]

    for phrase in filler_phrases:
        compressed = compressed.replace(phrase, "")

    # Collapse multiple spaces and newlines
    import re
    compressed = re.sub(r'\n{3,}', '\n\n', compressed)
```

```
    compressed = re.sub(r'  +', ' ', compressed)
    compressed = compressed.strip()

    return compressed


def build_efficient_prompt(
    task: str,
    context: str = "",
    output_format: str = "",
    constraints: list[str] | None = None,
    examples: list[dict[str, str]] | None = None,
    max_examples: int = 3,
) -> str:
    """Build a token-efficient prompt from structured components.

    Instead of writing prompts as free-form text (which tends to be
    verbose), this function constructs prompts from semantic components,
    producing a compact result.
    """
    parts = [task]

    if constraints:
        parts.append("Constraints: " + "; ".join(constraints))

    if output_format:
        parts.append(f"Output format: {output_format}")

    if examples:
        # Limit examples to control token count
        selected = examples[:max_examples]
        parts.append("Examples:")
        for ex in selected:
            parts.append(f"Input: {ex['input']}")
            parts.append(f"Output: {ex['output']}")

    if context:
        parts.append(f"Context: {context}")

    return "\n\n".join(parts)
```

**3. Truncate conversation history intelligently**

As we discussed in Chapter 2, conversation history grows linearly and is re-sent on every turn. The most common — and most wasteful — pattern is sending the entire history every time. Smart truncation keeps the context relevant without sending everything.

The cost arithmetic is stark. In a 20-turn conversation with an average of 200 tokens per message (user + assistant), the conversation history at turn 20 is approximately 4,000 tokens. Those 4,000 tokens are sent as input on every subsequent turn. If the conversation continues to turn 40, the

history reaches 8,000 tokens — per turn. Over the full conversation, the cumulative input tokens from history alone exceed 120,000. At Claude Sonnet rates (check `pricing.yaml`), that is over $0.36 in history-only input cost for a single conversation. Truncating the history to the most recent 4 turns caps the per-turn history cost and converts a quadratically growing cost curve into a roughly linear one. The `truncate_history` function below implements this with a configurable token budget and optional summarization of dropped messages.

```
"""tokenwatch/history.py — Conversation history management."""

from tokenwatch.tokens import estimate_tokens


def truncate_history(
    messages: list[dict],
    max_tokens: int = 4000,
    keep_system: bool = True,
    keep_last_n: int = 4,
    summarize_fn=None,
) -> list[dict]:
    """Truncate conversation history to fit within a token budget.

    Strategy:
    1. Always keep the last N messages (most recent context)
    2. Always keep the system message
    3. If over budget, drop oldest messages first
    4. If a summarize function is provided, replace dropped
       messages with a summary

    Args:
        messages: Full conversation history
        max_tokens: Maximum total tokens for the history
        keep_system: Whether to preserve system messages
        keep_last_n: Always keep this many recent messages
        summarize_fn: Optional function to summarize dropped messages
    """
    if not messages:
        return messages

    # Separate system messages and conversation messages
    system_msgs = [m for m in messages if m.get("role") == "system"]
    conv_msgs = [m for m in messages if m.get("role") != "system"]

    # Always keep the last N messages
    recent = conv_msgs[-keep_last_n:] if len(conv_msgs) > keep_last_n else
conv_msgs
    older = conv_msgs[:-keep_last_n] if len(conv_msgs) > keep_last_n else []

    # Count tokens in required messages
    required_tokens = sum(
```

```
        estimate_tokens(m.get("content", ""))
        for m in (system_msgs + recent)
    )

    if required_tokens >= max_tokens:
        # Even required messages exceed budget — trim recent messages
        result = system_msgs if keep_system else []
        remaining = max_tokens - sum(
            estimate_tokens(m.get("content", "")) for m in result
        )
        for msg in reversed(recent):
            msg_tokens = estimate_tokens(msg.get("content", ""))
            if remaining >= msg_tokens:
                result.append(msg)
                remaining -= msg_tokens
            else:
                break
        return result

    # Add older messages from most recent, dropping oldest first
    remaining_budget = max_tokens - required_tokens
    kept_older = []

    for msg in reversed(older):
        msg_tokens = estimate_tokens(msg.get("content", ""))
        if remaining_budget >= msg_tokens:
            kept_older.insert(0, msg)
            remaining_budget -= msg_tokens
        else:
            break

    # If we dropped messages and have a summary function, summarize them
    dropped = older[:len(older) - len(kept_older)]
    if dropped and summarize_fn:
        summary_text = summarize_fn(dropped)
        summary_msg = {
            "role": "user",
            "content": f"[Summary of earlier conversation: {summary_text}]",
        }
        return system_msgs + [summary_msg] + kept_older + recent

    return system_msgs + kept_older + recent
```

**4. Minimize tool definitions**

Tool definitions are sent with every request and can be surprisingly token-heavy. A tool with a detailed description and complex parameter schema can consume 200-500 tokens. If you have 10 tools, that's 2,000-5,000 tokens of tool definitions on every request — including requests where the model won't use most of them.

Strategies for reducing tool definition tokens:

- **Concise descriptions.** Replace "This tool searches the database for customer records matching the given criteria and returns the most recent matching records sorted by date" with "Search customer records by criteria."
- **Minimal parameter descriptions.** The parameter name and type are often sufficient. Don't describe what a "customer_id" parameter is.
- **Dynamic tool loading.** Only send tool definitions that are relevant to the current task. If you know the user is asking about billing, don't send the technical support tools.

```
def optimize_tool_definitions(
    tools: list[dict],
    relevant_only: list[str] | None = None,
) -> list[dict]:
    """Optimize tool definitions for token efficiency.

    Args:
        tools: Full list of tool definitions
        relevant_only: If provided, only include tools with these names
    """
    optimized = []
    for tool in tools:
        if relevant_only and tool["name"] not in relevant_only:
            continue

        # Create a compact copy
        compact = {
            "name": tool["name"],
            "description": tool["description"][:100],  # truncate long
descriptions
            "input_schema": tool["input_schema"],
        }

        # Remove parameter descriptions if the name is self-explanatory
        schema = compact["input_schema"]
        if "properties" in schema:
            for prop_name, prop_def in schema["properties"].items():
                if len(prop_name) > 3:  # name is descriptive enough
                    prop_def.pop("description", None)

        optimized.append(compact)

    return optimized
```

**5. Use structured output to reduce output tokens**

Output tokens cost 3-5x more than input tokens. Forcing structured output (JSON) instead of prose can dramatically reduce output length:

```
# Prose response: ~150 output tokens
# "Based on my analysis of the support ticket, I would classify this as a
#  billing issue because the customer mentions being charged twice for their
#  subscription. The key indicators are..."

# JSON response: ~15 output tokens
# {"category": "billing", "confidence": 0.95}
```

That's a 10x reduction in output tokens, which at 3-5x pricing means 30-50x less output cost per request. For high-volume classification and extraction tasks, this single change can reduce total cost by 50% or more.

**The five-technique priority matrix**

When optimizing a prompt, apply techniques in order of expected impact for your workload:

| Technique | Input savings | Output savings | Effort | Best for |
|---|---|---|---|---|
| Remove preamble/role-play | 10-30% | None | Minutes | All prompts |
| Structured output (JSON) | 5-15% | 50-90% | Low | Classification, extraction |
| Truncate conversation history | 30-80% | None | Medium | Multi-turn chat |
| Minimize tool definitions | 10-40% | None | Medium | Agent/tool-use features |
| Few-shot to zero-shot | 20-50% | None | Needs testing | Tasks where zero-shot quality suffices |

Start at the top. The first two techniques take minutes and typically deliver the largest combined impact. History truncation and tool minimization require more engineering but pay off at scale for the specific workloads they target.

A common mistake is applying these techniques in isolation without measuring their interaction. For example, removing preamble (technique 1) and switching to structured JSON output (technique 2) on the same prompt might reduce tokens by 60% — more than either technique alone — because the preamble often contains verbose format instructions that are made redundant by JSON schema constraints. Conversely, truncating history (technique 3) might reduce the effectiveness of caching (Chapter 6) if the cache boundary falls within the truncated region. Always measure the combined impact using the `PromptComparison` framework, and re-run your evaluation suite after each change to ensure quality is maintained. The techniques interact, and the optimal combination depends on your specific workload.

T> **Tip:** Anthropic and OpenAI both support structured output with JSON schema constraints. Use them. The model generates valid JSON without needing verbose format instructions in the prompt, saving tokens on both input (no format instructions) and output (no prose wrapper).

---

## Measuring compression impact

Prompt optimization should be measured, not assumed. Here's a framework for testing compression impact:

```
"""tokenwatch/prompt_test.py — A/B testing for prompt optimization."""

from dataclasses import dataclass
from tokenwatch.tokens import estimate_tokens
from tokenwatch.config import TokenwatchConfig


@dataclass
class PromptComparison:
    """Side-by-side comparison of original vs. optimized prompt."""
    original_tokens: int
    optimized_tokens: int
    token_reduction_pct: float
    estimated_monthly_savings_usd: float
    quality_delta: float | None  # requires manual evaluation


def compare_prompts(
    config: TokenwatchConfig,
    model: str,
    original_system: str,
    optimized_system: str,
    sample_messages: list[dict],
    daily_request_volume: int,
    avg_output_tokens: int = 200,
) -> PromptComparison:
    """Compare cost impact of original vs. optimized prompt."""
    pricing = config.get_model(model)

    orig_tokens = estimate_tokens(original_system)
    opt_tokens = estimate_tokens(optimized_system)

    # Add message tokens (same for both)
    msg_tokens = sum(
        estimate_tokens(m.get("content", ""))
        for m in sample_messages
    )

    orig_total_input = orig_tokens + msg_tokens
    opt_total_input = opt_tokens + msg_tokens
```

```
    orig_daily_cost = pricing.cost_for_tokens(
        input_tokens=orig_total_input * daily_request_volume,
        output_tokens=avg_output_tokens * daily_request_volume,
    )
    opt_daily_cost = pricing.cost_for_tokens(
        input_tokens=opt_total_input * daily_request_volume,
        output_tokens=avg_output_tokens * daily_request_volume,
    )

    savings_monthly = (orig_daily_cost - opt_daily_cost) * 30

    return PromptComparison(
        original_tokens=orig_total_input,
        optimized_tokens=opt_total_input,
        token_reduction_pct=(1 - opt_total_input / orig_total_input) * 100,
        estimated_monthly_savings_usd=savings_monthly,
        quality_delta=None,  # set after A/B evaluation
    )
```

---

## Common mistakes

**1. Compressing prompts without testing quality.** Aggressive compression can degrade output quality in subtle ways. Always run your evaluation suite after optimizing prompts. The goal is same quality, fewer tokens — not faster, worse outputs. The failure mode: an engineer removes "return only valid JSON" from the instructions, the model starts wrapping JSON in markdown code fences, and the downstream parser breaks on 5% of responses. The cost of debugging and retries exceeds the prompt compression savings.

**2. Over-engineering prompt compression.** A simple manual rewrite often outperforms automated compression. Read your prompts critically, remove redundancy, and test. Don't build an elaborate compression pipeline when a 10-minute rewrite would suffice. The `compress_system_prompt` function in tokenwatch is a starting point for identifying filler, not a replacement for human judgment.

**3. Ignoring tool definition tokens.** Tools can easily consume 2,000+ tokens on every request. If you're optimizing your system prompt from 400 to 200 tokens but sending 3,000 tokens of tool definitions, you're optimizing the wrong thing. Use `estimate_tokens(json.dumps(tools))` to measure your tool definition overhead before investing in system prompt compression.

**4. Using few-shot examples when zero-shot works.** Each few-shot example adds hundreds of tokens. Modern models (Claude Sonnet, GPT-4o) are strong zero-shot classifiers for common tasks. Test zero-shot first; add examples only when quality requires them. When examples are necessary, keep them minimal — a single well-chosen example is often as effective as five mediocre ones.

---

## Exercises

**Easy:** Take a prompt from a project you're working on (or use the wasteful classifier prompt from this chapter). Compress it using the techniques described and measure the token reduction. Does the output quality change?

**Medium:** Build a `PromptOptimizer` class that takes a system prompt, applies all the compression heuristics from `compress_system_prompt`, generates a "compression report" showing tokens before/after, and flags sections that might need manual review (e.g., few-shot examples that were not touched by automated compression).

**Hard:** Implement a conversation summarization function that takes a list of dropped messages (from `truncate_history`) and produces a concise summary using a cheap model (claude-haiku-4-5). Measure the cost of the summarization call itself against the savings from reduced history. At what conversation length does summarization become cost-effective? You should find that for short conversations (under 10 turns), the cost of the summarization call exceeds the savings. For long conversations (20+ turns), summarization pays for itself many times over because it replaces thousands of history tokens with a 100-200 token summary. The breakeven point depends on your model choice and the verbosity of your conversation — track it with the metering system from Chapter 3.

---

## Summary

In this chapter you learned:

- **Prompt waste is the most common source of unnecessary LLM cost** — verbose preambles, redundant descriptions, and unmanaged history add tokens without value
- **Five compression techniques** — remove preamble, use structured output, truncate history, minimize tool definitions, and constrain output format
- **Structured output is doubly effective** — it reduces both input tokens (no format instructions) and output tokens (JSON vs. prose), with output tokens being 3-5x more expensive
- **Conversation history management** is critical for multi-turn applications — smart truncation with optional summarization prevents quadratic cost growth
- **Always measure compression impact** — the `PromptComparison` framework ensures you're tracking both token savings and quality preservation

### Next chapter

Prompt compression reduces the tokens you send. Prompt caching reduces the *cost* of tokens you send repeatedly. Chapter 6 covers provider-native caching mechanics — how Anthropic prompt caching and OpenAI cached context work, how to design prompts for cache-ability, and how to measure and maximize your cache hit rate.

---

A> **Code for this chapter:** `project/cap05/tokenwatch/`

# Chapter 6 — Prompt Caching: Mechanics, Hit Rates, and Break-Even

Prompt compression reduces the number of tokens you send. Prompt caching reduces the *price* of the tokens you send. If your system prompt, tool definitions, or common context prefixes are the same across many requests, you're paying full price for the same tokens thousands of times a day. Caching lets you pay once to write those tokens into a cache and then read them back at a ~90% discount on every subsequent request.

The impact is dramatic. In a production system I worked on, enabling Anthropic prompt caching on a 2,000-token system prompt across 200,000 daily requests reduced input token costs by approximately 85% for those tokens. The engineering effort was minimal — restructure the prompt to put stable content first, add cache control markers, and monitor hit rates. This chapter teaches you exactly how to do that.

**Where this fits in your cost journey**

Prompt caching is the natural complement to prompt compression (Chapter 5). Compression reduces the number of tokens you send; caching reduces the price of the tokens that remain after compression. Applied together, the savings multiply: if compression removes 50% of tokens and caching discounts the remaining tokens by 90%, the combined input cost reduction is 95%. Caching also interacts with model routing (Chapter 8) — you can cache the system prompt once and reuse it across all requests, regardless of which model tier the router selects, as long as the cached prefix is the same. And caching has direct implications for agent cost control (Chapter 10), where the same system prompt and tool definitions are re-sent on every iteration of the agent loop. In an 8-iteration agent interaction, caching the stable prefix saves money on 7 of those 8 iterations.

---

## How provider-native caching works

Both Anthropic and OpenAI offer forms of prompt caching, but they work differently. Understanding the mechanics is essential for designing cache-friendly prompts.

**Anthropic prompt caching**

Anthropic's prompt caching is explicit and developer-controlled. You mark specific content blocks with `cache_control` to tell the API to cache everything from the beginning of the prompt up to that marker. Here's how it works:

1. **Cache write:** On the first request with a cache marker, the API processes all tokens normally and stores the cached prefix in a server-side cache. You pay a small cache-write premium (typically 25% more than normal input rate) for the cached tokens.
2. **Cache read:** On subsequent requests with the same prefix, the cached tokens are read from the cache instead of being reprocessed. You pay the cache-read rate — typically 90% less than the normal input rate.

3. **TTL (Time To Live):** Cached content has a TTL (currently 5 minutes for Anthropic, refreshed on each cache hit). If no request hits the cache within the TTL window, the cache entry expires and the next request pays the cache-write cost again.

4. **Minimum length:** There's a minimum prefix length for caching. For Claude Sonnet-class models, the cached prefix must be at least 1,024 tokens. Shorter prefixes can't be cached.

```
"""tokenwatch/cache.py — Prompt caching utilities and monitoring."""

import anthropic
from dataclasses import dataclass
from typing import Any

from tokenwatch.config import TokenwatchConfig


@dataclass
class CacheMetrics:
    """Metrics for cache performance tracking."""
    total_requests: int = 0
    cache_hits: int = 0
    cache_misses: int = 0
    cache_writes: int = 0
    tokens_cached_read: int = 0
    tokens_cached_write: int = 0
    tokens_uncached: int = 0
    estimated_savings_usd: float = 0.0

    @property
    def hit_rate(self) -> float:
        if self.total_requests == 0:
            return 0.0
        return self.cache_hits / self.total_requests

    @property
    def miss_rate(self) -> float:
        return 1.0 - self.hit_rate


class CacheOptimizer:
    """Helps design and monitor cache-friendly prompts."""

    def __init__(self, config: TokenwatchConfig):
        self.config = config
        self.metrics = CacheMetrics()

    def build_cached_messages(
        self,
        system_prompt: str,
        tool_definitions: list[dict] | None = None,
```

```
    static_context: str = "",
    user_message: str = "",
    conversation_history: list[dict] | None = None,
) -> dict[str, Any]:
    """Build a message payload optimized for Anthropic prompt caching.

    The key principle: put stable content FIRST, variable content LAST.
    Cache markers go on the boundary between stable and variable content.

    Optimal order:
    1. System prompt (stable) — cached
    2. Tool definitions (stable) — cached via system
    3. Static context (semi-stable) — cached if large enough
    4. Conversation history (variable) — not cached
    5. Current user message (variable) — not cached
    """
    # Build system message with cache control
    system_parts = []

    # Part 1: Core system prompt (always cached)
    system_parts.append({
        "type": "text",
        "text": system_prompt,
    })

    # Part 2: Static context (cached if provided)
    if static_context:
        system_parts.append({
            "type": "text",
            "text": static_context,
            "cache_control": {"type": "ephemeral"},
        })
    else:
        # Cache the system prompt itself
        system_parts[-1]["cache_control"] = {"type": "ephemeral"}

    # Build messages
    messages = []

    # Add conversation history (not cached — it changes)
    if conversation_history:
        messages.extend(conversation_history)

    # Add current user message
    if user_message:
        messages.append({"role": "user", "content": user_message})

    result = {
        "system": system_parts,
```

```
        "messages": messages,
    }

    if tool_definitions:
        result["tools"] = tool_definitions

    return result

def track_cache_result(
    self,
    response: anthropic.types.Message,
    model: str,
) -> dict[str, Any]:
    """Track cache performance from an API response.

    Call this after every API request to maintain cache metrics.
    """
    usage = response.usage
    cache_read = getattr(usage, "cache_read_input_tokens", 0) or 0
    cache_write = getattr(usage, "cache_creation_input_tokens", 0) or 0

    self.metrics.total_requests += 1

    if cache_read > 0:
        self.metrics.cache_hits += 1
        self.metrics.tokens_cached_read += cache_read
    elif cache_write > 0:
        self.metrics.cache_writes += 1
        self.metrics.tokens_cached_write += cache_write
    else:
        self.metrics.cache_misses += 1

    self.metrics.tokens_uncached += (
        usage.input_tokens - cache_read
    )

    # Calculate savings vs. not caching
    pricing = self.config.get_model(model)
    actual_cached_cost = (cache_read / 1_000_000) * pricing.cached_rate
    uncached_equivalent = (cache_read / 1_000_000) * pricing.input_rate
    savings = uncached_equivalent - actual_cached_cost
    self.metrics.estimated_savings_usd += savings

    return {
        "cache_hit": cache_read > 0,
        "cache_write": cache_write > 0,
        "tokens_cached_read": cache_read,
        "tokens_cached_write": cache_write,
        "savings_this_request": savings,
```

```
            "cumulative_savings": self.metrics.estimated_savings_usd,
            "hit_rate": self.metrics.hit_rate,
        }
```

**OpenAI automatic caching**

OpenAI's caching works differently — it's automatic and implicit. The API automatically caches prompt prefixes that are repeated across requests. You don't add cache markers; the system detects repeated prefixes and caches them. Cached tokens are billed at 50% of the normal input rate (less aggressive than Anthropic's 90% discount, but with zero developer effort).

The downside of automatic caching is less control. You can't force a cache hit, and you have less visibility into cache behavior. The upside is simplicity — just keep your prompt prefixes stable and caching happens automatically.

**Provider comparison table**

| Feature | Anthropic prompt caching | OpenAI automatic caching |
|---|---|---|
| Developer control | Explicit (`cache_control` markers) | Implicit (automatic detection) |
| Discount on cached tokens | ~90% off input rate | ~50% off input rate |
| Cache write premium | ~25% over input rate | None |
| Minimum prefix length | 1,024 tokens (Sonnet-class) | Not documented |
| TTL | 5 minutes (refreshed on hit) | Not documented |
| Visibility | Tokens reported in `usage` object | Tokens reported in `usage.prompt_tokens_details` |
| Best for | High-volume APIs with known stable prefixes | General use with minimal engineering effort |

The key takeaway: Anthropic's caching is more aggressive (deeper discount) but requires intentional prompt design. OpenAI's caching is simpler but delivers a smaller discount. For high-volume production endpoints, Anthropic's approach saves more money. For internal tools or lower-volume features, OpenAI's zero-effort caching is often sufficient.

**How caching breaks in production**

Caching failures in production are almost always silent. Your feature continues to work, responses look correct, but your cache hit rate drops to zero and your costs jump overnight. The most common causes are code changes that inadvertently reorder prompt content. A developer adds a dynamic timestamp to the system prompt ("Current time: 2026-06-12T14:30:00Z") for context, and suddenly the entire system prompt changes every second, destroying the cache. Another common failure: a feature flag rolls out a prompt variant to 50% of traffic, creating two distinct prefixes where there was one, halving the effective cache hit rate for each variant. A third pattern: an engineer moves the tool definitions to a different position in the message payload during a refactor, and the cached prefix no longer matches.

All three failures share a characteristic: the functional behavior of the feature does not change. Tests pass. Users get correct responses. The only symptom is a cost increase, which may not be noticed for days or weeks. This is why the `CacheMonitor` class exists and why you should wire its alerts to your team's notification channel. A cache hit rate alert that fires within minutes of a deployment can save thousands of dollars compared to discovering the regression in your next monthly bill review.

---

## Designing cache-friendly prompts

The single most important principle for prompt caching is: **stable content first, variable content last.** The cache works on prefixes — everything from the beginning of the prompt up to a certain point. If you put variable content (the user's question) before stable content (your system prompt), nothing gets cached.

Here's a concrete restructuring:

```
"""Example: Cache-unfriendly vs. cache-friendly prompt structure."""

# BAD: Variable content mixed with stable content
bad_structure = {
    "system": f"Help user {user_id} with their request about {topic}.",
    "messages": [
        {"role": "user", "content": question},
    ],
}
# Nothing is cacheable because user_id and topic vary per request

# GOOD: Stable prefix, variable suffix
good_structure = {
    "system": [
        {
            "type": "text",
            "text": "You are a customer support assistant. Follow these
guidelines: ...",
            "cache_control": {"type": "ephemeral"},
        }
    ],
    "messages": [
        {
            "role": "user",
            "content": f"User: {user_id}. Topic: {topic}. Question: {question}",
        },
    ],
}
# The system prompt (stable) is cached; user details are in the variable message
```

### Cache-friendly patterns

**Pattern 1: Large system prompt with cache marker.** The most common pattern. Your system prompt is 1,000+ tokens of instructions, guidelines, and few-shot examples. Mark it for caching and every request reads it from cache.

**Pattern 2: System prompt + RAG context.** If you have a large, relatively stable knowledge base context (e.g., product documentation that changes weekly), include it in the system message with a cache marker. The RAG context changes less frequently than user messages, so it benefits from caching.

**Pattern 3: Multi-turn conversations with shared prefix.** In a conversation, the system prompt and early messages are the same for all subsequent turns. Place a cache marker after the system prompt and optionally after the first few messages. Later turns benefit from reading the cached prefix.

```
def build_multiturn_cached(
    system: str,
    history: list[dict],
    new_message: str,
    cache_after_turn: int = 2,
) -> dict:
    """Build a multi-turn message with strategic cache markers.

    Caches the system prompt and optionally the first N turns
    of conversation history.
    """
    system_block = [
        {
            "type": "text",
            "text": system,
            "cache_control": {"type": "ephemeral"},
        }
    ]

    messages = []
    for i, msg in enumerate(history):
        if i == cache_after_turn * 2 - 1 and len(history) > cache_after_turn * 2:
            # Add cache marker after the Nth assistant response
            if isinstance(msg.get("content"), str):
                msg = {
                    **msg,
                    "content": [
                        {
                            "type": "text",
                            "text": msg["content"],
                            "cache_control": {"type": "ephemeral"},
                        }
                    ],
                }
```

```
        messages.append(msg)

    messages.append({"role": "user", "content": new_message})

    return {"system": system_block, "messages": messages}
```

## Cache break-even analysis

Caching isn't free. There's a cache-write cost (typically 25% premium on the first request) that you need to amortize across subsequent cache reads. The break-even calculation tells you whether caching makes economic sense for your workload.

```
"""tokenwatch/cache_economics.py — Cache break-even analysis."""

from dataclasses import dataclass
from tokenwatch.config import TokenwatchConfig


@dataclass
class CacheBreakEven:
    """Break-even analysis for prompt caching."""
    cacheable_tokens: int
    cache_write_cost: float
    per_request_savings: float
    break_even_requests: int
    ttl_seconds: int
    required_rps: float  # requests per second to sustain cache
    daily_savings_at_volume: float
    monthly_savings_at_volume: float
    recommendation: str


def analyze_cache_breakeven(
    config: TokenwatchConfig,
    model: str,
    cacheable_tokens: int,
    daily_request_volume: int,
    cache_ttl_seconds: int = 300,
    cache_write_premium_pct: float = 25.0,
) -> CacheBreakEven:
    """Calculate whether prompt caching makes economic sense.

    Args:
        cacheable_tokens: Number of tokens in the cacheable prefix
        daily_request_volume: Expected daily request volume
        cache_ttl_seconds: Cache TTL in seconds (default 300 = 5 min)
        cache_write_premium_pct: Premium for cache write (default 25%)
    """
    pricing = config.get_model(model)
```

```
    # Cost of a cache write (one-time, per TTL window)
    normal_cost = (cacheable_tokens / 1_000_000) * pricing.input_rate
    write_premium = normal_cost * (cache_write_premium_pct / 100)
    cache_write_cost = normal_cost + write_premium

    # Cost savings per cache hit
    uncached_cost = (cacheable_tokens / 1_000_000) * pricing.input_rate
    cached_cost = (cacheable_tokens / 1_000_000) * pricing.cached_rate
    per_request_savings = uncached_cost - cached_cost

    # Break-even: how many cache hits to recoup the write cost
    if per_request_savings > 0:
        break_even_requests = int(cache_write_cost / per_request_savings) + 1
    else:
        break_even_requests = float("inf")

    # Required RPS to keep cache alive (at least 1 request per TTL)
    required_rps = 1.0 / cache_ttl_seconds

    # Daily economics
    ttl_windows_per_day = 86400 / cache_ttl_seconds
    daily_write_cost = cache_write_cost * ttl_windows_per_day
    daily_read_savings = per_request_savings * daily_request_volume
    daily_net_savings = daily_read_savings - daily_write_cost

    # Recommendation
    actual_rps = daily_request_volume / 86400
    if cacheable_tokens < 1024:
        recommendation = (
            "NOT RECOMMENDED: Prefix too short for caching "
            "(minimum 1,024 tokens for Sonnet-class models)"
        )
    elif actual_rps < required_rps:
        recommendation = (
            f"MARGINAL: Traffic ({actual_rps:.3f} rps) is below cache-sustaining
"
            f"rate ({required_rps:.3f} rps) — cache will expire between requests"
        )
    elif daily_net_savings < 0:
        recommendation = "NOT RECOMMENDED: Cache write costs exceed savings"
    elif daily_net_savings < 1.0:
        recommendation = "MARGINAL: Savings are small — may not justify
complexity"
    else:
        recommendation = (
            f"RECOMMENDED: Net savings of ${daily_net_savings:.2f}/day "
            f"(${daily_net_savings * 30:.2f}/month)"
        )
```

```
    return CacheBreakEven(
        cacheable_tokens=cacheable_tokens,
        cache_write_cost=cache_write_cost,
        per_request_savings=per_request_savings,
        break_even_requests=break_even_requests,
        ttl_seconds=cache_ttl_seconds,
        required_rps=required_rps,
        daily_savings_at_volume=daily_net_savings,
        monthly_savings_at_volume=daily_net_savings * 30,
        recommendation=recommendation,
    )
```

**Worked example: break-even with concrete numbers**

Let's walk through the break-even math with `pricing.yaml` rates for Claude Sonnet (input: $3.00/M, cached: $0.30/M). Suppose your system prompt and tool definitions total 3,000 cacheable tokens:

```
Cache write cost (one-time per 5-minute window):
  Normal cost:  3,000 / 1,000,000 * $3.00 = $0.0090
  Write premium (25%):                      $0.0023
  Total write cost:                         $0.0113

Savings per cache hit:
  Uncached cost: 3,000 / 1,000,000 * $3.00 = $0.0090
  Cached cost:   3,000 / 1,000,000 * $0.30 = $0.0009
  Savings per hit:                           $0.0081

Break-even: $0.0113 / $0.0081 = 1.4 → 2 cache hits
```

You break even after just 2 cache hits within a 5-minute window. At 100,000 requests per day (roughly 1.16 RPS), you get approximately 347 hits per 5-minute window. The net daily savings are substantial:

```
TTL windows per day: 86,400 / 300 = 288
Daily write cost: 288 * $0.0113 = $3.25
Daily read savings: 100,000 * $0.0081 = $810.00
Daily net savings: $806.75
Monthly net savings: $24,202
```

The `analyze_cache_breakeven` function in tokenwatch automates this entire calculation:

```
"""example_cache_breakeven.py — Should you cache this prompt?"""

from tokenwatch.config import TokenwatchConfig
from tokenwatch.cache_economics import analyze_cache_breakeven

config = TokenwatchConfig.from_yaml("pricing.yaml")
```

```
# Scenario 1: High-volume API with large system prompt
result = analyze_cache_breakeven(
    config=config,
    model="claude-sonnet-4-20250514",
    cacheable_tokens=3_000,    # system prompt + tool definitions
    daily_request_volume=100_000,
)
print(f"Scenario 1 — High volume:")
print(f"  Break-even: {result.break_even_requests} requests")
print(f"  Monthly savings: ${result.monthly_savings_at_volume:.2f}")
print(f"  Recommendation: {result.recommendation}")

# Scenario 2: Low-volume internal tool
result2 = analyze_cache_breakeven(
    config=config,
    model="claude-sonnet-4-20250514",
    cacheable_tokens=1_500,
    daily_request_volume=50,
)
print(f"\nScenario 2 — Low volume:")
print(f"  Break-even: {result2.break_even_requests} requests")
print(f"  Monthly savings: ${result2.monthly_savings_at_volume:.2f}")
print(f"  Recommendation: {result2.recommendation}")
```

I> **Note:** Cache break-even depends heavily on traffic patterns, not just volume. A service handling 100,000 requests per day with even distribution (1.15 rps) easily sustains cache. The same volume concentrated into a 2-hour burst (13.9 rps during burst, 0 rps otherwise) will see cache expiration during off-hours, reducing effective hit rates.

---

## Monitoring cache health

Cache hit rate is the key metric, but it's not the only one. You also need to track cache churn (how often the cache is rewritten), savings over time, and anomalies like sudden drops in hit rate.

```
"""tokenwatch/cache_monitor.py — Cache health monitoring."""

from dataclasses import dataclass, field
from collections import deque
from datetime import datetime, timezone


@dataclass
class CacheHealthReport:
    """Health report for cache performance."""
    current_hit_rate: float
    rolling_hit_rate_1h: float
    total_savings_usd: float
    cache_churn_rate: float  # writes / total requests
```

```
    estimated_monthly_savings: float
    alerts: list[str] = field(default_factory=list)


class CacheMonitor:
    """Monitors cache performance and generates alerts."""

    def __init__(self, alert_hit_rate_threshold: float = 0.7):
        self.alert_threshold = alert_hit_rate_threshold
        self.events: deque = deque(maxlen=10_000)
        self.total_savings = 0.0

    def record_event(
        self,
        cache_hit: bool,
        cache_write: bool,
        savings_usd: float,
    ) -> None:
        """Record a cache event."""
        self.events.append({
            "timestamp": datetime.now(timezone.utc).isoformat(),
            "cache_hit": cache_hit,
            "cache_write": cache_write,
            "savings": savings_usd,
        })
        self.total_savings += savings_usd

    def health_report(self) -> CacheHealthReport:
        """Generate a cache health report."""
        if not self.events:
            return CacheHealthReport(
                current_hit_rate=0.0,
                rolling_hit_rate_1h=0.0,
                total_savings_usd=0.0,
                cache_churn_rate=0.0,
                estimated_monthly_savings=0.0,
            )

        events = list(self.events)
        total = len(events)
        hits = sum(1 for e in events if e["cache_hit"])
        writes = sum(1 for e in events if e["cache_write"])

        hit_rate = hits / total
        churn = writes / total

        # Rolling 1-hour hit rate (approximate with last 1000 events)
        recent = events[-1000:]
        recent_hits = sum(1 for e in recent if e["cache_hit"])
```

```
    rolling_rate = recent_hits / len(recent)

    # Estimate monthly savings from daily rate
    daily_savings = self.total_savings  # simplified
    monthly = daily_savings * 30

    alerts = []
    if hit_rate < self.alert_threshold:
        alerts.append(
            f"Cache hit rate ({hit_rate:.1%}) below threshold "
            f"({self.alert_threshold:.1%})"
        )
    if churn > 0.3:
        alerts.append(
            f"High cache churn ({churn:.1%}) — cache is being "
            f"rewritten too frequently"
        )

    return CacheHealthReport(
        current_hit_rate=hit_rate,
        rolling_hit_rate_1h=rolling_rate,
        total_savings_usd=self.total_savings,
        cache_churn_rate=churn,
        estimated_monthly_savings=monthly,
        alerts=alerts,
    )
```

---

## Common mistakes

**1. Putting variable content before stable content.** If your system prompt starts with `"Help user {user_id}..."`, the entire prefix changes per request and nothing is cached. Move user-specific content to the user message; keep the system prompt stable and cacheable.

**2. Ignoring minimum cache length.** Anthropic requires at least 1,024 tokens for the cacheable prefix (for Sonnet-class models). If your system prompt is 500 tokens, it won't be cached. Combine system prompt and tool definitions to reach the minimum, or accept that caching won't help for this endpoint.

**3. Not monitoring cache hit rates.** Enabling caching without monitoring is flying blind. A code change that reorders prompt content can silently break caching, turning your 90% hit rate into 0% overnight. The `CacheMonitor` class in tokenwatch tracks hit rates and fires alerts when they drop below threshold. Wire it to your alerting system — a cache regression at scale can add thousands of dollars per day.

**4. Caching low-volume endpoints.** If an endpoint handles 10 requests per day, the cache will expire (5-minute TTL) between most requests. You'll pay the cache-write premium on most requests without getting cache-read discounts. Run the break-even analysis before enabling

caching. The `required_rps` field in `CacheBreakEven` tells you the minimum request rate needed to sustain cache — if your actual rate is below this threshold, caching costs more than it saves.

**5. Assuming cache hit rates are stable.** Cache hit rates change with traffic patterns. During peak hours, your rate might be 95%. During off-hours, it might drop to 60% as gaps between requests approach the TTL. Weekend traffic might be so low that the cache expires entirely. The `CacheMonitor` tracks rolling hit rates, not just overall averages, so you can see these patterns. If your feature has strongly diurnal traffic, calculate the daily savings using the actual hit rate distribution, not the peak-hour rate.

---

## Exercises

**Easy:** Take the system prompt from your metered client (Chapter 3) and restructure it for caching using `CacheOptimizer.build_cached_messages`. Make a few requests and verify cache hits appear in the API response. You should see `cache_creation_input_tokens` on the first request (the cache write) and `cache_read_input_tokens` on subsequent requests (cache hits). If you do not see cache reads after the first request, check two things: that your cacheable prefix exceeds 1,024 tokens and that you are making subsequent requests within the 5-minute TTL window.

**Medium:** Run the `analyze_cache_breakeven` function on three different workload profiles: high-volume API (100K req/day), moderate internal tool (5K req/day), and low-volume batch job (100 req/day). For each, determine whether caching is recommended and at what volume it becomes worthwhile.

**Hard:** Build a `CacheSimulator` that takes a stream of requests with timestamps and simulates cache behavior. Model cache TTL expiration, write costs, and read savings. Produce a chart (using matplotlib) showing cumulative savings vs. time, hit rate over time, and the break-even point. Test with different traffic patterns: steady, bursty, and diurnal (high during business hours, low at night).

---

## Summary

In this chapter you learned:

- **Prompt caching reduces the cost of repeated prefixes** — Anthropic offers ~90% discount on cached tokens, OpenAI offers ~50% with automatic caching
- **Stable content first, variable content last** — this principle is the key to cache-friendly prompt design
- **Cache break-even analysis** determines whether caching makes economic sense given your traffic volume and pattern
- **Cache TTL matters** — low-traffic endpoints may not sustain cache between requests, negating the benefits
- **Monitor cache health** — hit rate, churn rate, and savings tracking catch silent regressions that can eliminate your cache benefits overnight

**Next chapter**
Caching saves money on synchronous requests. But what about workloads that don't need real-time responses? Chapter 7 covers batch processing — how to queue non-urgent LLM work for asynchronous processing at 50% discounted rates, including Anthropic's Message Batches API and patterns for splitting workloads between sync and async paths.

---

A> **Code for this chapter:** `project/cap06/tokenwatch/`

# Chapter 7 — Batch Processing and Async Workloads

Not every LLM request needs an answer in 200 milliseconds. Document classification, content moderation, report generation, data enrichment — these workloads can tolerate minutes or even hours of latency. And that tolerance is worth money. Both Anthropic and OpenAI offer batch processing APIs that discount token prices by 50% in exchange for asynchronous execution with longer turnaround times.

If 30-40% of your LLM workload is latency-tolerant — and in most production systems it is — batch processing can reduce your overall API spend by 15-20% with no quality degradation. The challenge is identifying which requests can be batched, building the queue infrastructure, and handling the asynchronous result flow. This chapter builds tokenwatch's batch processing module to solve all three.

**Where this fits in your cost journey**
Batch processing sits alongside prompt caching (Chapter 6) as a technique that reduces the price per token without changing the tokens themselves. The key difference is the mechanism: caching reduces price through reuse (you already computed this prefix, so you pay less to compute it again), while batching reduces price through scheduling flexibility (you are willing to wait, so the provider gives you a discount for filling idle capacity). The two techniques are complementary and can be combined. A batch request can also benefit from prompt caching if the system prompt is the same across batch items. In practice, batch processing is most impactful for workloads that were never latency-sensitive to begin with but were processed synchronously out of engineering convenience — overnight document classification, backfill enrichment, periodic quality audits. Identifying these workloads in your metering data (Chapter 3) and reclassifying them as batch-eligible is often the lowest-effort cost reduction available after prompt compression.

---

## Synchronous vs. asynchronous: the cost divide

Let's classify LLM workloads by latency sensitivity:

**Must be synchronous (real-time):** - Chat agents responding to users - Auto-complete / inline suggestions - Real-time content moderation (before publishing) - Interactive tool calling in agent loops

**Can be asynchronous (batch-eligible):** - Overnight document classification - Batch content enrichment (tagging, summarization) - Report generation - Data extraction from large document

sets - Quality evaluation of past responses - Embedding generation for knowledge bases - Automated test generation

The second category is often larger than teams realize. If you audit your LLM API calls and tag each by latency requirement, you'll typically find 30-50% of total token volume is batch-eligible. At a 50% batch discount, that's 15-25% savings on your total API bill.

**Worked example: batch savings with pricing.yaml rates**

Consider a document classification workload running on Claude Sonnet. Using the rates from `pricing.yaml`:

| Mode | Input rate ($/M) | $Output rate ($/M) |
|---|---|---|
| Synchronous | $3.00 | $15.00 |
| Batch | $1.50 | $7.50 |

For 50,000 daily classification requests (avg 800 input tokens, 50 output tokens):

```
Synchronous:
  Input:  50,000 * 800 / 1,000,000 * $3.00  = $120.00/day
  Output: 50,000 * 50  / 1,000,000 * $15.00 = $37.50/day
  Total:  $157.50/day → $4,725/month

Batch:
  Input:  50,000 * 800 / 1,000,000 * $1.50  = $60.00/day
  Output: 50,000 * 50  / 1,000,000 * $7.50  = $18.75/day
  Total:  $78.75/day → $2,363/month

Monthly savings: $2,362 (50%)
```

The savings are exactly proportional to the discount — 50% across the board. No code changes to the prompts, no quality degradation, no caching setup. The only cost is engineering the batch queue and accepting higher latency.

What makes batch processing particularly attractive is that it stacks with other optimizations. If you have already compressed prompts (Chapter 5) to reduce token count by 40%, and you then move the workload to batch, you save 50% on the already-reduced token count. The combined savings versus the original unoptimized synchronous approach is 70%. Add caching on the system prompt (Chapter 6) and the total savings can exceed 80%. This is the compounding effect that makes the optimization stack so powerful — each technique multiplies on top of the others rather than competing with them.

**Real-world scenario: overnight enrichment pipeline**

At a large fintech, a data enrichment pipeline tagged customer support interactions with sentiment, topic, and priority using Claude Sonnet. Initially implemented as synchronous calls triggered by a webhook, it processed 80,000 records per day. Nobody needed the enrichment in real-time — analysts reviewed the data the next morning. Moving this pipeline to batch processing

cut the monthly LLM bill for this feature from $9,400 to $4,700. The engineering effort was two days: modify the webhook handler to enqueue records into the `BatchQueue`, add a nightly cron job to flush the queue, and build a result handler that wrote classifications back to the database.

---

## The Anthropic Message Batches API

Anthropic's batch API lets you submit up to 10,000 requests in a single batch. Each request is a standard message payload. The API processes them asynchronously and returns results when the batch completes (typically within 24 hours, often much faster).

```
"""tokenwatch/batch.py — Batch processing for LLM workloads."""

import json
import time
import uuid
from dataclasses import dataclass, field
from pathlib import Path
from typing import Any

import anthropic

from tokenwatch.config import TokenwatchConfig
from tokenwatch.middleware import TokenMeter
from tokenwatch.meter import UsageRecord


@dataclass
class BatchRequest:
    """A single request within a batch."""
    custom_id: str
    model: str
    max_tokens: int
    messages: list[dict]
    system: str = ""
    metadata: dict = field(default_factory=dict)  # feature, user_id, team


@dataclass
class BatchJob:
    """A submitted batch job."""
    batch_id: str
    request_count: int
    status: str
    created_at: str
    custom_ids: list[str]


@dataclass
class BatchResult:
```

```
    """Result from a completed batch."""
    custom_id: str
    response: dict | None
    error: dict | None
    usage: dict | None


class BatchProcessor:
    """Manages batch submission, polling, and result processing."""

    def __init__(
        self,
        config: TokenwatchConfig,
        meter: TokenMeter | None = None,
        client: anthropic.Anthropic | None = None,
    ):
        self.config = config
        self.meter = meter
        self.client = client or anthropic.Anthropic()

    def create_batch(
        self,
        requests: list[BatchRequest],
    ) -> BatchJob:
        """Submit a batch of requests to the Anthropic API.

        Each request becomes a standard messages API call
        processed asynchronously at batch pricing.
        """
        # Build the batch request payload
        batch_requests = []
        for req in requests:
            params = {
                "model": req.model,
                "max_tokens": req.max_tokens,
                "messages": req.messages,
            }
            if req.system:
                params["system"] = req.system

            batch_requests.append({
                "custom_id": req.custom_id,
                "params": params,
            })

        # Submit the batch
        result = self.client.messages.batches.create(
            requests=batch_requests,
        )
```

```
        return BatchJob(
            batch_id=result.id,
            request_count=len(requests),
            status=result.processing_status,
            created_at=result.created_at.isoformat() if hasattr(result,
'created_at') else "",
            custom_ids=[r.custom_id for r in requests],
        )

    def check_status(self, batch_id: str) -> str:
        """Check the status of a batch job."""
        result = self.client.messages.batches.retrieve(batch_id)
        return result.processing_status

    def get_results(self, batch_id: str) -> list[BatchResult]:
        """Retrieve results from a completed batch.

        Polls until the batch is complete, then retrieves all results.
        """
        results = []
        for result in self.client.messages.batches.results(batch_id):
            custom_id = result.custom_id

            if result.result.type == "succeeded":
                message = result.result.message
                usage = {
                    "input_tokens": message.usage.input_tokens,
                    "output_tokens": message.usage.output_tokens,
                }
                response = {
                    "content": [
                        {"type": b.type, "text": getattr(b, "text", "")}
                        for b in message.content
                    ],
                    "model": message.model,
                    "stop_reason": message.stop_reason,
                }
                results.append(BatchResult(
                    custom_id=custom_id,
                    response=response,
                    error=None,
                    usage=usage,
                ))
            else:
                results.append(BatchResult(
                    custom_id=custom_id,
                    response=None,
                    error={"type": result.result.type},
```

```
                    usage=None,
                ))

        return results

    def wait_for_completion(
        self,
        batch_id: str,
        poll_interval_seconds: int = 30,
        timeout_seconds: int = 86400,
    ) -> list[BatchResult]:
        """Wait for a batch to complete and return results.

        Args:
            poll_interval_seconds: How often to check status
            timeout_seconds: Maximum wait time before raising
        """
        start = time.monotonic()

        while True:
            status = self.check_status(batch_id)

            if status == "ended":
                return self.get_results(batch_id)

            elapsed = time.monotonic() - start
            if elapsed > timeout_seconds:
                raise TimeoutError(
                    f"Batch {batch_id} did not complete within "
                    f"{timeout_seconds}s (status: {status})"
                )

            time.sleep(poll_interval_seconds)
```

## Building the batch queue

In a real application, you don't want to manually construct batch requests. You want a queue that collects eligible requests and automatically submits them as batches:

```
"""tokenwatch/batch_queue.py — Automatic batch collection and submission."""

import threading
import time
from dataclasses import dataclass, field
from typing import Any, Callable
from collections import deque

from tokenwatch.batch import BatchProcessor, BatchRequest, BatchJob
```

```
@dataclass
class QueuedRequest:
    """A request waiting in the batch queue."""
    request: BatchRequest
    callback: Callable[[dict], None] | None = None
    submitted_at: float = 0.0


class BatchQueue:
    """Collects requests and submits them as batches.

    Supports two triggering strategies:
    1. Size-based: Submit when queue reaches max_batch_size
    2. Time-based: Submit when oldest request has waited max_wait_seconds

    The queue runs a background thread that checks for submission triggers.
    """

    def __init__(
        self,
        processor: BatchProcessor,
        max_batch_size: int = 1000,
        max_wait_seconds: float = 300.0,
        default_model: str = "claude-sonnet-4-20250514",
    ):
        self.processor = processor
        self.max_batch_size = max_batch_size
        self.max_wait_seconds = max_wait_seconds
        self.default_model = default_model

        self._queue: deque[QueuedRequest] = deque()
        self._lock = threading.Lock()
        self._pending_batches: dict[str, list[QueuedRequest]] = {}
        self._running = False

    def enqueue(
        self,
        messages: list[dict],
        system: str = "",
        max_tokens: int = 1024,
        model: str | None = None,
        custom_id: str | None = None,
        metadata: dict | None = None,
        callback: Callable[[dict], None] | None = None,
    ) -> str:
        """Add a request to the batch queue.

        Returns the custom_id for tracking. The callback (if provided)
```

```
        will be called with the result when the batch completes.
        """
        import uuid
        cid = custom_id or str(uuid.uuid4())

        request = BatchRequest(
            custom_id=cid,
            model=model or self.default_model,
            max_tokens=max_tokens,
            messages=messages,
            system=system,
            metadata=metadata or {},
        )

        queued = QueuedRequest(
            request=request,
            callback=callback,
            submitted_at=time.monotonic(),
        )

        with self._lock:
            self._queue.append(queued)

            # Check if we should submit immediately (size trigger)
            if len(self._queue) >= self.max_batch_size:
                self._submit_batch()

        return cid

    def _submit_batch(self) -> BatchJob | None:
        """Submit the current queue as a batch. Must hold _lock."""
        if not self._queue:
            return None

        # Take up to max_batch_size requests
        batch_items = []
        while self._queue and len(batch_items) < self.max_batch_size:
            batch_items.append(self._queue.popleft())

        requests = [item.request for item in batch_items]
        job = self.processor.create_batch(requests)

        self._pending_batches[job.batch_id] = batch_items

        return job

    def flush(self) -> BatchJob | None:
        """Force-submit any queued requests as a batch."""
        with self._lock:
```

```
            return self._submit_batch()

    @property
    def queue_size(self) -> int:
        return len(self._queue)

    @property
    def pending_batches(self) -> int:
        return len(self._pending_batches)
```

## Deciding what to batch

Not every request should be batched. The key question is: "Can the user or system wait for this result?" The answer often depends on who consumes the result, not on the technical nature of the request. A sentiment analysis that feeds a real-time dashboard must be synchronous. The exact same sentiment analysis that feeds a weekly report can be batched. The model, prompt, and output are identical — only the latency tolerance differs.

In production, the most reliable way to identify batch candidates is to audit your existing synchronous workloads and ask, for each one: "What would happen if this result arrived 30 minutes later?" If the honest answer is "nothing — nobody looks at it for hours anyway," that workload is batch-eligible. Common examples include nightly data enrichment pipelines that were originally implemented as synchronous API calls inside a loop, quality evaluation of past responses (which by definition are not time-sensitive), and report generation that runs on a schedule.

Here's a pattern for routing requests between sync and batch paths:

```
"""tokenwatch/batch_router.py — Route requests between sync and batch."""

from dataclasses import dataclass
from enum import Enum


class ProcessingMode(Enum):
    SYNC = "sync"
    BATCH = "batch"


@dataclass
class RoutingRule:
    """Rule for deciding sync vs. batch processing."""
    feature: str
    mode: ProcessingMode
    reason: str


# Define routing rules per feature
DEFAULT_ROUTING: dict[str, ProcessingMode] = {
    # Real-time features — must be synchronous
```

```
    "chat-agent": ProcessingMode.SYNC,
    "autocomplete": ProcessingMode.SYNC,
    "content-moderation-realtime": ProcessingMode.SYNC,

    # Batch-eligible features — can tolerate latency
    "document-classification": ProcessingMode.BATCH,
    "report-generation": ProcessingMode.BATCH,
    "data-enrichment": ProcessingMode.BATCH,
    "quality-evaluation": ProcessingMode.BATCH,
    "content-moderation-backfill": ProcessingMode.BATCH,
    "embedding-generation": ProcessingMode.BATCH,
}


def route_request(
    feature: str,
    routing_rules: dict[str, ProcessingMode] | None = None,
    force_sync: bool = False,
) -> ProcessingMode:
    """Determine whether a request should be sync or batch.

    Args:
        feature: The feature name
        routing_rules: Custom routing rules (defaults to DEFAULT_ROUTING)
        force_sync: Override to force synchronous processing
    """
    if force_sync:
        return ProcessingMode.SYNC

    rules = routing_rules or DEFAULT_ROUTING
    return rules.get(feature, ProcessingMode.SYNC)  # default to sync for safety
```

T> **Tip:** When in doubt, default to synchronous. It's always safe to process synchronously; the cost is just higher. Once you've verified that a feature can tolerate batch latency, move it to batch processing. Going the other direction (batch to sync because users complained about delays) is harder.

**SLA design for mixed batch and realtime products**

When a product combines synchronous and asynchronous LLM features, you need explicit SLAs for each path — and those SLAs need to be communicated clearly to product managers and end users. The synchronous path might promise a 2-second response time. The batch path might promise results within 4 hours. Both are valid, but conflating them in a single "response time" metric creates confusion. Product managers who see "average response time: 45 minutes" will panic, because that number mixes sub-second chat responses with multi-hour batch jobs.

Design your SLAs in tiers. Define a "realtime SLA" (p95 latency under 3 seconds, availability 99.9%) and a "batch SLA" (completion within N hours, availability 99.5%). Publish both, and tag every feature with the SLA tier it uses. This clarity prevents the inevitable conversation where a PM

asks why the "document analysis" feature takes two hours and whether that is a bug. It is not a bug — it is a batch-tier feature operating within its SLA. The cost savings from batch processing only materialize if the organization accepts the latency trade-off, and that acceptance requires upfront communication, not after-the-fact explanation.

Communicating async latency to users requires different UX patterns than synchronous features. A synchronous feature shows a spinner for two seconds and then returns a result. A batch feature needs to set expectations at submission time ("Your analysis will be ready within 2 hours — we'll notify you by email"), provide a status-check mechanism (a "processing" state in the UI), and deliver results asynchronously (push notification, email, or a dashboard update). Teams that skip this UX investment find that users submit the same request repeatedly because they think the first one failed, which multiplies both cost and confusion. The engineering cost of building proper async UX is a one-time investment; the cost savings from batch processing are ongoing.

**Retry and partial-failure economics**

Batch processing introduces an economic dimension to failure handling that synchronous systems do not have. In a synchronous system, a failed request is retried immediately at the same per-token cost. In a batch system, the retry economics are more nuanced. If 5 out of 1,000 requests in a batch fail, you have three options: retry the 5 failures as a new mini-batch (still at batch rates, but with the overhead of a new batch submission), retry them synchronously (at 2x the batch rate, but with immediate results), or drop them (zero cost, but incomplete data). The right choice depends on whether the downstream consumer can tolerate incomplete results. A nightly enrichment pipeline that feeds a weekly report can usually tolerate a 0.5% data gap. A compliance audit that must process every document cannot.

The cost of retrying failed batch items as synchronous requests is often acceptable in practice. If your batch failure rate is 0.5% and synchronous processing costs 2x batch rates, the blended cost increase from synchronous retries is only 0.5% of 2x, which is 1% above the pure batch rate. This is negligible compared to the 50% savings from batch processing in the first place. The `process_batch_results` function in `batch_retry.py` separates successes from failures and builds the retry list — use it to implement a hybrid retry strategy where failed items fall back to synchronous processing rather than waiting for another batch cycle.

---

## Batch cost analysis

Let's build the tooling to quantify how much you'd save by moving eligible workloads to batch:

```
"""tokenwatch/batch_analysis.py — Batch savings analysis."""

from dataclasses import dataclass
from tokenwatch.config import TokenwatchConfig
from tokenwatch.store import UsageStore


@dataclass
class BatchSavingsReport:
```

```
    """Report on potential savings from batch processing."""
    total_current_cost: float
    sync_cost: float
    batch_eligible_cost: float
    batch_discounted_cost: float
    potential_savings: float
    savings_pct: float
    batch_eligible_pct: float  # % of requests that are batch-eligible
    features_to_batch: list[dict]


def analyze_batch_savings(
    config: TokenwatchConfig,
    store: UsageStore,
    batch_eligible_features: list[str],
    since: str | None = None,
) -> BatchSavingsReport:
    """Analyze potential savings from batch processing.

    Args:
        batch_eligible_features: Features that can be moved to batch
    """
    all_records = store.query(since=since)
    if not all_records:
        return BatchSavingsReport(
            total_current_cost=0, sync_cost=0,
            batch_eligible_cost=0, batch_discounted_cost=0,
            potential_savings=0, savings_pct=0,
            batch_eligible_pct=0, features_to_batch=[],
        )

    sync_records = [
        r for r in all_records
        if r.feature not in batch_eligible_features
    ]
    batch_records = [
        r for r in all_records
        if r.feature in batch_eligible_features
    ]

    total_cost = sum(r.cost_usd for r in all_records)
    sync_cost = sum(r.cost_usd for r in sync_records)
    batch_eligible_cost = sum(r.cost_usd for r in batch_records)

    # Calculate batch-discounted cost
    batch_discounted_cost = 0.0
    for r in batch_records:
        try:
            pricing = config.get_model(r.model)
```

```
            cost = pricing.cost_for_tokens(
                input_tokens=r.input_tokens,
                output_tokens=r.output_tokens,
                cached_tokens=r.cache_read_tokens,
                batch=True,
            )
            batch_discounted_cost += cost
        except KeyError:
            batch_discounted_cost += r.cost_usd  # no batch pricing available

    savings = batch_eligible_cost - batch_discounted_cost

    # Per-feature breakdown
    feature_savings = []
    for feature in batch_eligible_features:
        feat_records = [r for r in batch_records if r.feature == feature]
        if feat_records:
            current = sum(r.cost_usd for r in feat_records)
            discounted = 0.0
            for r in feat_records:
                try:
                    pricing = config.get_model(r.model)
                    discounted += pricing.cost_for_tokens(
                        input_tokens=r.input_tokens,
                        output_tokens=r.output_tokens,
                        batch=True,
                    )
                except KeyError:
                    discounted += r.cost_usd

            feature_savings.append({
                "feature": feature,
                "current_cost": current,
                "batch_cost": discounted,
                "savings": current - discounted,
                "request_count": len(feat_records),
            })

    return BatchSavingsReport(
        total_current_cost=total_cost,
        sync_cost=sync_cost,
        batch_eligible_cost=batch_eligible_cost,
        batch_discounted_cost=batch_discounted_cost,
        potential_savings=savings,
        savings_pct=(savings / total_cost * 100) if total_cost > 0 else 0,
        batch_eligible_pct=(len(batch_records) / len(all_records) * 100),
        features_to_batch=feature_savings,
    )
```

---

## Handling batch failures and retries

Batch processing introduces failure modes that don't exist in synchronous calls. Individual requests within a batch can fail while others succeed. The batch itself can time out. Your result-processing callback can fail. Here's a robust handler:

```
"""tokenwatch/batch_retry.py — Batch failure handling and retry logic."""

from dataclasses import dataclass, field
from tokenwatch.batch import BatchProcessor, BatchRequest, BatchResult


@dataclass
class BatchRetryPolicy:
    """Configuration for batch retry behavior."""
    max_retries: int = 3
    retry_failed_only: bool = True  # only retry failed requests, not entire
batch
    backoff_multiplier: float = 2.0


def process_batch_results(
    results: list[BatchResult],
    original_requests: dict[str, BatchRequest],
    retry_policy: BatchRetryPolicy | None = None,
) -> tuple[list[BatchResult], list[BatchRequest]]:
    """Process batch results, separating successes and failures.

    Returns:
        (successful_results, requests_to_retry)
    """
    successes = []
    failures = []

    for result in results:
        if result.response is not None:
            successes.append(result)
        else:
            failures.append(result)

    # Build retry list
    requests_to_retry = []
    if failures and retry_policy and retry_policy.max_retries > 0:
        for failure in failures:
            original = original_requests.get(failure.custom_id)
            if original:
                requests_to_retry.append(original)

    return successes, requests_to_retry
```

W> **Warning:** Batch retry logic needs careful attention to idempotency. If a request within a batch has side effects (writes to a database, sends a notification), retrying it can cause duplicates. Design your batch workloads to be idempotent — processing the same request twice should produce the same result without unwanted side effects.

---

## Common mistakes

**1. Batching latency-sensitive requests.** The most common mistake is batching something that needs a real-time response, causing user-visible delays. Be conservative about what goes into batch — default to sync and only move features to batch after confirming latency tolerance. The failure mode: product team complains that classification results "take forever," engineering discovers the feature was accidentally routed through the batch queue, and it takes a week to unwind the architectural change.

**2. Not monitoring batch completion times.** Batch APIs promise completion "within 24 hours" but typically complete much faster. Monitor actual completion times. If your batch processing SLA depends on results within 2 hours and the API takes 4, you need a fallback plan. Track the `created_at` to completion delta in your `BatchJob` records and alert when completion times exceed your SLA.

**3. Ignoring partial batch failures.** A batch of 1,000 requests might have 5 failures. If you only check the batch status (completed) without examining individual results, you'll silently lose those 5 responses. The `process_batch_results` function in `batch_retry.py` separates successes from failures and builds a retry list — always use it rather than assuming all results succeeded.

**4. Over-batching.** Submitting tiny batches (5-10 requests) every few seconds defeats the purpose. The batch API is designed for large volumes with relaxed latency. If your batch-eligible volume is very low, the infrastructure complexity may not justify the savings. A reasonable minimum: at least 100 requests per batch, submitted no more than a few times per hour.

**5. Not accounting for batch in cost reports.** If your metering system records all requests at synchronous rates, your cost reports will overstate actual spend for batch-processed workloads. The `BatchProcessor` in tokenwatch should use `batch=True` when computing costs via `pricing.cost_for_tokens`, so savings appear correctly in dashboards.

**6. Treating batch as all-or-nothing.** Some teams resist batch processing because they think they need to move an entire feature to batch mode. In practice, you can split a single feature's traffic: urgent requests go synchronous, non-urgent requests go to the batch queue. For example, a content moderation feature might process user-reported content synchronously (the reporter expects a response) while batch-processing proactive content scans (nobody is waiting). The `route_request` function supports this per-feature split, and the `BatchQueue` handles the batch side transparently.

---

## Exercises

**Easy:** Create a batch of 10 classification requests using the `BatchProcessor`. Submit the batch, wait for completion, and print the results. Compare the batch cost (using batch rates from `pricing.yaml`) against what the same requests would cost synchronously. Note the completion time — batch requests typically complete in minutes, not hours, despite the 24-hour SLA. Track this latency across multiple submissions to build an empirical distribution of batch turnaround times for your workload.

**Medium:** Implement a `BatchScheduler` that runs as a background thread and periodically (every 5 minutes) flushes the batch queue. It should submit whatever is in the queue, regardless of size. Add logging that shows: batch submitted with N requests, batch completed in X seconds, total cost, savings vs. sync.

**Hard:** Build an adaptive batch router that starts by processing all requests synchronously, measures latency sensitivity per feature (by tracking how quickly callers consume the response), and automatically recommends which features should be moved to batch processing. A feature where the caller doesn't use the result for >60 seconds is a batch candidate.

---

## Summary

In this chapter you learned:

- **Batch processing offers 50% token price discounts** for latency-tolerant workloads, typically saving 15-20% of total API spend
- **Not all workloads are batch-eligible** — real-time user interactions must stay synchronous; document processing, enrichment, and evaluation are prime batch candidates
- **The Anthropic Message Batches API** processes up to 10,000 requests asynchronously with per-request result tracking
- **Batch queues automate collection** — requests accumulate and are submitted when size or time thresholds are met
- **Failure handling and retries** require attention to idempotency and individual result inspection

### Next chapter

You've optimized prompt size (Chapter 5), cached repeated content (Chapter 6), and batched latency-tolerant work (Chapter 7). Chapter 8 introduces the most powerful cost lever: model routing. Instead of using one model for everything, you'll build a cascade that sends easy requests to cheap models and only escalates to expensive models when quality requires it. This single technique can reduce costs by 50-80% on mixed workloads.

---

A> **Code for this chapter:** `project/cap07/tokenwatch/`

# Chapter 8 — Model Routing and Cascades

Here's a question that reveals the biggest cost-saving opportunity in most LLM systems: does every request need your most powerful (and most expensive) model? The answer is almost always no. In a typical production workload, 60-80% of requests can be handled by a model that costs 5-10x less than the top-tier option, with no measurable quality difference.

Model routing — sending different requests to different models based on complexity, task type, or quality requirements — is the highest-leverage cost optimization technique available. In production systems I've worked on, routing has reduced total LLM spend by 50-80% compared to using a single high-end model for everything. The key is building a reliable quality assessment mechanism so you route confidently, not recklessly.

This chapter builds tokenwatch's model router: a cascade system that starts with the cheapest appropriate model and only escalates to more expensive models when the cheap one can't deliver sufficient quality.

**Where this fits in your cost journey**

Model routing is the most powerful single technique in this book, but it depends on the foundation built in earlier chapters. You need metering (Chapter 3) to measure per-request costs across model tiers. You need unit economics (Chapter 4) to understand the cost distribution of your workloads and identify which features have bimodal complexity patterns. You need prompt optimization (Chapter 5) to ensure prompts are already lean before routing — a bloated prompt sent to a cheap model is still wasteful. And you need caching (Chapter 6) to avoid paying full price for the stable prefix regardless of which model tier handles the request. Routing interacts with batch processing (Chapter 7) as well: batch-eligible requests can be routed to cheap models at batch rates, combining the 50% batch discount with the 60-80% routing savings for a total reduction of 80-90% on those workloads.

The key insight is that routing is not about making your system cheaper at the expense of quality. It is about matching model capability to task difficulty. A simple FAQ lookup handled by claude-haiku-4-5 produces the same quality answer as one handled by claude-opus-4-6 — but at a fraction of the cost. Routing saves money by eliminating the overspend on tasks where expensive models add no value.

---

## The routing thesis

Consider a customer support system that handles these types of requests:

| Request type | Volume | Complexity | Required model |
|---|---|---|---|
| Simple FAQ | 60% | Low | Haiku-class |
| Account lookup | 20% | Medium | Haiku/Sonnet |
| Complex troubleshooting | 15% | High | Sonnet |
| Escalation with reasoning | 5% | Very high | Opus/Sonnet w/thinking |

If you use Sonnet for everything, you're paying Sonnet prices on the 60% of requests that Haiku handles perfectly well. The token-price differential between Haiku and Sonnet is roughly 4-5x. Routing 60% of traffic to Haiku while keeping 40% on Sonnet could reduce your total cost by approximately 50%.

The challenge is classifying requests accurately. If you route a complex troubleshooting request to Haiku and it produces a bad answer, you've saved tokens but damaged user experience. The routing system needs to be conservative: when in doubt, send to the better model.

**Worked example: routing savings with pricing.yaml rates**
Using the customer support scenario above with Claude Haiku ($0.80/$4.00 per M input/output) and Claude Sonnet ($3.00/$15.00 per M input/output), at 200,000 daily requests with an average of 1,000 input and 200 output tokens:

```
All Sonnet:
  Input:  200,000 * 1,000 / 1M * $3.00  = $600/day
  Output: 200,000 * 200  / 1M * $15.00  = $600/day
  Total:  $1,200/day → $36,000/month

With routing (60% Haiku, 40% Sonnet):
  Haiku input:   120,000 * 1,000 / 1M * $0.80  = $96/day
  Haiku output:  120,000 * 200  / 1M * $4.00   = $96/day
  Sonnet input:  80,000  * 1,000 / 1M * $3.00  = $240/day
  Sonnet output: 80,000  * 200  / 1M * $15.00  = $240/day
  Total: $672/day → $20,160/month

Monthly savings: $15,840 (44%)
```

Even with a conservative 60/40 split, routing saves 44% of total spend. If the split reaches 80/20 (which is achievable for well-segmented workloads), savings climb to 60%+. The `RoutingPolicy.anthropic_cascade()` pre-configured in tokenwatch provides exactly this three-tier structure.

---

## Building the model router

The router has three components: a complexity classifier, a routing policy, and a quality verifier. Let's build each one.

```python
"""tokenwatch/router.py — Model routing and quality cascades."""

from dataclasses import dataclass, field
from enum import Enum
from typing import Any, Callable

import anthropic

from tokenwatch.config import TokenwatchConfig
from tokenwatch.middleware import TokenMeter
```

```
class Complexity(Enum):
    """Request complexity levels."""
    LOW = "low"
    MEDIUM = "medium"
    HIGH = "high"
    CRITICAL = "critical"


@dataclass
class RoutingDecision:
    """The result of a routing decision."""
    model: str
    complexity: Complexity
    reason: str
    estimated_cost_ratio: float  # vs. most expensive model


@dataclass
class ModelTier:
    """A tier in the model cascade."""
    model_id: str
    max_complexity: Complexity
    cost_ratio: float  # relative to the most expensive tier
    description: str


@dataclass
class RoutingPolicy:
    """Defines the model cascade and routing rules."""
    tiers: list[ModelTier]
    default_tier: int = 0  # index into tiers (cheapest first)

    @classmethod
    def anthropic_cascade(cls) -> "RoutingPolicy":
        """Pre-configured cascade for Anthropic models."""
        return cls(tiers=[
            ModelTier(
                model_id="claude-haiku-4-5-20250514",
                max_complexity=Complexity.LOW,
                cost_ratio=0.15,
                description="Fast, cheap — simple classification, extraction,
FAQ",
            ),
            ModelTier(
                model_id="claude-sonnet-4-20250514",
                max_complexity=Complexity.HIGH,
                cost_ratio=0.40,
```

```
                description="Balanced — most tasks, good quality/cost trade-off",
            ),
            ModelTier(
                model_id="claude-opus-4-20250514",
                max_complexity=Complexity.CRITICAL,
                cost_ratio=1.0,
                description="Most capable — complex reasoning, high-stakes
tasks",
            ),
        ])

    @classmethod
    def openai_cascade(cls) -> "RoutingPolicy":
        """Pre-configured cascade for OpenAI models."""
        return cls(tiers=[
            ModelTier(
                model_id="gpt-4o-mini",
                max_complexity=Complexity.LOW,
                cost_ratio=0.05,
                description="Fast, cheap — simple tasks",
            ),
            ModelTier(
                model_id="gpt-4o",
                max_complexity=Complexity.CRITICAL,
                cost_ratio=1.0,
                description="Most capable GPT-4 class model",
            ),
        ])
```

---

## Complexity classification

The classifier determines request complexity. There are three approaches, from simplest to most sophisticated. The right choice depends on your workload's complexity distribution and your tolerance for misclassification.

In practice, most production systems start with feature-based classification (Approach 2 below) because it requires zero runtime cost, is completely predictable, and aligns with how teams already think about their workloads. A ticket classifier always goes to Haiku. A legal analysis tool always goes to Sonnet or Opus. The feature name is a reliable proxy for complexity when each feature has a relatively uniform complexity profile. Rule-based classification (Approach 1) adds within-feature granularity at zero cost — useful for high-volume features like chat agents where complexity varies significantly between requests. LLM-based classification (Approach 3) is the most accurate but adds a per-request cost and latency. Reserve it for workloads where the savings from better classification exceed the cost of the classifier itself.

Here are the three approaches:

**Approach 1: Rule-based classification**

Fast, free, and predictable. Works well when you can define clear rules based on input characteristics.

```
"""tokenwatch/classifier.py — Request complexity classification."""

import re
from tokenwatch.router import Complexity


def classify_by_rules(
    messages: list[dict],
    system: str = "",
    tool_count: int = 0,
    history_turns: int = 0,
) -> Complexity:
    """Classify request complexity using heuristic rules.

    Rules are based on observable request characteristics:
    - Message length (proxy for task complexity)
    - Number of conversation turns (multi-turn = more complex)
    - Presence of tools (tool use = more complex)
    - Content patterns (questions vs. commands vs. analysis)
    """
    # Extract the user's message
    user_msg = ""
    for msg in reversed(messages):
        if msg.get("role") == "user":
            content = msg.get("content", "")
            user_msg = content if isinstance(content, str) else str(content)
            break

    msg_length = len(user_msg)

    # Rule 1: Very short messages are usually simple
    if msg_length < 100 and tool_count == 0 and history_turns < 3:
        return Complexity.LOW

    # Rule 2: Presence of analysis/reasoning keywords → higher complexity
    complex_patterns = [
        r"\b(analyze|compare|evaluate|explain why|reason|trade-?off)\b",
        r"\b(step by step|think through|consider)\b",
        r"\b(complex|nuanced|detailed analysis)\b",
    ]
    has_complex_keywords = any(
        re.search(p, user_msg, re.IGNORECASE)
        for p in complex_patterns
    )
```

```
    # Rule 3: Many tools available → medium+ complexity
    if tool_count > 5:
        return Complexity.HIGH if has_complex_keywords else Complexity.MEDIUM

    # Rule 4: Long conversation history → medium+ complexity
    if history_turns > 10:
        return Complexity.MEDIUM

    # Rule 5: Long messages with complex keywords → high
    if msg_length > 500 and has_complex_keywords:
        return Complexity.HIGH

    # Rule 6: Medium-length messages → medium
    if msg_length > 200 or has_complex_keywords:
        return Complexity.MEDIUM

    return Complexity.LOW


def classify_by_feature(
    feature: str,
    feature_complexity_map: dict[str, Complexity] | None = None,
) -> Complexity:
    """Classify complexity based on the feature name.

    The simplest and most reliable approach: you know your features
    and their complexity. Map them explicitly.
    """
    default_map = {
        "ticket-classifier": Complexity.LOW,
        "faq-bot": Complexity.LOW,
        "sentiment-analysis": Complexity.LOW,
        "chat-agent": Complexity.MEDIUM,
        "code-review": Complexity.HIGH,
        "legal-analysis": Complexity.CRITICAL,
        "medical-triage": Complexity.CRITICAL,
    }

    mapping = feature_complexity_map or default_map
    return mapping.get(feature, Complexity.MEDIUM)
```

**Approach 2: Cheap-model classification**

Use a cheap model (Haiku) to assess whether a request needs a more capable model. This costs a few tokens but is more accurate than rules.

```
def classify_with_llm(
    messages: list[dict],
    client: anthropic.Anthropic,
    classifier_model: str = "claude-haiku-4-5-20250514",
```

```
) -> Complexity:
    """Use a cheap model to classify request complexity.

    The classifier prompt is minimal to keep cost near-zero.
    """
    user_msg = ""
    for msg in reversed(messages):
        if msg.get("role") == "user":
            content = msg.get("content", "")
            user_msg = content if isinstance(content, str) else str(content)
            break

    response = client.messages.create(
        model=classifier_model,
        max_tokens=10,
        messages=[{
            "role": "user",
            "content": (
                f"Rate this query's complexity: low, medium, high, or critical.
\n"
                f"Respond with ONE word only.\n\n"
                f"Query: {user_msg[:500]}"
            ),
        }],
    )

    result = response.content[0].text.strip().lower()

    mapping = {
        "low": Complexity.LOW,
        "medium": Complexity.MEDIUM,
        "high": Complexity.HIGH,
        "critical": Complexity.CRITICAL,
    }

    return mapping.get(result, Complexity.MEDIUM)
```

I> **Note:** The LLM classifier adds a small cost per request (roughly 100-200 tokens on a cheap model). This is almost always worthwhile: if the classifier correctly routes 60% of traffic from Sonnet to Haiku, the per-request savings on those routed requests dwarf the classifier cost.

---

## The routing engine

Now let's combine classification with routing:

```
"""tokenwatch/routing_engine.py — The main routing engine."""

import time
from dataclasses import dataclass
```

```
from typing import Any

import anthropic

from tokenwatch.router import RoutingPolicy, RoutingDecision, Complexity,
ModelTier
from tokenwatch.classifier import classify_by_rules, classify_by_feature
from tokenwatch.config import TokenwatchConfig
from tokenwatch.middleware import TokenMeter
from tokenwatch.meter import UsageRecord


class RoutingEngine:
    """Routes requests to the optimal model based on complexity."""

    def __init__(
        self,
        config: TokenwatchConfig,
        policy: RoutingPolicy,
        meter: TokenMeter | None = None,
        client: anthropic.Anthropic | None = None,
    ):
        self.config = config
        self.policy = policy
        self.meter = meter
        self.client = client or anthropic.Anthropic()

    def route(
        self,
        messages: list[dict],
        system: str = "",
        feature: str = "",
        complexity_override: Complexity | None = None,
        tool_count: int = 0,
        history_turns: int = 0,
    ) -> RoutingDecision:
        """Determine which model should handle this request.

        Uses the complexity classification and routing policy
        to select the cheapest model that can handle the request.
        """
        if complexity_override:
            complexity = complexity_override
        elif feature:
            complexity = classify_by_feature(feature)
        else:
            complexity = classify_by_rules(
                messages, system, tool_count, history_turns
            )
```

```
        # Find the cheapest tier that handles this complexity
        complexity_order = [
            Complexity.LOW, Complexity.MEDIUM,
            Complexity.HIGH, Complexity.CRITICAL,
        ]
        request_level = complexity_order.index(complexity)

        selected_tier = self.policy.tiers[-1]  # default to most capable
        for tier in self.policy.tiers:
            tier_level = complexity_order.index(tier.max_complexity)
            if tier_level >= request_level:
                selected_tier = tier
                break

        return RoutingDecision(
            model=selected_tier.model_id,
            complexity=complexity,
            reason=f"Complexity={complexity.value} →
{selected_tier.description}",
            estimated_cost_ratio=selected_tier.cost_ratio,
        )

    def route_and_call(
        self,
        messages: list[dict],
        system: str = "",
        max_tokens: int = 4096,
        feature: str = "",
        user_id: str = "",
        team: str = "",
        **kwargs,
    ) -> tuple[anthropic.types.Message, RoutingDecision, UsageRecord | None]:
        """Route the request and execute the API call.

        Returns (response, routing_decision, usage_record).
        """
        decision = self.route(
            messages=messages,
            system=system,
            feature=feature,
            tool_count=len(kwargs.get("tools", [])),
        )

        start = time.monotonic()
        response = self.client.messages.create(
            model=decision.model,
            max_tokens=max_tokens,
            system=system,
```

```
            messages=messages,
            **kwargs,
        )
        latency_ms = (time.monotonic() - start) * 1000

        record = None
        if self.meter:
            record = self.meter.measure_anthropic(
                response=response,
                latency_ms=latency_ms,
                feature=feature,
                user_id=user_id,
                team=team,
            )

        return response, decision, record
```

---

## Quality-gated cascades

The most sophisticated routing pattern is the quality-gated cascade: start with the cheapest model, check the quality of the response, and escalate to a better model only if the quality is insufficient.

```
"""tokenwatch/cascade.py — Quality-gated model cascades."""

from dataclasses import dataclass
from typing import Any, Callable

import anthropic

from tokenwatch.router import RoutingPolicy, Complexity
from tokenwatch.config import TokenwatchConfig


@dataclass
class CascadeResult:
    """Result from a cascaded request."""
    response: anthropic.types.Message
    model_used: str
    models_tried: list[str]
    escalated: bool
    escalation_reason: str
    total_cost_usd: float


class QualityCascade:
    """Cascading model selection with quality verification.

    Starts with the cheapest model. If a quality check fails,
    escalates to the next tier. This ensures you only pay for
```

```
    expensive models when cheap ones can't deliver.
    """

    def __init__(
        self,
        config: TokenwatchConfig,
        policy: RoutingPolicy,
        quality_check: Callable[[anthropic.types.Message], bool] | None = None,
        client: anthropic.Anthropic | None = None,
    ):
        self.config = config
        self.policy = policy
        self.quality_check = quality_check or self._default_quality_check
        self.client = client or anthropic.Anthropic()

    def _default_quality_check(self, response: anthropic.types.Message) -> bool:
        """Default quality check: passes if response is non-empty
        and didn't hit max_tokens truncation."""
        if not response.content:
            return False
        if response.stop_reason == "max_tokens":
            return False  # truncated response is low quality
        text = response.content[0].text if hasattr(response.content[0], 'text')
else ""
        if len(text.strip()) < 10:
            return False  # suspiciously short
        return True

    def execute(
        self,
        messages: list[dict],
        system: str = "",
        max_tokens: int = 4096,
        start_tier: int = 0,
        **kwargs,
    ) -> CascadeResult:
        """Execute a request through the cascade.

        Starts at start_tier (default: cheapest) and escalates
        through tiers until quality passes or all tiers exhausted.
        """
        models_tried = []
        total_cost = 0.0

        for i, tier in enumerate(self.policy.tiers[start_tier:],
start=start_tier):
            model = tier.model_id
            models_tried.append(model)
```

```
            response = self.client.messages.create(
                model=model,
                max_tokens=max_tokens,
                system=system,
                messages=messages,
                **kwargs,
            )

            # Calculate cost for this attempt
            pricing = self.config.get_model(model)
            cost = pricing.cost_for_tokens(
                input_tokens=response.usage.input_tokens,
                output_tokens=response.usage.output_tokens,
            )
            total_cost += cost

            # Check quality
            if self.quality_check(response):
                return CascadeResult(
                    response=response,
                    model_used=model,
                    models_tried=models_tried,
                    escalated=i > start_tier,
                    escalation_reason="" if i == start_tier else "Quality check
failed on cheaper model",
                    total_cost_usd=total_cost,
                )

        # All tiers exhausted — return the last response
        return CascadeResult(
            response=response,
            model_used=models_tried[-1],
            models_tried=models_tried,
            escalated=True,
            escalation_reason="All tiers attempted, returning best available",
            total_cost_usd=total_cost,
        )
```

W> **Warning:** Quality cascades involve calling multiple models when the cheap one fails. In the worst case (every request escalates through all tiers), you pay for multiple API calls — which can cost *more* than just using the expensive model directly. Monitor your escalation rate. If more than 20-30% of requests escalate, your initial routing is too aggressive and you should adjust the complexity thresholds.

---

## Routing analytics

Track routing decisions to understand how traffic distributes across models and whether routing is actually saving money:

```
"""tokenwatch/routing_analytics.py — Analyze routing effectiveness."""

from collections import defaultdict
from tokenwatch.store import UsageStore


def routing_report(store: UsageStore, since: str | None = None) -> dict:
    """Analyze how traffic is distributed across models."""
    records = store.query(since=since)
    if not records:
        return {}

    by_model = defaultdict(lambda: {"count": 0, "cost": 0.0, "tokens": 0})
    total_cost = 0.0

    for r in records:
        entry = by_model[r.model]
        entry["count"] += 1
        entry["cost"] += r.cost_usd
        entry["tokens"] += r.input_tokens + r.output_tokens
        total_cost += r.cost_usd

    report = {"total_cost": total_cost, "models": {}}
    for model, data in sorted(by_model.items(), key=lambda x: x[1]["cost"],
reverse=True):
        report["models"][model] = {
            "requests": data["count"],
            "cost": data["cost"],
            "pct_of_total": data["cost"] / total_cost * 100 if total_cost > 0
else 0,
            "avg_cost_per_request": data["cost"] / data["count"],
        }

    return report
```

## Evaluating router quality over time

Deploying a routing policy is not a one-time event. The router makes thousands of decisions per day, and each decision is either correct (the chosen model handled the request well) or incorrect (the chosen model produced a low-quality response that should have been handled by a higher tier). You need an offline evaluation discipline to track routing quality over time — and this discipline connects directly to the evaluation practices discussed in the broader quality assurance of your LLM system.

The simplest approach is to sample routed requests weekly, send each sample to both the model that actually handled it and the next-higher model tier, and compare the outputs using an automated quality scorer (or human evaluation for high-stakes features). If the higher-tier model produces meaningfully better responses on more than 10-15% of samples, your routing is too

aggressive — you are saving money at the expense of quality. If the higher-tier model produces equivalent responses on 95% or more of samples, your routing may be too conservative — you could route more traffic to the cheap tier and save additional money. This evaluation loop is the mechanism that keeps routing calibrated as model capabilities evolve and your workload mix changes.

Track routing evaluation metrics alongside your cost metrics in the same dashboard. A routing decision that saves 5x on per-request cost but degrades quality on 20% of requests is not a net positive — those degraded responses generate user complaints, support tickets, and retry attempts that may cost more than the routing savings. The `routing_report` from `routing_analytics.py` shows traffic distribution across tiers; pair it with quality evaluation data to build a complete picture of routing effectiveness.

---

## Common mistakes

**1. Routing based on cost alone.** The cheapest model is not always the right model. A misclassified complex request sent to Haiku produces a wrong answer, which the user retries, which costs more than sending it to Sonnet in the first place. Accuracy matters more than initial cost. The failure mode is insidious: quality degrades gradually, users learn not to trust the feature, and adoption drops — a cost that doesn't appear on any API bill.

**2. Not measuring escalation rates.** If your cascade escalates 50% of the time, you're paying for two API calls on half your traffic. Monitor escalation rates and adjust classification thresholds to keep escalation below 15-20%. Use the `routing_report` function from `routing_analytics.py` to track the distribution of traffic across model tiers.

**3. Hardcoding model selection per feature.** Feature-based routing is a good starting point, but request complexity within a feature varies. A chat agent handles both "what are your hours?" (low) and "compare the performance of these three investment strategies" (high). The `classify_by_rules` function adds within-feature classification based on message length, keywords, and tool count — use it for high-volume features where the complexity distribution is wide.

**4. Ignoring model capability differences.** Some tasks require capabilities that cheap models simply don't have: complex tool calling, long-context reasoning, precise instruction following. Route based on task requirements, not just "this looks short enough for Haiku." The `Complexity.CRITICAL` level in our routing policy exists for this reason — it routes directly to the most capable model regardless of message length.

**5. Not A/B testing routing changes.** When you change routing thresholds or add a new model tier, run both configurations in parallel on a fraction of traffic and compare quality metrics. Routing is an optimization that can silently degrade the product if applied carelessly.

**6. No organizational ownership of routing rules.** Routing rules are a shared concern between product, engineering, and data science, but they often end up owned by nobody. Product defines quality requirements, engineering implements the routing logic, and data science evaluates

whether the cheaper model meets those requirements — but unless someone owns the routing configuration as a living artifact, rules drift out of date. Assign explicit ownership: one team or one person who reviews routing rules quarterly, updates them when new models are released, and signs off on changes to complexity thresholds. Without this ownership, routing rules ossify, and you miss opportunities to route more traffic to cheaper tiers as model capabilities improve.

**7. Ignoring cascade UX implications.** When a quality-gated cascade escalates from a cheap model to an expensive one, the user experiences the combined latency of both calls. If Haiku responds in 500 milliseconds and the quality check fails, the user then waits for Sonnet's response — a total of 1.5 to 2 seconds instead of the 1 second they would have waited with Sonnet alone. For interactive features where perceived speed matters, this added latency can degrade the user experience even though the final answer quality is identical. Monitor escalation latency separately from single-model latency, and consider whether the UX can absorb the extra delay. For some features, it may be better to route directly to the mid-tier model rather than risk the cascade penalty.

**8. Treating routing as static.** Model capabilities change with every provider release. When Anthropic releases a new version of claude-haiku-4-5 or claude-sonnet-4-6, the quality gap between tiers may shift, meaning your routing thresholds may need recalibration. Schedule a quarterly review of your routing policy using the `routing_report` from `routing_analytics.py` — check the per-tier quality metrics and adjust the complexity boundaries if the cheap tier's accuracy has improved (route more traffic to it) or degraded (route less). Provider model updates are also an opportunity to re-evaluate the cost ratios in your `ModelTier` definitions, since pricing may change alongside capability.

---

## Exercises

**Easy:** Configure a `RoutingPolicy.anthropic_cascade()` and route 10 sample requests with varying complexity through the `RoutingEngine`. Print the routing decision for each and calculate the total cost vs. using Sonnet for all requests. Pay attention to how the `classify_by_rules` function categorizes your sample requests — you may find that messages you consider "simple" are classified as "medium" due to length or keyword triggers, and vice versa. This mismatch between your intuition and the classifier's output is exactly what you need to calibrate before deploying routing in production.

**Medium:** Build a custom `quality_check` function for a classification task that verifies the model's output is one of the expected categories. Use it with `QualityCascade` to route classification requests — start with Haiku and escalate to Sonnet only when Haiku returns an invalid category. Measure the escalation rate and net cost savings.

**Hard:** Implement an adaptive routing system that tracks historical accuracy per model per feature. Over time, it learns which features can be reliably handled by cheap models and adjusts routing thresholds automatically. Use a sliding window of the last 1,000 requests per feature to compute accuracy, and only route to the cheap model if its accuracy exceeds 95% for that feature.

---

## Summary

In this chapter you learned:

- **Model routing is the highest-leverage cost optimization** — routing 60-80% of traffic to cheaper models can reduce total spend by 50-80%
- **Complexity classification** can be rule-based (free, fast), feature-based (simple, reliable), or LLM-based (accurate, small cost)
- **The cascade pattern** starts cheap and escalates — you only pay for expensive models when cheap ones can't deliver
- **Quality-gated cascades** add automatic quality verification, ensuring routing never degrades user experience
- **Escalation rate monitoring** is critical — high escalation rates mean your routing is too aggressive and may increase total cost

**Next chapter**

Routing optimizes which model handles each request. Chapter 9 focuses on a specific workload type that deserves its own optimization strategy: RAG (Retrieval Augmented Generation). RAG systems have unique cost drivers — embeddings, retrieval, context injection — and unique optimization opportunities. We'll build tokenwatch's RAG cost module to measure and minimize the cost of knowledge-augmented generation.

---

A> **Code for this chapter:** `project/cap08/tokenwatch/`

# Chapter 9 — RAG Cost Optimization

RAG — Retrieval Augmented Generation — is one of the most popular patterns for giving LLMs access to private or up-to-date knowledge. It's also one of the most expensive when done carelessly. A naive RAG pipeline retrieves too many chunks, stuffs them all into the context window, and pays full price for thousands of tokens of retrieved content — most of which the model ignores.

The cost equation for RAG has multiple components: embedding generation, vector storage, retrieval computation, and the LLM generation itself. Each component has its own optimization levers. In this chapter, we'll build tokenwatch's RAG cost module and learn how to measure, analyze, and reduce the cost of knowledge-augmented generation.

**Where this fits in your cost journey**

RAG occupies a unique position in the optimization stack because it introduces a cost layer that does not exist in non-RAG features: the retrieved context. Every chunk injected into the prompt is input tokens that you pay for, and unlike system prompts or tool definitions, the retrieved chunks change per request — which means they resist caching (Chapter 6). The primary cost lever for RAG is therefore not caching but context volume: how many chunks you retrieve and how large each chunk is. This chapter's optimization techniques interact directly with prompt compression (Chapter 5), because shorter chunks mean fewer input tokens per query. They also interact with model routing (Chapter 8), because simple factual lookups against retrieved context can often be handled by claude-haiku-4-5, while complex analytical queries that synthesize across multiple

chunks may require claude-sonnet-4-6. The combination of reduced context size, cheaper model routing, and caching of the stable system prompt prefix can reduce RAG feature costs by 70% or more compared to a naive implementation.

---

## The RAG cost stack

A typical RAG pipeline has four cost layers, and most teams only think about the last one:

**Layer 1: Embedding generation.** Every document chunk must be converted to a vector embedding. Embedding models are cheap per-token (fractions of a cent per million tokens), but at scale — millions of documents, re-indexed weekly — the cost adds up.

**Layer 2: Vector storage and retrieval.** Vector databases (Pinecone, Weaviate, Qdrant, pgvector) charge for storage and query operations. This is typically a small cost relative to LLM generation, but it's non-zero and grows with index size.

**Layer 3: Reranking.** Many pipelines use a reranker (Cohere, cross-encoder models) to re-score retrieved chunks before injection. This adds a per-query cost.

**Layer 4: LLM generation.** The retrieved chunks are injected into the prompt as context. This is usually the dominant cost, because retrieved chunks can easily add 2,000-10,000 tokens of input per request.

### Worked example: RAG cost breakdown

Consider a customer FAQ system running on Claude Sonnet ($3.00/$15.00 per M) with 10,000 daily queries, using OpenAI `text-embedding-3-small` ($0.02/M tokens) for embeddings and top-k=5 retrieval with 500-token chunks:

```
Layer 1 — Embedding (query only): 10,000 * 50 tokens / 1M * $0.02  = $0.01/day
Layer 2 — Vector storage (Pinecone, 100K vectors):                  ~$2.00/day
Layer 3 — No reranker in this setup:                                 $0.00/day
Layer 4 — LLM generation:
  Context tokens: 5 chunks * 500 tokens = 2,500 per query
  Query tokens:   50 per query
  Total input:    2,550 * 10,000 / 1M * $3.00                      = $76.50/day
  Output:         200 * 10,000 / 1M * $15.00                       = $30.00/day
  Subtotal:                                                         $106.50/day

Total: $108.51/day → $3,255/month
```

The LLM generation layer is 98% of total cost. This is why context optimization (reducing chunks or chunk size) has 20x more impact than switching to a cheaper embedding model. The `RAGCostTracker` in tokenwatch automates this breakdown and highlights the dominant cost layer.

```
"""tokenwatch/rag.py — RAG cost tracking and optimization."""

from dataclasses import dataclass, field
from tokenwatch.config import TokenwatchConfig
```

```
from tokenwatch.tokens import estimate_tokens


@dataclass
class RAGCostBreakdown:
    """Cost breakdown for a single RAG request."""
    # Retrieval costs
    embedding_tokens: int = 0
    embedding_cost_usd: float = 0.0
    chunks_retrieved: int = 0
    chunks_after_rerank: int = 0
    rerank_cost_usd: float = 0.0

    # Context injection costs
    context_tokens: int = 0
    context_cost_usd: float = 0.0

    # Generation costs
    query_tokens: int = 0
    output_tokens: int = 0
    generation_cost_usd: float = 0.0

    # Totals
    total_cost_usd: float = 0.0
    context_waste_pct: float = 0.0  # estimated % of context not used

    @property
    def retrieval_pct(self) -> float:
        if self.total_cost_usd == 0:
            return 0.0
        return (self.embedding_cost_usd + self.rerank_cost_usd) /
self.total_cost_usd * 100

    @property
    def context_pct(self) -> float:
        if self.total_cost_usd == 0:
            return 0.0
        return self.context_cost_usd / self.total_cost_usd * 100


class RAGCostTracker:
    """Tracks and analyzes RAG pipeline costs."""

    def __init__(
        self,
        config: TokenwatchConfig,
        embedding_model: str = "text-embedding-3-small",
        embedding_rate_per_million: float = 0.02,
        rerank_rate_per_query: float = 0.0005,
```

```
    ):
        self.config = config
        self.embedding_model = embedding_model
        self.embedding_rate = embedding_rate_per_million
        self.rerank_rate = rerank_rate_per_query
        self.history: list[RAGCostBreakdown] = []

    def track_request(
        self,
        query: str,
        retrieved_chunks: list[str],
        reranked_chunks: list[str] | None = None,
        llm_model: str = "claude-sonnet-4-20250514",
        output_tokens: int = 500,
    ) -> RAGCostBreakdown:
        """Track costs for a complete RAG request.

        Args:
            query: The user's query
            retrieved_chunks: All chunks retrieved from the vector store
            reranked_chunks: Chunks after reranking (subset of retrieved)
            llm_model: Model used for generation
            output_tokens: Actual or estimated output tokens
        """
        # Embedding cost (for the query — document embeddings are a one-time
cost)
        query_tokens_count = estimate_tokens(query)
        embedding_cost = (query_tokens_count / 1_000_000) * self.embedding_rate

        # Reranking cost
        used_chunks = reranked_chunks if reranked_chunks else retrieved_chunks
        rerank_cost = self.rerank_rate if reranked_chunks else 0.0

        # Context tokens (the chunks injected into the prompt)
        context_tokens = sum(estimate_tokens(chunk) for chunk in used_chunks)

        # Generation cost
        pricing = self.config.get_model(llm_model)
        total_input = query_tokens_count + context_tokens
        generation_cost = pricing.cost_for_tokens(
            input_tokens=total_input,
            output_tokens=output_tokens,
        )

        # Context cost (just the chunk tokens at input rate)
        context_cost = (context_tokens / 1_000_000) * pricing.input_rate

        # Estimate context waste (heuristic: chunks beyond top-3 are often
unused)
```

```
        if len(used_chunks) > 3:
            extra_tokens = sum(estimate_tokens(c) for c in used_chunks[3:])
            waste_pct = (extra_tokens / context_tokens * 100) if context_tokens >
0 else 0
        else:
            waste_pct = 0.0

        total = embedding_cost + rerank_cost + generation_cost

        breakdown = RAGCostBreakdown(
            embedding_tokens=query_tokens_count,
            embedding_cost_usd=embedding_cost,
            chunks_retrieved=len(retrieved_chunks),
            chunks_after_rerank=len(used_chunks),
            rerank_cost_usd=rerank_cost,
            context_tokens=context_tokens,
            context_cost_usd=context_cost,
            query_tokens=query_tokens_count,
            output_tokens=output_tokens,
            generation_cost_usd=generation_cost,
            total_cost_usd=total,
            context_waste_pct=waste_pct,
        )

        self.history.append(breakdown)
        return breakdown
```

---

## Chunking strategy and cost

How you chunk documents directly affects RAG cost. Larger chunks mean fewer retrieval results but more tokens per chunk. Smaller chunks mean better precision but potentially more chunks needed for complete answers.

```
"""tokenwatch/chunking.py — Token-aware chunking strategies."""

from dataclasses import dataclass
from tokenwatch.tokens import estimate_tokens


@dataclass
class ChunkingStrategy:
    """Configuration for document chunking."""
    chunk_size_tokens: int = 500    # target tokens per chunk
    chunk_overlap_tokens: int = 50  # overlap between adjacent chunks
    top_k: int = 5                  # chunks to retrieve per query


def estimate_chunking_cost(
    document_tokens: int,
```

```
    strategy: ChunkingStrategy,
    embedding_rate_per_million: float = 0.02,
) -> dict:
    """Estimate the cost of chunking and embedding a document.

    Returns cost breakdown including embedding and storage estimates.
    """
    effective_chunk_size = strategy.chunk_size_tokens -
strategy.chunk_overlap_tokens
    num_chunks = max(1, document_tokens // effective_chunk_size)

    # Total tokens to embed (chunks have overlap, so total > document)
    total_embedding_tokens = num_chunks * strategy.chunk_size_tokens
    embedding_cost = (total_embedding_tokens / 1_000_000) *
embedding_rate_per_million

    return {
        "document_tokens": document_tokens,
        "num_chunks": num_chunks,
        "chunk_size_tokens": strategy.chunk_size_tokens,
        "total_embedding_tokens": total_embedding_tokens,
        "embedding_cost_usd": embedding_cost,
        "tokens_per_query": strategy.chunk_size_tokens * strategy.top_k,
        "note": (
            f"Each query injects ~{strategy.chunk_size_tokens * strategy.top_k:,}
"
            f"context tokens ({strategy.top_k} chunks x "
            f"{strategy.chunk_size_tokens} tokens/chunk)"
        ),
    }


def optimize_chunking(
    query_volume_daily: int,
    avg_document_tokens: int,
    document_count: int,
    llm_model_input_rate: float,
    embedding_rate: float = 0.02,
) -> list[dict]:
    """Compare chunking strategies and their cost impact.

    Returns a comparison of different chunk sizes showing the
    trade-off between retrieval precision and query cost.
    """
    strategies = [
        ChunkingStrategy(chunk_size_tokens=200, chunk_overlap_tokens=20,
top_k=8),
        ChunkingStrategy(chunk_size_tokens=500, chunk_overlap_tokens=50,
top_k=5),
```

```
        ChunkingStrategy(chunk_size_tokens=1000, chunk_overlap_tokens=100,
top_k=3),
        ChunkingStrategy(chunk_size_tokens=2000, chunk_overlap_tokens=200,
top_k=2),
    ]

    results = []
    for strategy in strategies:
        tokens_per_query = strategy.chunk_size_tokens * strategy.top_k

        # Cost per query (just the context tokens at LLM input rate)
        context_cost_per_query = (tokens_per_query / 1_000_000) *
llm_model_input_rate

        # Daily context cost
        daily_context_cost = context_cost_per_query * query_volume_daily

        # One-time embedding cost
        total_doc_tokens = avg_document_tokens * document_count
        embedding_info = estimate_chunking_cost(
            total_doc_tokens, strategy, embedding_rate
        )

        results.append({
            "chunk_size": strategy.chunk_size_tokens,
            "overlap": strategy.chunk_overlap_tokens,
            "top_k": strategy.top_k,
            "tokens_per_query": tokens_per_query,
            "context_cost_per_query": context_cost_per_query,
            "daily_context_cost": daily_context_cost,
            "monthly_context_cost": daily_context_cost * 30,
            "total_chunks": embedding_info["num_chunks"],
            "embedding_cost": embedding_info["embedding_cost_usd"],
        })

    return sorted(results, key=lambda x: x["monthly_context_cost"])
```

T> **Tip:** The optimal chunk size is workload-specific. Factual question-answering benefits from smaller chunks (200-500 tokens) with higher top-k, because precision matters. Summarization and analysis benefit from larger chunks (1,000-2,000 tokens) with lower top-k, because the model needs broader context. Run the `optimize_chunking` comparison with your actual workload parameters.

---

## Reducing retrieval waste

The biggest cost lever in RAG is reducing the number of tokens injected into the context. This is where the 98% cost dominance of the LLM generation layer translates into optimization opportunity: every token you remove from the injected context saves money at the LLM input

rate, which is orders of magnitude more expensive than the embedding rate. A 30% reduction in context tokens translates to a 30% reduction in the dominant cost component.

The challenge is that reducing context tokens can degrade answer quality if relevant information is excluded. The techniques below are ordered by their safety profile — the first techniques (reducing top-k, using rerankers) have minimal quality risk because they remove low-relevance chunks, while the later techniques (summarization) require more careful quality testing. Here are four techniques:

**1. Reduce top-k**
The simplest optimization: retrieve fewer chunks. Most naive pipelines use top-k=10 or even top-k=20. In practice, the top 3-5 chunks contain the vast majority of relevant information. Reducing from top-k=10 to top-k=3 cuts context tokens by 70%.

**2. Use a reranker to filter**
Retrieve more chunks initially (top-k=20 from the vector store) but then use a reranker to select only the best 3-5. The reranker cost is minimal compared to the LLM cost savings from injecting fewer chunks.

**3. Chunk summarization**
Instead of injecting full chunks, summarize each chunk before injection. A 500-token chunk might be condensed to a 100-token summary. This costs a small amount for the summarization call but saves 400 tokens of LLM input on every subsequent query that retrieves this chunk.

**4. Relevance thresholding**
Don't inject chunks below a minimum similarity score. If your top-k=5 retrieval returns chunks with similarity scores [0.92, 0.87, 0.81, 0.45, 0.32], the last two are probably noise. Set a threshold (e.g., 0.7) and only inject chunks above it.

```
"""tokenwatch/rag_optimizer.py — RAG cost optimization strategies."""

from tokenwatch.tokens import estimate_tokens


def filter_by_relevance(
    chunks: list[dict],
    min_score: float = 0.7,
    max_chunks: int = 5,
) -> list[dict]:
    """Filter retrieved chunks by relevance score.

    Args:
        chunks: List of {"text": str, "score": float}
        min_score: Minimum similarity score to include
        max_chunks: Maximum chunks to return
    """
    filtered = [c for c in chunks if c.get("score", 0) >= min_score]
    filtered.sort(key=lambda c: c.get("score", 0), reverse=True)
```

```
    return filtered[:max_chunks]


def estimate_rag_savings(
    current_top_k: int,
    current_chunk_size: int,
    optimized_top_k: int,
    optimized_chunk_size: int,
    daily_queries: int,
    input_rate_per_million: float,
) -> dict:
    """Estimate savings from RAG optimization.

    Compares current vs. optimized context injection costs.
    """
    current_tokens = current_top_k * current_chunk_size
    optimized_tokens = optimized_top_k * optimized_chunk_size

    current_daily = (current_tokens / 1_000_000) * input_rate_per_million *
daily_queries
    optimized_daily = (optimized_tokens / 1_000_000) * input_rate_per_million *
daily_queries

    savings_daily = current_daily - optimized_daily

    return {
        "current_tokens_per_query": current_tokens,
        "optimized_tokens_per_query": optimized_tokens,
        "token_reduction_pct": (1 - optimized_tokens / current_tokens) * 100,
        "daily_savings_usd": savings_daily,
        "monthly_savings_usd": savings_daily * 30,
        "annual_savings_usd": savings_daily * 365,
    }
```

---

## Embedding model cost comparison

Embedding models vary widely in cost and quality. The choice of embedding model affects both the index quality (which impacts retrieval precision) and the indexing cost (which matters for large document collections).

```
"""tokenwatch/embedding_costs.py — Embedding model cost comparison."""

from dataclasses import dataclass


@dataclass
class EmbeddingModel:
    """An embedding model with its cost characteristics."""
    name: str
```

```
    provider: str
    dimensions: int
    rate_per_million_tokens: float
    max_tokens: int


# Common embedding models (rates are examples — check current pricing)
EMBEDDING_MODELS = [
    EmbeddingModel("text-embedding-3-small", "openai", 1536, 0.02, 8191),
    EmbeddingModel("text-embedding-3-large", "openai", 3072, 0.13, 8191),
    EmbeddingModel("voyage-3", "voyageai", 1024, 0.06, 32000),
    EmbeddingModel("voyage-3-lite", "voyageai", 512, 0.02, 32000),
]


def compare_embedding_costs(
    document_count: int,
    avg_tokens_per_doc: int,
    daily_query_volume: int,
    avg_query_tokens: int = 50,
) -> list[dict]:
    """Compare embedding costs across models.

    Includes both indexing (one-time) and query (ongoing) costs.
    """
    results = []
    for model in EMBEDDING_MODELS:
        # Indexing cost (one-time)
        total_doc_tokens = document_count * avg_tokens_per_doc
        index_cost = (total_doc_tokens / 1_000_000) *
model.rate_per_million_tokens

        # Query cost (daily)
        daily_query_tokens = daily_query_volume * avg_query_tokens
        daily_query_cost = (daily_query_tokens / 1_000_000) *
model.rate_per_million_tokens

        results.append({
            "model": model.name,
            "provider": model.provider,
            "dimensions": model.dimensions,
            "index_cost_usd": index_cost,
            "daily_query_cost_usd": daily_query_cost,
            "monthly_query_cost_usd": daily_query_cost * 30,
            "total_first_month_usd": index_cost + daily_query_cost * 30,
        })

    return sorted(results, key=lambda x: x["total_first_month_usd"])
```

I> **Note:** Embedding costs are typically 1-5% of total RAG pipeline cost. The LLM generation (with injected context) is 90%+ of the cost. Optimize the context injection first — it has 20x more impact than switching embedding models.

**Embedding model migration costs**

Switching embedding models is not just a pricing decision — it is a re-indexing event. When you change embedding models, every document in your vector store must be re-embedded because embeddings from different models are not compatible. A 100,000-document knowledge base at 2,000 tokens per document is 200 million tokens to re-embed. At the new model's embedding rate, this is a one-time cost, but it also requires engineering time to run the migration, validate that retrieval quality has not regressed, and update any similarity thresholds that were tuned for the previous model's embedding space.

The migration cost is often the reason teams stay on a suboptimal embedding model longer than they should. If you are evaluating a new embedding model that offers better retrieval quality or lower per-token pricing, calculate the total migration cost (re-indexing tokens plus engineering time plus quality validation) and amortize it over the expected lifetime of the new index. If the monthly savings from the new model's lower query rates or the quality improvement from better retrieval exceed the amortized migration cost within three to six months, the switch is justified. If not, wait until the next natural re-indexing event (a major knowledge base update, a schema change, a vector database migration) to bundle the embedding model switch with other work.

**When to skip RAG entirely: long-context trade-offs**

As context windows expand — models now routinely support 128,000 to 200,000 tokens — a question that deserves serious consideration is whether you need RAG at all. If your entire knowledge base fits within the model's context window, you can skip the embedding, retrieval, and reranking layers entirely and simply include the full knowledge base in every prompt. This eliminates retrieval errors (the model sees everything, so it cannot miss relevant information) and removes the RAG infrastructure stack entirely.

The trade-off is cost. Injecting 100,000 tokens of knowledge base content into every request is expensive at LLM input rates. At Claude Sonnet's pricing, 100,000 input tokens costs $0.30 per request. If you have 10,000 daily queries, that is $3,000 per day just for context injection — far more than a well-optimized RAG pipeline would cost. However, if your knowledge base is small (under 10,000 tokens) and your query volume is moderate, the "stuff everything in the prompt" approach may be both simpler and cheaper than building and maintaining a RAG pipeline.

The decision framework is straightforward. Calculate the per-request cost of injecting the full knowledge base as context tokens. Compare it to the per-request cost of a RAG pipeline (embedding query, vector retrieval, reranking, and injecting only the top-k chunks). If the full-context approach is cheaper — or only marginally more expensive — and your knowledge base fits within the context window with room for the query and response, skip RAG. The engineering complexity you avoid (no vector database, no embedding pipeline, no chunking strategy, no relevance tuning) has real value that does not appear in the per-token cost comparison. If the full-context approach is significantly more expensive, RAG pays for itself through context reduction. For most production

systems with knowledge bases above 20,000 tokens and moderate-to-high query volumes, RAG remains the more economical choice.

**Index storage and refresh economics**

Beyond embedding generation, the ongoing cost of maintaining a vector index deserves attention. Vector databases charge for storage (proportional to index size and embedding dimensionality), queries (per-search cost or throughput-based pricing), and in some cases, update operations. These costs are small relative to LLM generation but grow linearly with index size and can become significant for large knowledge bases that are updated frequently.

The refresh strategy for your index — how often you re-embed changed documents — has direct cost implications. A real-time refresh strategy that re-embeds documents on every edit maximizes freshness but generates continuous embedding costs. A nightly batch refresh collects all changes from the day and re-embeds them in a single batch, reducing embedding API calls and potentially qualifying for batch pricing. A weekly refresh reduces costs further but introduces up to a week of staleness. The right cadence depends on how time-sensitive your knowledge base content is. A customer FAQ that changes monthly can use weekly refreshes. A news aggregation knowledge base may need real-time updates.

---

## Common mistakes

**1. Using top-k=10 or higher by default.** Most RAG frameworks default to retrieving 10+ chunks. Each additional chunk adds hundreds of tokens to the input. Start with top-k=3 and increase only if retrieval quality demands it. The `optimize_chunking` function in tokenwatch lets you compare the cost of different top-k values side by side — use it to find the minimum top-k that maintains answer quality.

**2. Not measuring context utilization.** Are the retrieved chunks actually being used by the model? If you inject 5 chunks and the model's answer only references information from chunk 1, you're paying for 4 chunks of wasted context. The `context_waste_pct` field in `RAGCostBreakdown` estimates this using a heuristic (chunks beyond top-3 are assumed less useful). For more precise measurement, ask the model to cite which chunks it used and track utilization rates over time.

**3. Re-indexing entire document collections instead of incremental updates.** If your knowledge base has 100,000 documents and you change 500 per week, re-index only the changed documents. Re-embedding the entire collection weekly costs 200x more than incremental updates. Track document change timestamps and only re-embed modified documents.

**4. Ignoring chunk overlap in cost calculations.** With 50-token overlap between 500-token chunks, you're embedding 10% redundant content. At scale, this adds up. The `estimate_chunking_cost` function accounts for overlap when calculating total embedding tokens — use its output rather than naively multiplying document tokens by embedding rate.

**5. Not combining RAG with caching.** If your RAG system injects the same system prompt and tool definitions on every request, those stable tokens should be cached (Chapter 6). The

combination of RAG context optimization (reducing chunk tokens) and caching (reducing the cost of the stable prefix) can cut total RAG request cost by 60-70% compared to a naive implementation.

**6. Treating all queries equally.** Not every query needs the same retrieval depth. A factual lookup ("What is the return policy?") can be answered from a single chunk. An analytical query ("Compare our return policy across product categories") may need 5-8 chunks from different sections. Using the same top-k for both wastes tokens on simple queries and may under-retrieve for complex ones. The `DynamicTopK` strategy in the Hard exercise explores this adaptive approach — consider implementing it for high-volume RAG features where the query complexity distribution is bimodal.

---

## Exercises

**Easy:** Use `optimize_chunking` to compare four chunking strategies for a knowledge base of 10,000 documents (avg 2,000 tokens each) with 5,000 daily queries, using Claude Sonnet pricing. Which strategy gives the lowest monthly cost? Which gives the best precision/cost ratio? You should observe that the strategy with the fewest total context tokens per query (typically large chunks with low top-k) has the lowest cost, but may sacrifice retrieval precision. The strategy with small chunks and high top-k has the highest cost but the best precision. The sweet spot is usually in the middle — 500-token chunks with top-k=5 — balancing cost and quality.

**Medium:** Build a `RAGCostDashboard` that reads from `RAGCostTracker.history` and produces a report showing: average context tokens per query, average cost per RAG request vs. non-RAG request, context waste percentage, and the estimated savings from reducing top-k by 1.

**Hard:** Implement a `DynamicTopK` strategy that adjusts the number of retrieved chunks per query based on the similarity score distribution. If the top-3 chunks all have scores above 0.85, use only those 3. If the scores are more spread out (top chunk at 0.75, second at 0.72, third at 0.68), include more chunks. Compare the cost and quality of dynamic top-k vs. fixed top-k=5 on a test set.

---

## Summary

In this chapter you learned:

- **RAG has four cost layers** — embedding, storage, reranking, and generation — with LLM generation dominating at 90%+ of total cost
- **Chunking strategy directly affects cost** — chunk size multiplied by top-k determines context tokens per query; smaller chunks with lower top-k are often the cheapest without sacrificing quality
- **Relevance filtering eliminates waste** — thresholding by similarity score and using rerankers to select only the best chunks reduces context injection cost
- **Embedding model choice has minimal impact** on total RAG cost compared to context optimization
- **Measure context utilization** — track whether retrieved chunks are actually used by the model; unused chunks are pure cost waste

**Next chapter**
RAG is a common pattern, but the most expensive LLM workload pattern is agents. Chapter 10 tackles the cost multiplier effect of agent loops: how tool calls, recursive reasoning, and memory management create runaway costs, and how to build budget controls that keep agents economically viable.

---

A> **Code for this chapter:** `project/cap09/tokenwatch/`

# Chapter 10 — Agents: The Cost Multiplier

Agents are the most powerful pattern in LLM applications — and the most expensive. While a single-shot API call has predictable, bounded cost, an agent can call the model dozens of times in a single user interaction. Each iteration adds context from previous tool calls, growing the input token count with every loop. An agent that calls 8 tools in sequence might process 50,000 total tokens to accomplish what a human would describe as "one request."

This cost multiplier effect is not theoretical. In production systems I've worked on, agent-based features consumed 10-50x more tokens per user interaction than single-shot features. Without explicit budget controls, agent costs scale unpredictably and can spike dramatically when the model gets stuck in a reasoning loop or calls tools unnecessarily.

This chapter builds tokenwatch's agent cost control module: per-request budgets, loop limits, tool call metering, and strategies for making agents economically sustainable.

**Where this fits in your cost journey**
Agents represent the convergence of every cost challenge discussed in the previous chapters. Each iteration of the agent loop is an LLM call that benefits from prompt compression (Chapter 5), prompt caching (Chapter 6), and model routing (Chapter 8). The growing conversation history within the agent loop is the exact problem that conversation truncation (Chapter 5) addresses. The tool definitions sent on every iteration are candidates for the tool minimization techniques from the same chapter. And the agent's overall cost must be governed by the budgets and quotas from Chapter 12. In short, agents are the most expensive workload pattern, and they benefit from every optimization technique in this book. If you skip straight to this chapter without implementing the earlier techniques, your agent costs will be needlessly high even with budget controls. Budget controls prevent runaway spend, but they do not reduce the cost of well-behaved requests. For that, you need the optimization stack.

---

## Why agents are expensive

The agent cost multiplier comes from three sources:

### 1. The iterative loop

Every iteration of the agent loop requires a full API call. Each call includes the growing conversation history — the system prompt, all previous messages, all tool call results. By iteration 5, the input context might be 10x what it was at iteration 1.

```
Iteration 1: system(500) + user(200)                    = 700 input tokens
Iteration 2: system(500) + user(200) + assistant(300) + tool_result(800) = 1,800
input tokens
Iteration 3: system(500) + previous(1,800) + assistant(200) + tool_result(1,200)
= 3,700 input tokens
Iteration 5: ~8,000+ input tokens
Iteration 8: ~15,000+ input tokens
```

The total tokens processed across all iterations far exceeds the final response size.

**2. Tool call overhead**

Every tool definition is sent with every API call. If you have 15 tools defined, that's 1,500-3,000 tokens of tool schemas on every single iteration — even iterations where the model doesn't use most of them.

**3. Unnecessary iterations**

Models sometimes call tools unnecessarily, repeat tool calls with the same arguments, or enter reasoning loops where they keep thinking without making progress. Without explicit controls, these patterns can consume thousands of tokens before the agent produces a final response.

**Worked example: the cost of an 8-iteration agent**

Using Claude Sonnet ($3.00/$15.00 per M) with a 500-token system prompt and 10 tool definitions (~2,000 tokens), let's trace the cost of an 8-iteration agent interaction:

| Iteration | Input tokens | Output tokens | Cumulative input | Iter. cost |
|---|---|---|---|---|
| 1 | 2,700 | 200 | 2,700 | $0.011 |
| 2 | 3,900 | 150 + tool call | 6,600 | $0.014 |
| 3 | 5,500 | 200 + tool call | 12,100 | $0.020 |
| 4 | 7,200 | 300 + tool call | 19,300 | $0.026 |
| 5 | 9,100 | 250 + tool call | 28,400 | $0.031 |
| 6 | 11,000 | 200 + tool call | 39,400 | $0.037 |
| 7 | 13,200 | 350 | 52,600 | $0.045 |
| 8 | 15,500 | 500 (final) | 68,100 | $0.054 |
| **Total** | | | **68,100** | **$0.238** |

The same user request handled as a single-shot call (no tools, no loop) would cost approximately $0.011. The agent version costs 22x more. This is why agent cost control is not optional — it is a prerequisite for deploying agents in production.

**Real-world scenario: the runaway agent**

At a large fintech, a customer support agent was deployed without iteration limits. A customer asked "summarize all my transactions this year." The agent called a transaction-list tool 12 times (paginating through results), accumulated 45,000 tokens of tool results in context, and then attempted to summarize the entire history. The single interaction cost $1.40 — roughly 1,000x the average request cost. Worse, the agent hit the context window limit and produced a truncated, useless response. The `AgentBudget` class we build below would have caught this at iteration 8 or at the $0.50 cost threshold and returned a graceful partial response instead.

---

## Agent budget controls

The first line of defense is a per-request budget that limits how much an agent can spend on a single user interaction:

```
"""tokenwatch/agents.py — Agent cost controls and budget management."""

from dataclasses import dataclass, field
from typing import Any, Callable

from tokenwatch.config import TokenwatchConfig
from tokenwatch.tokens import estimate_tokens


class AgentBudgetExceeded(Exception):
    """Raised when an agent exceeds its cost budget."""
    def __init__(self, budget: float, spent: float, message: str = ""):
        self.budget = budget
        self.spent = spent
        super().__init__(
            message or f"Agent budget exceeded: spent ${spent:.4f} of
${budget:.4f} budget"
        )


@dataclass
class AgentBudget:
    """Budget constraints for a single agent execution."""
    max_cost_usd: float = 1.0       # maximum total cost
    max_iterations: int = 15         # maximum loop iterations
    max_tool_calls: int = 20         # maximum total tool calls
    max_input_tokens: int = 100_000  # maximum total input tokens
    max_thinking_tokens: int = 10_000  # maximum thinking tokens per iteration
    warn_at_pct: float = 0.7        # warn callback at 70% of budget

    # Tracking
    current_cost: float = 0.0
    current_iterations: int = 0
    current_tool_calls: int = 0
```

```
    current_input_tokens: int = 0
    warnings_issued: list[str] = field(default_factory=list)

    def check(self, additional_cost: float = 0.0) -> None:
        """Check if the budget allows continued execution.

        Raises AgentBudgetExceeded if any limit is reached.
        """
        projected = self.current_cost + additional_cost

        if projected > self.max_cost_usd:
            raise AgentBudgetExceeded(
                self.max_cost_usd, projected,
                f"Cost limit: ${projected:.4f} exceeds ${self.max_cost_usd:.4f}",
            )
        if self.current_iterations >= self.max_iterations:
            raise AgentBudgetExceeded(
                self.max_cost_usd, self.current_cost,
                f"Iteration limit: {self.current_iterations} >=
{self.max_iterations}",
            )
        if self.current_tool_calls >= self.max_tool_calls:
            raise AgentBudgetExceeded(
                self.max_cost_usd, self.current_cost,
                f"Tool call limit: {self.current_tool_calls} >=
{self.max_tool_calls}",
            )
        if self.current_input_tokens >= self.max_input_tokens:
            raise AgentBudgetExceeded(
                self.max_cost_usd, self.current_cost,
                f"Input token limit: {self.current_input_tokens:,} >=
{self.max_input_tokens:,}",
            )

    def record_iteration(
        self,
        cost: float,
        input_tokens: int,
        tool_calls: int = 0,
    ) -> None:
        """Record an iteration's resource usage."""
        self.current_cost += cost
        self.current_iterations += 1
        self.current_input_tokens += input_tokens
        self.current_tool_calls += tool_calls

        # Check for warning threshold
        if self.current_cost > self.max_cost_usd * self.warn_at_pct:
            warning = (
```

```
                f"Budget warning: ${self.current_cost:.4f} spent "
                f"({self.current_cost/self.max_cost_usd*100:.0f}% of limit)"
            )
            if warning not in self.warnings_issued:
                self.warnings_issued.append(warning)

    @property
    def remaining_budget_usd(self) -> float:
        return max(0, self.max_cost_usd - self.current_cost)

    @property
    def remaining_iterations(self) -> int:
        return max(0, self.max_iterations - self.current_iterations)

    def summary(self) -> dict:
        """Get a summary of budget consumption."""
        return {
            "cost_spent": self.current_cost,
            "cost_limit": self.max_cost_usd,
            "cost_pct": self.current_cost / self.max_cost_usd * 100,
            "iterations": self.current_iterations,
            "iteration_limit": self.max_iterations,
            "tool_calls": self.current_tool_calls,
            "input_tokens": self.current_input_tokens,
            "warnings": self.warnings_issued,
        }
```

---

## The budget-aware agent loop

Now let's integrate budget controls into an actual agent loop:

```
"""tokenwatch/agent_loop.py — Budget-aware agent execution loop."""

import time
from typing import Any

import anthropic

from tokenwatch.agents import AgentBudget, AgentBudgetExceeded
from tokenwatch.config import TokenwatchConfig
from tokenwatch.middleware import TokenMeter


def run_agent_with_budget(
    client: anthropic.Anthropic,
    config: TokenwatchConfig,
    meter: TokenMeter,
    messages: list[dict],
    system: str,
```

```
    tools: list[dict],
    tool_executor: dict[str, Any],
    model: str = "claude-sonnet-4-20250514",
    budget: AgentBudget | None = None,
    feature: str = "agent",
    user_id: str = "",
    max_tokens: int = 4096,
) -> tuple[str, AgentBudget]:
    """Run an agent loop with budget controls.

    The agent executes until the model stops calling tools,
    the budget is exceeded, or the iteration limit is reached.

    Args:
        tool_executor: Dict mapping tool names to callable functions
        budget: Budget constraints (defaults to reasonable limits)

    Returns:
        (final_response_text, budget_summary)
    """
    budget = budget or AgentBudget()
    conversation = list(messages)

    while True:
        # Pre-flight budget check
        budget.check()

        # Call the model
        start = time.monotonic()
        response = client.messages.create(
            model=model,
            max_tokens=max_tokens,
            system=system,
            messages=conversation,
            tools=tools,
        )
        latency_ms = (time.monotonic() - start) * 1000

        # Meter the request
        record = meter.measure_anthropic(
            response=response,
            latency_ms=latency_ms,
            feature=feature,
            user_id=user_id,
        )

        # Count tool calls in this response
        tool_use_blocks = [
            b for b in response.content if b.type == "tool_use"
```

```
]

# Record iteration in budget
budget.record_iteration(
    cost=record.cost_usd,
    input_tokens=record.input_tokens,
    tool_calls=len(tool_use_blocks),
)

# Check budget after recording
try:
    budget.check()
except AgentBudgetExceeded:
    # Budget exceeded — return what we have
    final_text = ""
    for block in response.content:
        if hasattr(block, "text"):
            final_text += block.text
    if not final_text:
        final_text = (
            "[Agent stopped: budget limit reached. "
            f"Spent ${budget.current_cost:.4f} across "
            f"{budget.current_iterations} iterations.]"
        )
    return final_text, budget

# If no tool calls, we're done
if response.stop_reason == "end_turn" or not tool_use_blocks:
    final_text = ""
    for block in response.content:
        if hasattr(block, "text"):
            final_text += block.text
    return final_text, budget

# Execute tool calls
conversation.append({"role": "assistant", "content": response.content})

tool_results = []
for tool_block in tool_use_blocks:
    tool_name = tool_block.name
    tool_input = tool_block.input

    if tool_name in tool_executor:
        try:
            result = tool_executor[tool_name](**tool_input)
            result_str = str(result)
        except Exception as e:
            result_str = f"Error: {e}"
    else:
```

```
            result_str = f"Error: Unknown tool '{tool_name}'"

        tool_results.append({
            "type": "tool_result",
            "tool_use_id": tool_block.id,
            "content": result_str,
        })

    conversation.append({"role": "user", "content": tool_results})
```

## Tool call optimization

Reducing the number of tool calls is one of the most effective ways to reduce agent costs. Every tool call has a direct cost (the tokens consumed by the tool call block, the tool result tokens added to context) and an indirect cost (the tool definitions re-sent on the next iteration). A single unnecessary tool call in an 8-iteration agent loop adds its result tokens to context for all subsequent iterations, compounding the cost. In production systems, I have seen agents make duplicate tool calls (calling the same API with the same parameters twice), unnecessary tool calls (looking up information that was already provided in the user message), and exploratory tool calls (calling a search tool with overly broad parameters, getting a large result, and then calling it again with refined parameters). Each of these patterns represents avoidable cost.

The most impactful optimization is often the simplest: improve the system prompt to tell the agent not to repeat tool calls, to check existing context before calling tools, and to use specific rather than broad search parameters. This prompt engineering for agents can reduce average tool calls per interaction by 20-40% with no infrastructure changes. Beyond prompt engineering, here are architectural strategies:

```
"""tokenwatch/tool_optimizer.py — Tool call cost optimization."""

from dataclasses import dataclass
from collections import defaultdict
from tokenwatch.store import UsageStore


@dataclass
class ToolCallAnalysis:
    """Analysis of tool calling patterns for cost optimization."""
    total_tool_calls: int
    unique_tool_calls: int
    duplicate_calls: int
    duplicate_pct: float
    most_called_tools: list[dict]
    estimated_waste_from_duplicates_usd: float


def analyze_tool_patterns(
    store: UsageStore,
```

```
    feature: str,
    since: str | None = None,
) -> dict:
    """Analyze tool calling patterns to find optimization opportunities.

    Looks for:
    - Duplicate tool calls (same tool, same arguments)
    - Excessive tool calls per interaction
    - Unused tools (defined but never called)
    """
    records = store.query(feature=feature, since=since)
    if not records:
        return {"error": "No records found"}

    # Aggregate statistics
    total_iterations = sum(1 for r in records if r.stop_reason == "tool_use")
    total_cost = sum(r.cost_usd for r in records)
    avg_iterations_per_session = total_iterations / len(records) if records else
0

    return {
        "total_requests": len(records),
        "total_cost_usd": total_cost,
        "avg_cost_per_request": total_cost / len(records) if records else 0,
        "total_tool_use_iterations": total_iterations,
        "avg_iterations_per_request": avg_iterations_per_session,
        "recommendations": _generate_recommendations(records),
    }


def _generate_recommendations(records) -> list[str]:
    """Generate cost optimization recommendations from patterns."""
    recs = []

    avg_cost = sum(r.cost_usd for r in records) / len(records) if records else 0
    high_cost_count = sum(1 for r in records if r.cost_usd > avg_cost * 3)

    if high_cost_count > len(records) * 0.1:
        recs.append(
            f"HIGH VARIANCE: {high_cost_count} requests ({high_cost_count/
len(records)*100:.0f}%) "
            f"cost >3x the average. Consider tighter budget limits."
        )

    # Check for requests with many iterations
    multi_tool = sum(1 for r in records if r.stop_reason == "tool_use")
    if multi_tool > len(records) * 0.5:
        recs.append(
            "HIGH TOOL USE: >50% of requests involve tool calls. "
```

```
            "Review whether all tools are necessary and reduce tool definitions."
        )

    return recs
```

**Dynamic tool loading**

Instead of sending all tool definitions on every iteration, only send tools that are relevant to the current state:

```
def select_tools_for_iteration(
    all_tools: list[dict],
    iteration: int,
    previous_tool_calls: list[str],
    task_type: str = "",
) -> list[dict]:
    """Select relevant tools for the current iteration.

    Strategies:
    - First iteration: only include discovery/reading tools
    - After data is gathered: include action/writing tools
    - Never include tools that have already been called with
      identical arguments (prevent duplicate calls)
    """
    # Basic strategy: categorize tools
    read_tools = [t for t in all_tools if any(
        kw in t["name"].lower()
        for kw in ["get", "read", "search", "list", "fetch"]
    )]
    write_tools = [t for t in all_tools if any(
        kw in t["name"].lower()
        for kw in ["create", "update", "delete", "send", "write"]
    )]
    other_tools = [
        t for t in all_tools
        if t not in read_tools and t not in write_tools
    ]

    if iteration == 0:
        # First iteration: read tools + other
        return read_tools + other_tools
    else:
        # Subsequent iterations: all tools
        return all_tools
```

---

## Memory cost management

Long-running agents accumulate memory (conversation history, tool results) that grows the context window. Managing this memory is critical for cost control:

```python
"""tokenwatch/agent_memory.py — Cost-aware agent memory management."""

from tokenwatch.tokens import estimate_tokens


def compress_agent_memory(
    messages: list[dict],
    max_tokens: int = 10_000,
    preserve_last_n_tool_results: int = 3,
) -> list[dict]:
    """Compress agent conversation history to fit within a token budget.

    Strategy:
    1. Always preserve the system prompt
    2. Always preserve the last N tool results (recent context)
    3. Summarize older tool results to reduce tokens
    4. Drop intermediate assistant messages that are just tool calls
    """
    if not messages:
        return messages

    system_msgs = [m for m in messages if m.get("role") == "system"]
    other_msgs = [m for m in messages if m.get("role") != "system"]

    # Count system tokens
    system_tokens = sum(
        estimate_tokens(str(m.get("content", "")))
        for m in system_msgs
    )

    remaining_budget = max_tokens - system_tokens
    if remaining_budget <= 0:
        return system_msgs  # system prompt alone exceeds budget

    # Keep last N messages always
    keep_last = min(preserve_last_n_tool_results * 2, len(other_msgs))
    recent = other_msgs[-keep_last:]
    older = other_msgs[:-keep_last] if keep_last < len(other_msgs) else []

    recent_tokens = sum(
        estimate_tokens(str(m.get("content", "")))
        for m in recent
    )

    if recent_tokens >= remaining_budget:
        # Even recent messages exceed budget
        return system_msgs + recent[-4:]  # keep just last 4

    # Add older messages if budget allows
```

```
    budget_for_older = remaining_budget - recent_tokens
    compressed_older = []

    for msg in reversed(older):
        msg_tokens = estimate_tokens(str(msg.get("content", "")))
        if budget_for_older >= msg_tokens:
            compressed_older.insert(0, msg)
            budget_for_older -= msg_tokens
        else:
            # Summarize remaining messages into a single message
            summary = {
                "role": "user",
                "content": (
                    "[Earlier in this conversation, the agent performed "
                    f"{len(older) - len(compressed_older)} additional steps "
                    "including tool calls and analysis. Key context has been
preserved above.]"
                ),
            }
            compressed_older.insert(0, summary)
            break

    return system_msgs + compressed_older + recent
```

W> **Warning:** Compressing agent memory can cause the model to lose context from earlier steps, potentially leading to repeated tool calls or contradictory actions. If your agent tasks require referencing early results, be conservative about compression — or use explicit "working memory" objects that persist key findings outside the conversation history.

---

## Budgeting agent features at the product level

Agent cost control is not just an engineering concern — it is a product planning concern. When a product manager proposes a new agent-based feature, the cost conversation needs to happen during the design phase, not after launch. The per-interaction cost of an agent feature is fundamentally different from a single-shot feature, and that difference must be reflected in the feature's business case.

Start by estimating the average interaction cost using the token growth model from earlier in this chapter: system prompt tokens, average number of iterations, average tool result size, and the model tier. Multiply by the expected daily interaction volume to get a projected daily cost. Compare this to the revenue or value the feature generates per interaction. If a customer support agent costs $0.25 per interaction and deflects a $5 support ticket, the unit economics are strong. If an internal analytics agent costs $1.50 per query and replaces a manual process that took an analyst 10 minutes, the value is clear. If the numbers do not work, the feature needs redesign before it needs engineering.

**Sandboxing cost experiments**

Before deploying agent features to production traffic, run sandboxed cost experiments on representative workloads. Create a test harness that replays real user queries through the agent with budget controls enabled, and measure the cost distribution. You care about the mean cost per interaction, but you care even more about the tail: what does the p95 cost look like? What about p99? Agent cost distributions are typically heavy-tailed — most interactions are cheap, but a small fraction are extremely expensive. If the p99 cost is 20x the mean, your budget limits need to account for that variance without being so tight that they interrupt normal interactions at p95.

**Explaining agent cost variance to stakeholders**

Agent features produce inherently variable costs, and this variability confuses stakeholders who are accustomed to the predictable per-request costs of single-shot LLM features. A monthly cost report that shows "agent feature: $4,200, up 40% from last month" triggers alarm, but the increase may be entirely explained by higher interaction volume, not by any regression in per-interaction efficiency. Decompose agent costs in your reporting: show the number of interactions, the average cost per interaction, and the average iterations per interaction as separate metrics. If the cost-per-interaction is stable and the total cost increase is driven by adoption growth, that is a success story, not a problem. If the cost-per-interaction is climbing, investigate whether agents are taking more iterations (possible prompt regression), using more expensive tools (a new data source returning larger payloads), or hitting budget limits more often (indicating the budget is too low for the feature's actual complexity).

**Kill-switch design rationale**

Every agent feature should ship with a kill-switch: a configuration flag that can instantly disable the agent loop and fall back to a non-agent behavior (a single-shot LLM call, a static response, or a human handoff). The kill-switch is not a sign of distrust in the agent architecture — it is a recognition that agents are the most unpredictable cost center in your LLM system. When a cost anomaly is detected (a sudden spike in per-interaction cost, an agent entering a loop that the budget controls are not catching fast enough, or a new failure mode that causes retries), the kill-switch lets you stop the bleeding within seconds while you diagnose the issue. Without a kill-switch, your only option during an agent cost incident is to disable the entire feature, which affects all users rather than just the agent behavior. A well-designed kill-switch degrades gracefully — users still get a response, just one that does not involve the agent loop.

---

## Common mistakes

**1. No iteration limit.** An agent without an iteration limit can loop indefinitely when the model gets stuck. Always set a hard iteration limit (10-15 is reasonable for most tasks). When the limit is reached, return the best available answer with an explanation. The `AgentBudget` class enforces this with `max_iterations` — set it at initialization and never override it at call time.

**2. No cost budget.** Even with an iteration limit, some iterations are expensive (large context, many tool results). A cost budget caps the total spend regardless of how many iterations it takes

to reach the limit. Set `max_cost_usd` to a value that represents the maximum acceptable cost for a single user interaction — typically 10-50x your average single-shot request cost.

**3. Sending all tools on every iteration.** If you have 20 tools, you're sending 4,000+ tokens of tool definitions on every iteration. Dynamic tool loading — sending only relevant tools for the current step — can reduce per-iteration input cost by 30-50%. The `select_tools_for_iteration` function demonstrates one approach: send read-only tools on iteration 1, then expand to write tools on subsequent iterations.

**4. Not compressing tool results.** Tool results can be verbose — a database query might return 2,000 tokens of JSON. If that result is re-sent in context on every subsequent iteration, it compounds quickly. The `compress_agent_memory` function handles this by summarizing older tool results while preserving recent ones.

**5. Not tracking agent cost separately from single-shot cost.** Agent interactions are fundamentally different cost events. If you mix them in the same cost-per-request metrics, agents inflate the average and make single-shot features look more expensive than they are. Tag agent requests with a distinct feature name (e.g., `"chat-agent"` vs. `"ticket-classifier"`) and analyze their economics separately.

---

## Exercises

**Easy:** Create an `AgentBudget` with a $0.50 limit and 10-iteration cap. Simulate an agent loop that adds $0.05 per iteration. At which iteration does the budget warn? At which does it halt? Note that the `warn_at_pct` default is 70%, so the warning should trigger at iteration 7 ($0.35, which is 70% of $0.50) and the budget should halt at iteration 10 ($0.50). In production, you would wire the warning callback to a logging system so that operations teams can observe when agents approach their limits, and the halt to a graceful degradation path that returns a partial response rather than an error.

**Medium:** Implement `select_tools_for_iteration` with a more sophisticated strategy: track which tools have been called in previous iterations and exclude them from subsequent iterations (since the results are already in context). Measure the token savings from reducing tool definitions by 2-3 tools per iteration over a 6-iteration agent run.

**Hard:** Build an `AgentCostAnalyzer` that reads agent usage records from the store and identifies optimization opportunities: agents that consistently hit budget limits (need restructuring), agents with high duplicate tool call rates (need better prompting), and agents where 80% of cost comes from the last 20% of iterations (would benefit from early stopping heuristics).

---

## Summary

In this chapter you learned:

- **Agents are the most expensive LLM pattern** — iterative loops with growing context create a cost multiplier that can make a single user interaction cost 10-50x more than a single-shot call

- **Three sources of agent cost** — the iterative loop (growing context), tool call overhead (tool definitions on every iteration), and unnecessary iterations (loops, duplicates)
- **Budget controls are essential** — per-request cost limits, iteration caps, and tool call limits prevent runaway spend
- **Dynamic tool loading** reduces per-iteration input cost by only sending relevant tool definitions
- **Memory compression** prevents context growth from making late iterations prohibitively expensive

**Next chapter**

So far we've focused on optimizing managed API costs — paying providers per token. Chapter 11 asks a different question: when does it make economic sense to run models yourself? Self-hosting (with quantized open-weight models on your own GPUs) has a completely different cost model. We'll build a break-even calculator that compares managed API costs against self-hosting costs for your specific workload.

---

A> **Code for this chapter:** `project/cap10/tokenwatch/`

# Chapter 11 — Self-Hosting Break-Even: Quantization, GPUs, and Serverless Inference

Every team running LLM workloads at scale eventually asks: "Should we self-host?" The answer is never a simple yes or no. It depends on your workload volume, latency requirements, model capability needs, and engineering capacity. Self-hosting trades variable per-token costs for fixed infrastructure costs — and that trade-off only makes sense above a certain volume threshold.

This chapter builds tokenwatch's self-hosting break-even calculator. We'll model the costs of running open-weight models on GPU infrastructure, compare them against managed API costs for your actual workload, and help you make an informed build-vs-buy decision. The analysis covers GPU provisioning, quantization trade-offs, serverless inference platforms, and the hidden costs that make self-hosting more expensive than the hardware alone.

**Where this fits in your cost journey**

Self-hosting is not an optimization technique in the same way as prompt compression or caching. It is an architectural decision that changes your entire cost model from variable (pay-per-token) to fixed (pay-per-GPU-hour). This decision should only be evaluated after you have implemented the per-token optimizations from Chapters 5-10, for two reasons. First, optimizing your token consumption reduces your managed API bill, which raises the break-even volume threshold for self-hosting — you may find that after optimization, managed APIs are cheaper than you thought. Second, the techniques you learn in those chapters (prompt compression, caching, routing) apply equally to self-hosted models, so the per-request cost of your self-hosted inference also benefits. The break-even calculator in this chapter takes your optimized cost profile as input, which means the analysis is only meaningful once you know your true token consumption after optimization.

Self-hosting also interacts with model routing (Chapter 8) in a particularly powerful way: you can route simple requests to a self-hosted open-weight model (which has zero marginal cost per token once the GPU is provisioned) and route complex requests to a managed API (where you pay per token but get frontier-model quality). This hybrid architecture captures the best of both cost models.

---

## The self-hosting cost model

Managed APIs have a simple cost model: you pay per token. Self-hosting has a complex cost model with multiple components:

**Infrastructure costs:** - GPU instance costs (hourly rates for cloud GPUs) - Storage for model weights - Network transfer for inference requests - Load balancer / API gateway

**Operational costs:** - Engineering time for deployment, monitoring, upgrades - On-call rotations for inference infrastructure - Model update cycles (deploying new versions)

**Capability costs:** - Quality gap: open-weight models may not match Claude/GPT-4o quality - Missing features: no native prompt caching, limited tool calling support - Latency variability: self-managed scaling vs. provider-managed

### Team skills and opportunity cost in the build-vs-buy equation

The break-even calculator models engineering time as a dollar cost per hour, but this framing understates the true cost. Engineering hours are not fungible. The 40 hours per month that a team spends maintaining inference infrastructure are 40 hours not spent building product features, reducing technical debt, or improving system reliability. If your team has five engineers and inference infrastructure consumes one engineer's full capacity, you have effectively reduced your feature development throughput by 20%. For a startup where time-to-market determines survival, this opportunity cost can dwarf the infrastructure savings.

The skills required for self-hosting are also specialized. Deploying and operating GPU inference infrastructure requires familiarity with CUDA drivers, inference frameworks (vLLM, TGI, or similar), model quantization tooling, GPU memory management, and distributed serving. If your team does not already have these skills, the ramp-up period is measured in weeks to months, during which productivity on other projects suffers. Hiring for these skills is competitive — ML infrastructure engineers are among the most in-demand roles in the industry. Factor the recruiting timeline and the salary premium for GPU infrastructure expertise into your break-even calculation. If the marginal cost of the hire exceeds the infrastructure savings, managed APIs remain the better choice even if the per-token arithmetic favors self-hosting.

### When self-hosting makes sense (and when it doesn't)

The decision matrix is straightforward:

| Factor | Favors self-hosting | Favors managed API |
|---|---|---|
| Volume | >500K requests/day | <50K requests/day |
| Task complexity | Simple (classification, extraction) | Complex (reasoning, tool use) |
| Latency requirement | Flexible or very tight (on-premise GPU) | Standard (200-2000ms) |
| Engineering capacity | Dedicated ML infra team | Small team, no GPU expertise |
| Data sensitivity | Cannot send data to third parties | Standard cloud SaaS acceptable |
| Model capability | Open-weight models suffice | Frontier model quality needed |

The gray zone is 50K-500K requests/day, where the break-even depends heavily on model choice, quantization, and operational costs. This is exactly where the `break_even_analysis` function adds the most value — it turns intuition into numbers.

```
"""tokenwatch/selfhost.py — Self-hosting break-even calculator."""

from dataclasses import dataclass, field
from typing import Any


@dataclass
class GPUInstance:
    """A GPU instance option for self-hosting."""
    name: str
    provider: str
    gpu_model: str
    gpu_count: int
    vram_gb: int
    hourly_cost: float
    tokens_per_second: int  # estimated throughput for the target model
    max_batch_size: int = 32
    spot_discount_pct: float = 60.0  # typical spot discount


# Common GPU instances (prices are approximate — verify with providers)
GPU_CATALOG = [
    GPUInstance(
        name="A100-80GB-1x",
        provider="aws/gcp/azure",
        gpu_model="A100-80GB",
        gpu_count=1,
        vram_gb=80,
        hourly_cost=3.50,
```

```
        tokens_per_second=80,
    ),
    GPUInstance(
        name="A100-80GB-4x",
        provider="aws/gcp/azure",
        gpu_model="A100-80GB",
        gpu_count=4,
        vram_gb=320,
        hourly_cost=14.00,
        tokens_per_second=300,
    ),
    GPUInstance(
        name="H100-80GB-1x",
        provider="aws/gcp/azure",
        gpu_model="H100-80GB",
        gpu_count=1,
        vram_gb=80,
        hourly_cost=5.00,
        tokens_per_second=150,
    ),
    GPUInstance(
        name="H100-80GB-8x",
        provider="aws/gcp/azure",
        gpu_model="H100-80GB",
        gpu_count=8,
        vram_gb=640,
        hourly_cost=40.00,
        tokens_per_second=1000,
    ),
    GPUInstance(
        name="L4-24GB-1x",
        provider="gcp",
        gpu_model="L4",
        gpu_count=1,
        vram_gb=24,
        hourly_cost=0.80,
        tokens_per_second=30,
        max_batch_size=8,
    ),
]


@dataclass
class SelfHostModel:
    """An open-weight model for self-hosting analysis."""
    name: str
    parameter_count: str  # e.g., "7B", "70B"
    vram_required_fp16_gb: float
    vram_required_int8_gb: float
```

```
    vram_required_int4_gb: float
    quality_score: float  # 0-1 relative to frontier models
    context_window: int


OPEN_MODELS = [
    SelfHostModel(
        name="Llama-3.1-8B-Instruct",
        parameter_count="8B",
        vram_required_fp16_gb=16,
        vram_required_int8_gb=8,
        vram_required_int4_gb=5,
        quality_score=0.55,
        context_window=128_000,
    ),
    SelfHostModel(
        name="Llama-3.1-70B-Instruct",
        parameter_count="70B",
        vram_required_fp16_gb=140,
        vram_required_int8_gb=70,
        vram_required_int4_gb=40,
        quality_score=0.78,
        context_window=128_000,
    ),
    SelfHostModel(
        name="Mistral-Large-2",
        parameter_count="123B",
        vram_required_fp16_gb=246,
        vram_required_int8_gb=123,
        vram_required_int4_gb=65,
        quality_score=0.80,
        context_window=128_000,
    ),
    SelfHostModel(
        name="Qwen-2.5-72B-Instruct",
        parameter_count="72B",
        vram_required_fp16_gb=144,
        vram_required_int8_gb=72,
        vram_required_int4_gb=40,
        quality_score=0.76,
        context_window=131_072,
    ),
]


@dataclass
class BreakEvenAnalysis:
    """Break-even analysis comparing managed API vs. self-hosting."""
    # Managed API costs
```

```
    managed_monthly_cost: float
    managed_cost_per_request: float

    # Self-hosting costs
    gpu_instance: str
    gpu_monthly_cost: float
    operational_monthly_cost: float
    selfhost_monthly_total: float
    selfhost_cost_per_request: float

    # Analysis
    savings_monthly: float
    savings_pct: float
    break_even_daily_requests: int
    recommendation: str
    quality_warning: str


def break_even_analysis(
    daily_requests: int,
    avg_input_tokens: int,
    avg_output_tokens: int,
    managed_model: str,
    managed_input_rate: float,
    managed_output_rate: float,
    selfhost_model: SelfHostModel,
    gpu_instance: GPUInstance,
    engineering_hours_monthly: float = 40.0,
    engineering_hourly_rate: float = 100.0,
    use_spot: bool = False,
    quantization: str = "int8",  # "fp16", "int8", "int4"
    redundancy_factor: float = 1.5,  # extra capacity for reliability
) -> BreakEvenAnalysis:
    """Compare managed API costs against self-hosting.

    Args:
        engineering_hours_monthly: Hours of engineering time for maintenance
        engineering_hourly_rate: Cost per hour of engineering time
        use_spot: Whether to use spot/preemptible instances
        quantization: Quantization level for the model
        redundancy_factor: Extra instances for reliability (1.5 = 50% extra)
    """
    # --- Managed API costs ---
    monthly_requests = daily_requests * 30
    managed_input_cost = (avg_input_tokens * monthly_requests / 1_000_000) *
managed_input_rate
    managed_output_cost = (avg_output_tokens * monthly_requests / 1_000_000) *
managed_output_rate
    managed_monthly = managed_input_cost + managed_output_cost
```

```
    managed_per_request = managed_monthly / monthly_requests if monthly_requests
> 0 else 0

    # --- Self-hosting costs ---
    # Check VRAM requirement
    vram_map = {
        "fp16": selfhost_model.vram_required_fp16_gb,
        "int8": selfhost_model.vram_required_int8_gb,
        "int4": selfhost_model.vram_required_int4_gb,
    }
    required_vram = vram_map.get(quantization,
selfhost_model.vram_required_int8_gb)

    # How many instances needed?
    instances_for_vram = max(1, int(required_vram / gpu_instance.vram_gb) + 1)

    # How many instances needed for throughput?
    total_tokens_per_second = daily_requests * (avg_input_tokens +
avg_output_tokens) / 86400
    instances_for_throughput = max(
        1, int(total_tokens_per_second / gpu_instance.tokens_per_second) + 1
    )

    instances_needed = max(instances_for_vram, instances_for_throughput)
    instances_with_redundancy = int(instances_needed * redundancy_factor)

    # GPU cost
    hourly_cost = gpu_instance.hourly_cost
    if use_spot:
        hourly_cost *= (1 - gpu_instance.spot_discount_pct / 100)

    gpu_monthly = instances_with_redundancy * hourly_cost * 24 * 30

    # Operational cost
    ops_monthly = engineering_hours_monthly * engineering_hourly_rate

    selfhost_total = gpu_monthly + ops_monthly
    selfhost_per_request = selfhost_total / monthly_requests if monthly_requests
> 0 else 0

    # --- Analysis ---
    savings = managed_monthly - selfhost_total
    savings_pct = (savings / managed_monthly * 100) if managed_monthly > 0 else 0

    # Break-even volume (where self-hosting becomes cheaper)
    if selfhost_total > 0 and managed_per_request > 0:
        # At what volume does selfhost_total = managed_per_request * volume * 30?
        break_even_daily = int(selfhost_total / (managed_per_request * 30))
    else:
```

```
        break_even_daily = 0

    # Quality warning
    quality_gap = 1.0 - selfhost_model.quality_score
    if quality_gap > 0.3:
        quality_warning = (
            f"SIGNIFICANT quality gap: {selfhost_model.name} scores "
            f"{selfhost_model.quality_score:.0%} relative to frontier models. "
            "Only suitable for simple tasks."
        )
    elif quality_gap > 0.15:
        quality_warning = (
            f"MODERATE quality gap: {selfhost_model.name} scores "
            f"{selfhost_model.quality_score:.0%}. Suitable for most tasks "
            "but may struggle with complex reasoning."
        )
    else:
        quality_warning = (
            f"SMALL quality gap: {selfhost_model.name} scores "
            f"{selfhost_model.quality_score:.0%}. Suitable for most production
tasks."
        )

    # Recommendation
    if savings_pct > 30 and quality_gap < 0.25:
        recommendation = (
            f"RECOMMENDED: Self-hosting saves ${savings:,.0f}/month "
            f"({savings_pct:.0f}%) with acceptable quality."
        )
    elif savings_pct > 30:
        recommendation = (
            f"CONDITIONAL: Savings of ${savings:,.0f}/month but quality gap "
            f"may impact user experience. Test thoroughly."
        )
    elif savings_pct > 0:
        recommendation = (
            f"MARGINAL: Savings of ${savings:,.0f}/month ({savings_pct:.0f}%) "
            f"may not justify the operational complexity."
        )
    else:
        recommendation = (
            f"NOT RECOMMENDED: Self-hosting costs ${-savings:,.0f}/month MORE "
            f"than managed API at current volume."
        )

    return BreakEvenAnalysis(
        managed_monthly_cost=managed_monthly,
        managed_cost_per_request=managed_per_request,
        gpu_instance=gpu_instance.name,
```

```
        gpu_monthly_cost=gpu_monthly,
        operational_monthly_cost=ops_monthly,
        selfhost_monthly_total=selfhost_total,
        selfhost_cost_per_request=selfhost_per_request,
        savings_monthly=savings,
        savings_pct=savings_pct,
        break_even_daily_requests=break_even_daily,
        recommendation=recommendation,
        quality_warning=quality_warning,
    )
```

---

## Quantization: trading quality for cost

Quantization reduces model precision from 16-bit floating point (FP16) to 8-bit integers (INT8) or 4-bit integers (INT4). This reduces VRAM requirements proportionally, letting you run larger models on smaller GPUs or fit more concurrent requests on the same hardware.

The trade-off is quality. Each step down in precision introduces small numerical errors that accumulate through the model's layers. The quality impact varies by model and task:

- **FP16 (full precision):** Baseline quality. Requires the most VRAM.
- **INT8 (8-bit quantization):** Typically 0-2% quality degradation. Halves VRAM requirement. This is the sweet spot for most production deployments.
- **INT4 (4-bit quantization):** 3-8% quality degradation. Quarters VRAM requirement. Viable for simple tasks (classification, extraction) but noticeable on complex reasoning.

```
"""tokenwatch/quantization.py — Quantization cost-quality trade-off analysis."""

from dataclasses import dataclass


@dataclass
class QuantizationOption:
    """A quantization level with its trade-offs."""
    name: str
    vram_multiplier: float  # relative to FP16
    quality_retention: float  # 1.0 = no degradation
    throughput_multiplier: float  # relative to FP16


QUANTIZATION_OPTIONS = {
    "fp16": QuantizationOption("FP16 (full precision)", 1.0, 1.0, 1.0),
    "int8": QuantizationOption("INT8 (8-bit)", 0.5, 0.98, 1.3),
    "int4": QuantizationOption("INT4 (4-bit)", 0.25, 0.93, 1.6),
    "gptq-4bit": QuantizationOption("GPTQ 4-bit", 0.25, 0.94, 1.5),
    "awq-4bit": QuantizationOption("AWQ 4-bit", 0.25, 0.95, 1.5),
}
```

```python
def compare_quantization_options(
    model_vram_fp16_gb: float,
    gpu_vram_gb: float,
    gpu_hourly_cost: float,
    daily_requests: int,
) -> list[dict]:
    """Compare quantization options for a model-GPU combination.

    Shows VRAM requirement, whether it fits, and cost implications.
    """
    results = []
    for key, quant in QUANTIZATION_OPTIONS.items():
        required_vram = model_vram_fp16_gb * quant.vram_multiplier
        fits = required_vram <= gpu_vram_gb
        gpus_needed = max(1, int(required_vram / gpu_vram_gb) + (1 if
required_vram % gpu_vram_gb > 0 else 0))

        monthly_cost = gpus_needed * gpu_hourly_cost * 24 * 30
        cost_per_request = monthly_cost / (daily_requests * 30) if daily_requests
> 0 else 0

        results.append({
            "quantization": quant.name,
            "vram_required_gb": required_vram,
            "fits_single_gpu": fits,
            "gpus_needed": gpus_needed,
            "quality_retention": f"{quant.quality_retention:.0%}",
            "throughput_multiplier": f"{quant.throughput_multiplier:.1f}x",
            "monthly_cost_usd": monthly_cost,
            "cost_per_request": cost_per_request,
        })

    return sorted(results, key=lambda x: x["monthly_cost_usd"])
```

---

## Serverless inference platforms

Between fully managed APIs and fully self-hosted, there are serverless inference platforms that offer a middle ground: you deploy a model, they manage the infrastructure, and you pay per request or per second of compute. Examples include AWS Bedrock, Together AI, Fireworks AI, and Replicate.

These platforms typically offer lower per-token prices than Anthropic/OpenAI for open-weight models, with less operational overhead than full self-hosting. The trade-off is less control and potential cold-start latency.

Serverless inference is often the right starting point for teams exploring the self-hosting spectrum. It lets you validate that an open-weight model meets your quality requirements without committing to GPU infrastructure management. If the quality is acceptable and the volume justifies

dedicated GPUs, you can migrate to full self-hosting later. If the quality falls short, you have lost nothing — no GPUs to decommission, no inference code to maintain. Think of serverless inference as the staging environment for your self-hosting decision.

The economic profile of serverless inference is also worth understanding in detail. Unlike managed APIs from frontier providers, serverless platforms price open-weight models at their actual computational cost plus a margin for the platform's infrastructure management. This typically means 50-70% lower per-token prices than frontier API models, with the caveat that the model quality may be 10-30% lower on complex tasks. For workloads where an open-weight model's quality is sufficient (classification, extraction, simple summarization), serverless inference delivers most of the cost benefit of self-hosting with none of the operational burden.

```
"""tokenwatch/serverless.py — Serverless inference cost comparison."""

from dataclasses import dataclass


@dataclass
class ServerlessOption:
    """A serverless inference platform option."""
    platform: str
    model: str
    input_rate_per_million: float
    output_rate_per_million: float
    cold_start_seconds: float
    max_context_tokens: int


def compare_serverless_options(
    daily_requests: int,
    avg_input_tokens: int,
    avg_output_tokens: int,
    options: list[ServerlessOption],
) -> list[dict]:
    """Compare serverless inference platforms for a workload."""
    monthly_requests = daily_requests * 30
    results = []

    for opt in options:
        monthly_input_cost = (
            avg_input_tokens * monthly_requests / 1_000_000
        ) * opt.input_rate_per_million
        monthly_output_cost = (
            avg_output_tokens * monthly_requests / 1_000_000
        ) * opt.output_rate_per_million
        monthly_total = monthly_input_cost + monthly_output_cost

        results.append({
            "platform": opt.platform,
```

```
            "model": opt.model,
            "monthly_cost_usd": monthly_total,
            "cost_per_request": monthly_total / monthly_requests if
monthly_requests > 0 else 0,
            "cold_start_seconds": opt.cold_start_seconds,
        })

    return sorted(results, key=lambda x: x["monthly_cost_usd"])
```

T> **Tip:** Serverless inference platforms are often the right answer for workloads where you want lower prices than managed APIs but don't have the engineering capacity for full self-hosting. They're particularly attractive for batch workloads where cold-start latency doesn't matter.

---

## The hidden costs of self-hosting

GPU costs are the visible part of self-hosting. The hidden costs often double or triple the effective price:

1. **Engineering time.** Deploying, monitoring, and maintaining inference infrastructure takes real engineering hours. At $100-200/hour fully loaded, 40 hours/month of maintenance adds $4,000-8,000 to your monthly cost.
2. **Reliability overhead.** You need redundant instances for high availability. A 1.5x redundancy factor increases GPU costs by 50%.
3. **Scaling complexity.** Managed APIs auto-scale seamlessly. Self-hosted inference requires you to manage scaling policies, handle traffic spikes, and provision capacity for peak load (not average load).
4. **Model updates.** When a new model version releases, you need to download, test, quantize, and deploy it. This is a multi-day engineering effort per update.
5. **Opportunity cost.** Engineering hours spent on inference infrastructure are hours not spent on product features.

### Worked example: full break-even calculation

Using `pricing.yaml` rates for Claude Sonnet ($3.00/$15.00 per M) as the managed baseline, and Llama-3.1-70B on a 4x A100-80GB instance ($14.00/hour) as the self-hosted option:

```
Managed API (200,000 requests/day, 1,000 input + 200 output tokens):
  Input cost:  200K * 1,000 / 1M * $3.00   = $600/day
  Output cost: 200K * 200 / 1M * $15.00    = $600/day
  Total: $1,200/day → $36,000/month

Self-hosted (Llama-3.1-70B, INT8 on 4xA100):
  GPU cost:  2 instances (1 primary + redundancy) * $14.00/hr * 24 * 30 =
$20,160/month
  Eng. time: 40 hrs/month * $100/hr                                    = $4,000/
month
```

```
  Total: $24,160/month

Savings: $11,840/month (33%)
Break-even: ~120,000 requests/day
```

At 200,000 requests/day, self-hosting saves 33%. But the quality score of Llama-3.1-70B (0.78 relative to frontier) means you should only consider this for workloads where that quality gap is acceptable — classification, extraction, simple summarization. For complex reasoning or nuanced tool calling, the quality difference will generate user complaints and support escalations that cost more than the infrastructure savings.

**What changes at each scale tier**

The self-hosting decision looks fundamentally different depending on your organization's scale, and the factors that matter shift as you grow.

At startup scale (under 50,000 requests per day), self-hosting almost never makes sense. Your total managed API bill is likely under $5,000 per month. A single GPU instance costs $2,500 to $3,600 per month before operational overhead. The break-even arithmetic does not work, and the engineering time spent on infrastructure directly competes with product development. Use managed APIs, implement the optimization techniques from Chapters 5-10, and revisit the self-hosting question when your volume grows by an order of magnitude.

At scaleup scale (50,000 to 500,000 requests per day), the economics become interesting but the decision is nuanced. Your managed API bill is $5,000 to $50,000 per month, and self-hosting can reduce it by 30-50%. However, this is also the scale where engineering capacity is most constrained — every engineer pulled into infrastructure work is an engineer not shipping features. The hybrid approach (self-host simple tasks on open-weight models, use managed APIs for complex tasks) often provides the best balance at this scale: you capture the cost savings on high-volume, low-complexity workloads without committing your entire team to infrastructure operations.

At enterprise scale (over 500,000 requests per day), self-hosting is almost certainly part of the answer. Your managed API bill exceeds $50,000 per month, and the savings from self-hosting justify a dedicated ML infrastructure team. At this scale, you can also negotiate volume discounts with cloud GPU providers, invest in reserved instances for further cost reduction, and build internal tooling that amortizes the operational overhead across multiple self-hosted models. The capacity planning challenge is also more tractable at enterprise scale because your traffic patterns are more predictable — you have months of historical data to forecast demand, and the law of large numbers smooths out individual request variability.

---

## Common mistakes

**1. Comparing GPU cost to API cost without including operational overhead.** The GPU alone is always cheaper at scale. But the total cost of self-hosting includes engineering, redundancy, and management. Include all costs in the break-even analysis.

**2. Assuming model quality is equivalent.** Open-weight models have improved dramatically, but frontier models (Claude Opus, GPT-4o) still outperform them on complex reasoning, instruction following, and tool use. Test on your actual tasks before committing.

**3. Provisioning for average load instead of peak load.** If your traffic peaks at 3x the average, you need GPU capacity for the peak. Under-provisioning leads to request queueing and latency spikes. The `redundancy_factor` parameter in `break_even_analysis` accounts for this — set it to your peak/average ratio (typically 1.5-3.0).

**4. Ignoring cold-start latency in serverless.** Serverless inference platforms can have 10-30 second cold starts for large models. This is fine for batch workloads but unacceptable for real-time user interactions. The `ServerlessOption.cold_start_seconds` field in tokenwatch surfaces this trade-off in the comparison output.

**5. Forgetting about model updates.** Open-weight models release new versions frequently. Each update requires downloading weights (50-200 GB), quantizing, testing, and deploying. If you update monthly, budget 2-3 days of engineering time per update cycle. This operational burden is invisible in initial break-even calculations but very real over a year.

**6. Ignoring the inference framework learning curve.** Running a model on a GPU requires an inference framework (vLLM, TGI, llama.cpp, or similar). Each framework has its own configuration, performance characteristics, and failure modes. Your team needs to learn the framework, tune its parameters (batch size, KV cache management, tensor parallelism), and debug its edge cases. This learning investment is a hidden cost that does not appear in any financial calculation but can consume weeks of engineering time in the first deployment. Factor this into your decision timeline — if you need cost savings within a month, self-hosting is too slow; if you are planning for the next quarter, it may be viable.

---

## Exercises

**Easy:** Run `break_even_analysis` for your current workload (or estimate one) comparing Claude Sonnet against Llama-3.1-70B on an H100 instance. At what daily request volume does self-hosting become cheaper?

**Medium:** Create a multi-tier analysis: use Llama-3.1-8B (self-hosted on an L4 GPU) for simple tasks and Claude Sonnet (managed API) for complex tasks. Compare the blended cost of this hybrid approach against using Claude Sonnet for everything.

**Hard:** Build a `CapacityPlanner` that takes a traffic pattern (hourly request counts for a typical day) and calculates the optimal number of GPU instances for each hour. Compare the cost of (a) fixed provisioning at peak capacity, (b) auto-scaling with 5-minute spin-up time, and (c) serverless inference. Include spot instance pricing for the fixed-capacity option.

---

## Summary

In this chapter you learned:

- **Self-hosting trades variable costs for fixed costs** — it's only cheaper above a volume threshold that depends on your specific workload and infrastructure choices
- **Quantization enables running larger models on smaller GPUs** — INT8 is the sweet spot for most production workloads, with minimal quality degradation
- **Hidden costs** (engineering time, redundancy, scaling, updates) often double the effective cost of self-hosting beyond GPU prices alone
- **Serverless inference platforms** offer a middle ground between managed APIs and full self-hosting
- **The break-even calculator** helps make data-driven build-vs-buy decisions by modeling all cost components

**Next chapter**

Whether you use managed APIs, self-hosted models, or a hybrid, you need governance. Chapter 12 covers budgets, quotas, showback/chargeback, and the organizational practices that keep AI costs under control as usage scales across teams.

---

A> **Code for this chapter:** `project/cap11/tokenwatch/`

# Chapter 12 — Governance: Budgets, Quotas, and Cost Accountability

Optimization techniques — caching, routing, batching — reduce the cost of individual requests. Governance ensures that the overall system stays within budget as usage scales, new features are added, and new teams start using LLMs. Without governance, every optimization gain gets eaten by uncontrolled growth.

This chapter is partly about code and partly about organizational practices. We'll build tokenwatch's governance module — budgets, quotas, circuit breakers, and showback reports — but we'll also cover the human side: how to implement cost accountability structures that work across engineering teams. If you're a tech lead or engineering manager accountable for AI spend, this chapter is written specifically for you.

**Where this fits in your cost journey**

Governance is the capstone of the optimization stack. Chapters 5-10 reduce the cost of individual requests. Governance ensures that the total cost across all requests, all features, and all teams stays within organizational limits. Without governance, every optimization gain is eventually consumed by growth: more features launch, more users adopt them, more teams experiment with LLMs, and the aggregate bill climbs even as the per-request cost declines. Governance is the mechanism that converts per-request efficiency into organizational cost control. It also creates the accountability structures that motivate teams to apply the optimization techniques from earlier chapters — when a team sees their cost dashboard approaching a budget limit, they have a concrete incentive to implement caching, routing, or batch processing rather than requesting a budget increase.

The governance tools in this chapter depend directly on the metering foundation from Chapter 3 (for real-time spend tracking), the unit economics from Chapter 4 (for cost attribution to features and teams), and the model routing from Chapter 8 (for the DOWNGRADE action that switches to a cheaper model when a budget is exceeded). If you have not implemented those chapters, the governance module will not have the data or the levers it needs to function effectively.

---

## The governance framework

AI cost governance has four pillars:

**1. Visibility** — Everyone who builds AI features can see what those features cost. This is the metering and dashboards from Chapters 3-4.

**2. Accountability** — Every LLM cost is attributed to a team, feature, and budget. Nobody gets to say "I didn't know it was expensive."

**3. Controls** — Budgets, quotas, and circuit breakers prevent runaway spend. The system enforces limits automatically.

**4. Optimization** — Teams have incentives and tools to reduce costs. Savings are visible and celebrated.

Let's build the controls layer.

**Real-world scenario: governance saves a launch**

At a large fintech, a product team launched a new AI-powered document analysis feature without governance controls. The feature used Claude Opus for complex document reasoning — appropriate for the task, but expensive ($15.00/$75.00 per M input/output tokens from `pricing.yaml`). On launch day, a marketing email drove 50,000 users to try the feature. Each user uploaded 2-3 documents, generating 150,000 API calls. The daily cost was $18,000 — the team's entire monthly AI budget consumed in one day.

Had the team deployed with tokenwatch governance, a daily budget of $2,000 (DOWNGRADE action) would have automatically switched to Claude Haiku after the budget was reached, reducing cost per request by ~95% while still providing a functional (if less capable) experience for users 20,001 through 150,000. The $18,000 day would have been a $2,800 day, and the team would have had time to implement proper caching, routing, and usage throttling before the next spike.

This is why governance is not a "nice to have" — it is the difference between a manageable cost overrun and a career-limiting incident.

---

## Budget enforcement

Budget enforcement operates at multiple levels: per-request, per-feature, per-team, and global. Each level serves a different purpose.

```
"""tokenwatch/governance.py — Budget enforcement and quota management."""
```

```
from dataclasses import dataclass, field
from datetime import datetime, timezone, timedelta
from typing import Any, Callable
from enum import Enum

from tokenwatch.store import UsageStore
from tokenwatch.config import TokenwatchConfig


class BudgetAction(Enum):
    """Action to take when a budget is exceeded."""
    WARN = "warn"          # Log warning, allow the request
    THROTTLE = "throttle"  # Slow down requests (add delay)
    DOWNGRADE = "downgrade"  # Route to a cheaper model
    BLOCK = "block"        # Reject the request


@dataclass
class Budget:
    """A cost budget with enforcement rules."""
    name: str
    limit_usd: float
    period: str             # "hourly", "daily", "weekly", "monthly"
    scope_type: str         # "feature", "team", "global"
    scope_value: str        # the feature name, team name, or "*"
    action: BudgetAction = BudgetAction.WARN
    warn_at_pct: float = 80.0
    current_spend: float = 0.0

    @property
    def remaining(self) -> float:
        return max(0, self.limit_usd - self.current_spend)

    @property
    def utilization_pct(self) -> float:
        return (self.current_spend / self.limit_usd * 100) if self.limit_usd > 0
else 0

    @property
    def is_exceeded(self) -> bool:
        return self.current_spend >= self.limit_usd

    @property
    def is_warning(self) -> bool:
        return self.utilization_pct >= self.warn_at_pct and not self.is_exceeded


@dataclass
class BudgetCheckResult:
```

```python
    """Result of a budget check before a request."""
    allowed: bool
    action: BudgetAction
    budget_name: str
    remaining_usd: float
    utilization_pct: float
    message: str
    downgrade_model: str | None = None  # model to use if downgraded


class BudgetManager:
    """Manages budgets and enforces spending limits."""

    def __init__(
        self,
        store: UsageStore,
        budgets: list[Budget] | None = None,
        downgrade_model: str = "claude-haiku-4-5-20250514",
    ):
        self.store = store
        self.budgets = budgets or []
        self.downgrade_model = downgrade_model
        self._alert_callbacks: list[Callable] = []

    def add_budget(self, budget: Budget) -> None:
        """Add a budget to the manager."""
        self.budgets.append(budget)

    def on_alert(self, callback: Callable[[Budget, str], None]) -> None:
        """Register a callback for budget alerts."""
        self._alert_callbacks.append(callback)

    def check_budget(
        self,
        feature: str,
        team: str,
        estimated_cost: float,
    ) -> BudgetCheckResult:
        """Check all applicable budgets before a request.

        Returns the most restrictive result across all matching budgets.
        """
        # Refresh budget spend from store
        self._refresh_budgets()

        applicable = self._get_applicable_budgets(feature, team)
        if not applicable:
            return BudgetCheckResult(
                allowed=True,
```

```
                action=BudgetAction.WARN,
                budget_name="none",
                remaining_usd=float("inf"),
                utilization_pct=0,
                message="No budgets configured",
            )

        # Check each applicable budget, return the most restrictive
        most_restrictive = None
        for budget in applicable:
            result = self._check_single_budget(budget, estimated_cost)
            if most_restrictive is None or not result.allowed:
                most_restrictive = result
            elif not most_restrictive.allowed:
                continue  # already blocking
            elif result.utilization_pct > most_restrictive.utilization_pct:
                most_restrictive = result

        return most_restrictive

    def _check_single_budget(
        self,
        budget: Budget,
        estimated_cost: float,
    ) -> BudgetCheckResult:
        """Check a single budget."""
        projected = budget.current_spend + estimated_cost

        if projected > budget.limit_usd:
            # Budget would be exceeded
            if budget.action == BudgetAction.BLOCK:
                self._fire_alert(budget, "BLOCKED: budget exceeded")
                return BudgetCheckResult(
                    allowed=False,
                    action=BudgetAction.BLOCK,
                    budget_name=budget.name,
                    remaining_usd=budget.remaining,
                    utilization_pct=budget.utilization_pct,
                    message=f"Budget '{budget.name}' exceeded:
${budget.current_spend:.2f} / ${budget.limit_usd:.2f}",
                )
            elif budget.action == BudgetAction.DOWNGRADE:
                self._fire_alert(budget, "DOWNGRADED: budget exceeded")
                return BudgetCheckResult(
                    allowed=True,
                    action=BudgetAction.DOWNGRADE,
                    budget_name=budget.name,
                    remaining_usd=budget.remaining,
                    utilization_pct=budget.utilization_pct,
```

```
                    message=f"Budget '{budget.name}' exceeded — downgrading to
cheaper model",
                    downgrade_model=self.downgrade_model,
                )
            else:
                self._fire_alert(budget, "WARNING: budget exceeded")
                return BudgetCheckResult(
                    allowed=True,
                    action=BudgetAction.WARN,
                    budget_name=budget.name,
                    remaining_usd=budget.remaining,
                    utilization_pct=budget.utilization_pct,
                    message=f"Budget '{budget.name}' exceeded (warn mode):
${budget.current_spend:.2f} / ${budget.limit_usd:.2f}",
                )

        if budget.is_warning:
            self._fire_alert(budget, f"APPROACHING: {budget.utilization_pct:.0f}%
used")

        return BudgetCheckResult(
            allowed=True,
            action=BudgetAction.WARN,
            budget_name=budget.name,
            remaining_usd=budget.remaining,
            utilization_pct=budget.utilization_pct,
            message="OK",
        )

    def _get_applicable_budgets(self, feature: str, team: str) -> list[Budget]:
        """Find all budgets that apply to this request."""
        applicable = []
        for budget in self.budgets:
            if budget.scope_type == "global":
                applicable.append(budget)
            elif budget.scope_type == "feature" and budget.scope_value ==
feature:
                applicable.append(budget)
            elif budget.scope_type == "team" and budget.scope_value == team:
                applicable.append(budget)
        return applicable

    def _refresh_budgets(self) -> None:
        """Refresh current spend for all budgets from the store."""
        now = datetime.now(timezone.utc)
        for budget in self.budgets:
            since = self._period_start(budget.period, now)
            if budget.scope_type == "global":
                records = self.store.query(since=since.isoformat())
```

```
            elif budget.scope_type == "feature":
                records = self.store.query(
                    feature=budget.scope_value, since=since.isoformat()
                )
            elif budget.scope_type == "team":
                records = self.store.query(
                    team=budget.scope_value, since=since.isoformat()
                )
            else:
                records = []

            budget.current_spend = sum(r.cost_usd for r in records)

    def _period_start(self, period: str, now: datetime) -> datetime:
        """Calculate the start of the current budget period."""
        if period == "hourly":
            return now.replace(minute=0, second=0, microsecond=0)
        elif period == "daily":
            return now.replace(hour=0, minute=0, second=0, microsecond=0)
        elif period == "weekly":
            start = now - timedelta(days=now.weekday())
            return start.replace(hour=0, minute=0, second=0, microsecond=0)
        elif period == "monthly":
            return now.replace(day=1, hour=0, minute=0, second=0, microsecond=0)
        return now

    def _fire_alert(self, budget: Budget, message: str) -> None:
        """Fire alert callbacks."""
        for callback in self._alert_callbacks:
            try:
                callback(budget, message)
            except Exception:
                pass  # Don't let alert failures break the request path
```

---

## Quota management

While budgets limit cost, quotas limit volume — the number of requests a team or feature can make per period. This prevents a single feature from consuming all the shared API capacity.

```
"""tokenwatch/quotas.py — Request quota management."""

from dataclasses import dataclass
from datetime import datetime, timezone
from collections import defaultdict


@dataclass
class Quota:
    """A request volume quota."""
```

```
    name: str
    max_requests: int
    period: str  # "hourly", "daily"
    scope_type: str
    scope_value: str
    current_count: int = 0


class QuotaManager:
    """Manages request quotas independently of cost budgets."""

    def __init__(self):
        self.quotas: list[Quota] = []
        self._counters: dict[str, int] = defaultdict(int)
        self._last_reset: dict[str, datetime] = {}

    def add_quota(self, quota: Quota) -> None:
        self.quotas.append(quota)

    def check_and_increment(
        self, feature: str, team: str
    ) -> tuple[bool, str]:
        """Check quota and increment counter.

        Returns (allowed, message).
        """
        self._maybe_reset_counters()

        for quota in self.quotas:
            key = f"{quota.scope_type}:{quota.scope_value}:{quota.period}"

            if quota.scope_type == "feature" and quota.scope_value == feature:
                if self._counters[key] >= quota.max_requests:
                    return False, f"Quota '{quota.name}' exceeded:
{self._counters[key]} >= {quota.max_requests}"
                self._counters[key] += 1
            elif quota.scope_type == "team" and quota.scope_value == team:
                if self._counters[key] >= quota.max_requests:
                    return False, f"Quota '{quota.name}' exceeded:
{self._counters[key]} >= {quota.max_requests}"
                self._counters[key] += 1

        return True, "OK"

    def _maybe_reset_counters(self) -> None:
        """Reset counters at period boundaries."""
        now = datetime.now(timezone.utc)
        for quota in self.quotas:
            key = f"{quota.scope_type}:{quota.scope_value}:{quota.period}"
```

```
        last = self._last_reset.get(key)

        should_reset = False
        if last is None:
            should_reset = True
        elif quota.period == "hourly" and now.hour != last.hour:
            should_reset = True
        elif quota.period == "daily" and now.date() != last.date():
            should_reset = True

        if should_reset:
            self._counters[key] = 0
            self._last_reset[key] = now
```

---

## Showback and chargeback reports

Showback makes costs visible to teams without actually charging them. Chargeback goes further — costs are allocated to team budgets. Both require clear attribution (which we built in Chapter 3) and regular reporting.

```
"""tokenwatch/showback.py — Showback and chargeback reporting."""

from dataclasses import dataclass
from collections import defaultdict
from tokenwatch.store import UsageStore


@dataclass
class TeamCostReport:
    """Cost report for a single team."""
    team: str
    total_cost_usd: float
    request_count: int
    top_features: list[dict]
    month_over_month_change_pct: float
    budget_utilization_pct: float
    recommendations: list[str]


def generate_showback_report(
    store: UsageStore,
    budgets: dict[str, float] | None = None,  # team -> monthly budget
    since: str | None = None,
    previous_period_since: str | None = None,
) -> list[TeamCostReport]:
    """Generate a showback report for all teams.

    This report is designed to be shared with team leads
    and engineering managers for cost accountability.
```

```
    """
    records = store.query(since=since)
    if not records:
        return []

    # Current period by team
    by_team: dict[str, list] = defaultdict(list)
    for r in records:
        by_team[r.team].append(r)

    # Previous period for comparison
    prev_costs: dict[str, float] = {}
    if previous_period_since:
        prev_records = store.query(since=previous_period_since)
        prev_in_range = [r for r in prev_records if since and r.timestamp <
since]
        for r in prev_in_range:
            prev_costs[r.team] = prev_costs.get(r.team, 0) + r.cost_usd

    reports = []
    budgets = budgets or {}

    for team, team_records in by_team.items():
        total_cost = sum(r.cost_usd for r in team_records)
        request_count = len(team_records)

        # Top features
        feature_costs: dict[str, float] = defaultdict(float)
        for r in team_records:
            feature_costs[r.feature] += r.cost_usd

        top_features = sorted(
            [{"feature": f, "cost": c, "pct": c/total_cost*100}
             for f, c in feature_costs.items()],
            key=lambda x: x["cost"],
            reverse=True,
        )[:5]

        # MoM change
        prev = prev_costs.get(team, 0)
        mom_change = ((total_cost - prev) / prev * 100) if prev > 0 else 0

        # Budget utilization
        team_budget = budgets.get(team, 0)
        budget_util = (total_cost / team_budget * 100) if team_budget > 0 else 0

        # Recommendations
        recs = []
        if budget_util > 90:
```

```
            recs.append(f"Budget critically high ({budget_util:.0f}%) — review
spend immediately")
        if mom_change > 50:
            recs.append(f"Cost increased {mom_change:.0f}% vs. last period —
investigate")

        cache_eligible = sum(1 for r in team_records if not r.is_cache_hit)
        if cache_eligible > request_count * 0.7:
            recs.append("Low cache utilization — consider enabling prompt
caching")

        reports.append(TeamCostReport(
            team=team,
            total_cost_usd=total_cost,
            request_count=request_count,
            top_features=top_features,
            month_over_month_change_pct=mom_change,
            budget_utilization_pct=budget_util,
            recommendations=recs,
        ))

    return sorted(reports, key=lambda x: x.total_cost_usd, reverse=True)
```

T> **Tip:** Start with showback (visibility without financial consequences) before implementing chargeback. Teams need time to understand their costs and optimize before being held financially accountable. A common progression: Month 1: visibility only; Month 2-3: showback with optimization guidance; Month 4+: chargeback against team budgets.

---

## Organizational patterns that work

Technology alone doesn't solve governance. Here are organizational patterns from production experience:

**AI Cost Review.** A monthly meeting where each team's AI spend is reviewed against budgets and targets. The purpose is not blame — it's learning. Share optimization wins, flag regressions, and allocate improvement effort.

**Cost Champion.** Designate one engineer per team as the "AI cost champion" responsible for monitoring their team's spend and driving optimization. This works better than making cost everyone's responsibility (which means it's nobody's responsibility).

**Optimization Sprints.** Dedicate a quarterly sprint to cost optimization across all teams. Teams compete on who achieves the biggest percentage reduction. The competition element drives engagement, and the dedicated time ensures optimization doesn't always lose to feature work.

**Guardrails by Default.** Every new AI feature launches with metering, budgets, and quotas pre-configured. The platform team provides templates and defaults. Teams can adjust limits, but they can't ship without them.

**Incident Postmortems for Cost Events.** When a cost incident occurs (a feature exceeds its budget, a regression doubles the bill, an agent loop runs away), treat it with the same rigor as a production outage. Run a blameless postmortem, identify the root cause, implement prevention measures, and share the learnings across teams. Cost incidents are production incidents — they consume organizational resources and, if recurring, erode trust in AI features. The `BudgetManager` alert callbacks provide the trigger for these postmortems, and the metering data from Chapter 3 provides the forensic evidence to reconstruct what happened.

**Rolling Out Governance Without Becoming the "No" Team.** The biggest risk in implementing governance is that the governance team (or the platform team that owns the governance tooling) becomes perceived as a blocker. If every cost conversation starts with "you can't do that," teams will route around governance rather than through it. Frame governance as a service, not a checkpoint. The governance team's job is to help feature teams understand their costs, identify optimization opportunities, and set budgets that are achievable — not to reject proposals or block launches. Lead with visibility: show teams what they are spending before telling them what they should spend. Lead with tools: give teams the `BudgetManager`, the `QuotaManager`, and the showback reports before imposing limits. When limits are necessary, explain the reasoning (the total organizational budget, the cost of the workload relative to its business value, the risk of uncontrolled growth) and offer alternatives (can the feature use a cheaper model? can part of the workload be batched?). Governance that enables is adopted voluntarily; governance that restricts is circumvented systematically.

**Cost Review Rituals.** The cadence and format of cost reviews matter as much as their existence. A weekly tactical review (15 minutes, focused on anomalies and budget utilization) catches problems early. A monthly strategic review (30 to 60 minutes, focused on trends, optimization opportunities, and budget adjustments) keeps the governance framework aligned with business priorities. The weekly review should be lightweight: pull the `generate_showback_report` output, highlight any team above 80% budget utilization or any feature with a month-over-month increase above 30%, and assign action items. The monthly review should be deeper: review the optimization backlog, celebrate cost reductions (teams that achieved significant savings should present their approach), adjust budgets for the next month based on planned launches and historical trends, and update routing policies if model capabilities have changed. The worst pattern is a quarterly review that tries to cover everything — by the time you notice a cost regression, three months of overspend have already accumulated.

**Progressive Rollout of Controls.** Do not launch all governance controls simultaneously. Start with visibility (showback reports, no enforcement), then add warnings (WARN action, no blocking), then soft controls (DOWNGRADE action, graceful degradation), and finally hard controls (BLOCK action, request rejection). Each stage gives teams time to adjust their behavior and optimize their workloads before the next level of control takes effect. Rushing to hard controls without this progression creates resentment and workarounds — both of which undermine the governance program.

### The governance maturity model

Organizations typically progress through four stages of AI cost governance:

| Stage | Characteristics | tokenwatch modules used |
|---|---|---|
| 1. Blind | No metering, monthly bill surprises | None (pre-tokenwatch) |
| 2. Visible | Per-feature cost tracking, basic dashboards | meter, store, economics, dashboard |
| 3. Controlled | Budgets, quotas, automated alerts | governance, quotas + Stage 2 |
| 4. Optimized | Active routing, caching, batch, cost targets | All modules, provider negotiation |

Most teams reading this book are at Stage 1 or 2. The goal is to reach Stage 3 within the first month of deployment and Stage 4 within a quarter. Stages 1 to 2 is the highest-impact transition — it changes every conversation from "how much did we spend?" to "which feature is driving spend and what should we do about it?"

The transition between stages is not just a matter of deploying more tokenwatch modules. Each stage requires a corresponding organizational shift. Moving from Stage 1 (Blind) to Stage 2 (Visible) requires convincing teams to instrument their LLM calls with metering — a technical change, but also a cultural one, because it makes costs transparent in a way that some teams find uncomfortable. Moving from Stage 2 to Stage 3 (Controlled) requires executive buy-in for budget limits, which means building a business case for governance (the cost incident scenarios from earlier in this chapter are persuasive here). Moving from Stage 3 to Stage 4 (Optimized) requires sustained engineering investment in optimization techniques — caching, routing, batching — and a culture where cost efficiency is valued alongside feature velocity. Each transition takes four to eight weeks when actively managed, or months when left to happen organically. Do not try to skip stages; the organizational learning that happens at each stage builds the foundation for the next.

---

## Common mistakes

**1. Setting budgets too tight initially.** If budgets are so tight that teams constantly hit limits, they'll find workarounds (unmetered API keys, shadow features) that destroy your visibility. Start with generous budgets and tighten gradually. A good heuristic: set initial budgets at 2x the current spend, then reduce by 20% per quarter as teams optimize.

**2. Governance without visibility.** If teams can't see their costs in near-real-time, budget enforcement feels arbitrary and punitive. Invest in dashboards before enforcement. The `generate_showback_report` function produces team-level cost reports with trend analysis — ship these weekly before activating any BLOCK or DOWNGRADE actions.

**3. One-size-fits-all quotas.** A team that runs a customer-facing chat agent has fundamentally different cost patterns than a team that runs batch analytics. Quotas should be per-feature, not

per-team. The `BudgetManager` supports both `scope_type="feature"` and `scope_type="team"` — use feature-level budgets for granular control and team-level budgets as an outer safety net.

**4. No exception process.** Sometimes a team legitimately needs to exceed their budget (product launch, special analysis, incident response). Have a documented exception process that allows temporary budget increases with approval. In tokenwatch, this is as simple as calling `budget_manager.add_budget()` with a temporary higher limit and removing it after the exception period.

**5. Treating governance as a one-time setup.** Budget limits need regular adjustment as workloads change, features launch, and provider pricing evolves. Schedule monthly budget reviews as part of the AI Cost Review meeting pattern described above. Stale budgets are almost as dangerous as no budgets — they either block legitimate work or fail to catch real overruns.

---

## Exercises

**Easy:** Configure three budgets: a daily budget for the "chat-agent" feature ($50/day, WARN), a daily budget for the "data-enrichment" team ($100/day, DOWNGRADE), and a monthly global budget ($5,000, BLOCK). Simulate requests and verify that each budget triggers its expected action. Notice the layered enforcement: a single request might pass the feature budget check but fail the global budget check. The `BudgetManager.check_budget` method evaluates all applicable budgets and returns the most restrictive result — understanding this layered behavior is essential for configuring budgets that protect without being overly restrictive at any single level.

**Medium:** Build a `GovernanceMiddleware` class that sits between the metered client and the API. It checks budgets and quotas before every request, enforces the most restrictive result (blocking if any budget blocks, downgrading if any budget downgrades), and logs governance decisions. Integrate it with the `MeteredAnthropic` client.

**Hard:** Create an automated showback email generator that produces a weekly HTML report for each team lead, showing their team's AI spend, top features, cost trends, and optimization recommendations. Include a comparison against other teams (anonymized) to create constructive peer pressure.

---

## Summary

In this chapter you learned:

- **Governance has four pillars** — visibility, accountability, controls, and optimization — all four are needed for sustainable AI cost management
- **Budget enforcement** operates at multiple levels (per-request, per-feature, per-team, global) with configurable actions (warn, throttle, downgrade, block)
- **Quotas** limit request volume independently of cost, preventing single features from consuming shared capacity
- **Showback reports** make costs visible to teams, enabling accountability before chargeback

- **Organizational patterns** — cost reviews, cost champions, optimization sprints — are as important as the technology

**Next chapter**
Governance keeps current costs under control. Chapter 13 tackles the forward-looking problem: how to negotiate with AI providers, forecast future spend, and optimize your provider mix. This is where engineering meets procurement.

---

A> **Code for this chapter:** `project/cap12/tokenwatch/`

# Chapter 13 — Negotiating and Forecasting: Provider Mix, Committed Use, and Spend Projection

You've optimized tokens, cached prefixes, routed to cheaper models, batched latency-tolerant work, and built governance. The remaining cost levers are strategic: how you structure your relationship with AI providers, how you forecast future spend, and how you decide which provider mix serves your workload best.

This chapter covers the business side of AI cost management. We'll build tokenwatch's forecasting module and explore the commercial levers — committed-use agreements, volume discounts, multi-provider strategies — that reduce costs at the contract level rather than the code level. This chapter will be especially relevant to tech leads and engineering managers who participate in vendor negotiations.

**Where this fits in your cost journey**
Forecasting and negotiation operate at a different level of abstraction than the optimization techniques in Chapters 5-10. Those chapters reduce the cost of individual tokens and requests. This chapter reduces the price of the tokens you cannot eliminate — the baseline consumption that remains after all optimizations are applied. The two approaches are complementary: optimization reduces volume, negotiation reduces rate. A team that has optimized their token consumption by 50% (through compression, caching, routing, and batching) and then negotiated a 20% committed-use discount has reduced their total cost by 60%. Neither technique alone achieves that result.

Forecasting also depends heavily on the metering foundation from Chapter 3 and the unit economics from Chapter 4. Without consistent, per-feature cost data collected over multiple months, your forecasts will be little more than guesses. With that data, the `SpendForecaster` module can extract growth rates, identify seasonal patterns, and project spend with confidence bands narrow enough to support commitment decisions. The investment you made in metering pays its most visible dividend here — at the negotiation table, where accurate data is the difference between a confident commitment and an anxious guess.

---

## Forecasting AI spend

Accurate forecasting is the foundation of both budgeting and negotiation. If you can predict next quarter's spend with reasonable accuracy, you can negotiate committed-use discounts confidently and allocate budgets proactively.

The challenge is that AI workloads are inherently harder to forecast than traditional infrastructure. Token consumption depends on user behavior, feature adoption, prompt changes, and model updates — all of which introduce variance.

## Usage growth modeling

Understanding what drives growth in your LLM spend is essential for accurate forecasting. AI cost growth has three distinct components, and conflating them leads to inaccurate projections.

**Organic growth** is the increase in usage from existing features as more users adopt them. If your ticket classifier handles 100,000 requests today and your user base grows 5% monthly, organic growth adds roughly 5,000 requests per month. This component is relatively predictable and can be modeled with your product's user growth rate.

**Feature-driven growth** comes from launching new AI features. Each new feature adds a step function to your spend curve. A new chat agent launching in month 3 does not gradually ramp — it appears at some initial volume and then grows organically from there. The `planned_features` parameter in `SpendForecaster.forecast_monthly` models these step functions explicitly.

**Behavioral growth** is the trickiest component. As users become more comfortable with AI features, they use them more intensively. Conversations get longer. Queries become more complex. Users who started with 2 interactions per day move to 10. This growth multiplier is invisible in user-count metrics but very visible in token metrics. To capture it, track tokens-per-user-per-day as a separate metric and include its trend in your growth rate assumption.

A robust forecast separates these three components rather than lumping them into a single growth rate. The single-rate approach works for stable, mature features but severely underestimates spend for features in their adoption growth phase, where behavioral growth can double the token-per-user rate within a quarter.

## Seasonality and traffic patterns

Many AI workloads exhibit seasonal patterns that naive forecasting ignores. Customer support volumes spike during product launches, holiday periods, and billing cycles. Internal AI tools see higher usage during business hours and lower usage on weekends and holidays. Data enrichment pipelines may run heavier during quarterly reporting periods.

Seasonality affects both your forecast accuracy and your caching effectiveness (Chapter 6). A feature with strongly diurnal traffic will see cache TTL expiration overnight, meaning your effective cache hit rate is lower than the peak-hour rate. A feature with weekly seasonality may have a different optimal batch schedule (Chapter 7) during high-traffic weeks than low-traffic weeks.

To account for seasonality, the `SpendForecaster` examines monthly cost variation through the coefficient of variation. However, for features with known seasonal patterns, you should adjust

the growth assumption manually. If you know that December volume is 2x the annual average for your customer support agent, do not let the December spike inflate your annual growth rate — model it as a seasonal multiplier instead.

## Scenario planning: best, expected, and worst case

The confidence bands in `ForecastResult` provide optimistic and pessimistic scenarios, but for budget planning and commitment decisions, you need to think through the scenarios more deliberately.

**Best case (optimistic):** All planned optimizations succeed, no new features launch ahead of schedule, user growth follows the lower bound of projections, and provider pricing drops (as it historically has). This scenario produces the floor for your commitment level — you should not commit below this amount, because you will almost certainly spend at least this much.

**Expected case (base forecast):** Current growth trends continue, planned features launch on schedule at estimated volumes, and no major prompt regressions or pricing changes occur. This is the center of your forecast and the anchor for budget requests.

**Worst case (pessimistic):** A viral feature drives unexpected adoption, a prompt regression goes undetected for two weeks (Chapter 3 metering would catch it faster, but assume the worst), a new model release changes your routing economics, and a provider raises prices. This scenario produces the ceiling that your governance controls (Chapter 12) must be configured to handle.

The `band_multiplier` parameter in the forecaster controls the width between these scenarios. For features with high historical variance (CV > 0.30), the bands should be wider. For stable, mature features (CV < 0.15), narrower bands are appropriate. The key discipline is reviewing which scenario materialized each month and adjusting your band width accordingly — if reality consistently falls outside your pessimistic band, your bands are too narrow.

### Worked example: quarterly forecast with planned features

Consider a system currently spending $25,000/month with a historical growth rate of 8% per month. The team plans to launch two new features next quarter:

- **Feature A** (document summarizer): launches month 1, estimated $3,000/month
- **Feature B** (agent-based support): launches month 2, estimated $8,000/month

```
Month 1 base: $25,000 * 1.08 = $27,000 + $3,000 (Feature A) = $30,000
Month 2 base: $27,000 * 1.08 = $29,160 + $3,000 + $8,000     = $40,160
Month 3 base: $29,160 * 1.08 = $31,493 + $3,000 + $8,000     = $42,493

Confidence bands (medium confidence, CV=0.20):
  Month 1: $25,500 – $34,500
  Month 2: $32,700 – $47,600
  Month 3: $33,700 – $51,300
```

Without the planned features, the forecast would be $27K-$31.5K. The new features add $11K-$14K per month by quarter's end. This is the kind of projection you need for committed-use

negotiations: commit at 80% of the base forecast ($34,000/month), which gives you room for the pessimistic scenario while still qualifying for volume discounts.

The `SpendForecaster.forecast_monthly` method accepts a `planned_features` parameter that handles exactly this calculation, including widening confidence bands as the forecast horizon extends.

```
"""tokenwatch/forecast.py — AI spend forecasting."""

from dataclasses import dataclass, field
from datetime import datetime, timezone, timedelta
from collections import defaultdict
import math

from tokenwatch.store import UsageStore


@dataclass
class ForecastResult:
    """Spend forecast for a future period."""
    period_label: str
    base_forecast_usd: float      # based on current trends
    optimistic_usd: float          # best case (lower bound)
    pessimistic_usd: float         # worst case (upper bound)
    growth_rate_monthly_pct: float
    confidence_level: str          # "high", "medium", "low"
    assumptions: list[str]
    risk_factors: list[str]


class SpendForecaster:
    """Forecasts future AI spend based on historical patterns."""

    def __init__(self, store: UsageStore):
        self.store = store

    def forecast_monthly(
        self,
        months_ahead: int = 3,
        growth_assumption_pct: float | None = None,
        planned_features: list[dict] | None = None,
    ) -> list[ForecastResult]:
        """Forecast monthly spend for the next N months.

        Args:
            months_ahead: Number of months to forecast
            growth_assumption_pct: Override growth rate (uses historical if None)
            planned_features: List of {"name": str, "estimated_monthly_cost":
float,
                                       "launch_month": int} for upcoming features
```

```
    """
    # Calculate historical monthly costs
    monthly_costs = self._get_monthly_costs()
    if len(monthly_costs) < 2:
        return [ForecastResult(
            period_label="Insufficient data",
            base_forecast_usd=0, optimistic_usd=0, pessimistic_usd=0,
            growth_rate_monthly_pct=0, confidence_level="low",
            assumptions=["Need at least 2 months of data"],
            risk_factors=[],
        )]

    # Calculate historical growth rate
    if growth_assumption_pct is not None:
        growth_rate = growth_assumption_pct / 100
    else:
        growth_rate = self._calculate_growth_rate(monthly_costs)

    # Calculate variance for confidence bands
    costs = list(monthly_costs.values())
    avg = sum(costs) / len(costs)
    variance = sum((c - avg) ** 2 for c in costs) / len(costs)
    std_dev = math.sqrt(variance)
    cv = std_dev / avg if avg > 0 else 0  # coefficient of variation

    # Confidence level based on variance
    if cv < 0.15:
        confidence = "high"
        band_multiplier = 0.15
    elif cv < 0.30:
        confidence = "medium"
        band_multiplier = 0.25
    else:
        confidence = "low"
        band_multiplier = 0.40

    last_month_cost = costs[-1]
    results = []

    for month in range(1, months_ahead + 1):
        base = last_month_cost * ((1 + growth_rate) ** month)

        # Add planned features
        feature_cost = 0.0
        if planned_features:
            for feature in planned_features:
                if feature.get("launch_month", 0) <= month:
                    feature_cost += feature.get("estimated_monthly_cost", 0)
```

```
            base += feature_cost

            # Confidence bands widen with time
            band = base * band_multiplier * math.sqrt(month)

            results.append(ForecastResult(
                period_label=f"Month +{month}",
                base_forecast_usd=base,
                optimistic_usd=base - band,
                pessimistic_usd=base + band,
                growth_rate_monthly_pct=growth_rate * 100,
                confidence_level=confidence,
                assumptions=[
                    f"Historical growth rate: {growth_rate*100:.1f}%/month",
                    f"Coefficient of variation: {cv:.2f}",
                    f"Planned new features: {len(planned_features or [])}",
                ],
                risk_factors=self._identify_risks(monthly_costs, growth_rate),
            ))

        return results

    def _get_monthly_costs(self) -> dict[str, float]:
        """Get monthly cost totals from the usage store."""
        records = self.store.query()
        monthly: dict[str, float] = defaultdict(float)

        for r in records:
            month_key = r.timestamp[:7]  # "2026-06"
            monthly[month_key] += r.cost_usd

        return dict(sorted(monthly.items()))

    def _calculate_growth_rate(self, monthly_costs: dict[str, float]) -> float:
        """Calculate the average monthly growth rate."""
        costs = list(monthly_costs.values())
        if len(costs) < 2:
            return 0.0

        growth_rates = []
        for i in range(1, len(costs)):
            if costs[i-1] > 0:
                rate = (costs[i] - costs[i-1]) / costs[i-1]
                growth_rates.append(rate)

        return sum(growth_rates) / len(growth_rates) if growth_rates else 0.0

    def _identify_risks(
        self,
```

```
        monthly_costs: dict[str, float],
        growth_rate: float,
    ) -> list[str]:
        """Identify risk factors that could invalidate the forecast."""
        risks = []

        if growth_rate > 0.3:
            risks.append(
                f"High growth rate ({growth_rate*100:.0f}%/month) — "
                "may indicate uncontrolled adoption"
            )

        costs = list(monthly_costs.values())
        if len(costs) >= 3:
            recent_growth = (costs[-1] - costs[-2]) / costs[-2] if costs[-2] > 0
else 0
            if recent_growth > growth_rate * 2:
                risks.append(
                    "Recent month shows acceleration — "
                    "forecast may underestimate future spend"
                )

        risks.append(
            "Provider pricing changes could impact forecast "
            "(review pricing.yaml quarterly)"
        )

        return risks
```

## Provider mix optimization

Most production systems use a single AI provider, but there are compelling reasons to consider a multi-provider strategy:

- **Price competition.** Different providers have different pricing for different model tiers. Using OpenAI for cheap tasks and Anthropic for complex tasks can reduce total cost.
- **Feature leverage.** Anthropic offers superior prompt caching. OpenAI offers automatic caching with less effort. Choosing the right provider per feature optimizes for each provider's strengths.
- **Negotiating leverage.** Having credible alternatives gives you negotiating power with each provider.

```
"""tokenwatch/provider_mix.py — Multi-provider cost optimization."""

from dataclasses import dataclass
from tokenwatch.config import TokenwatchConfig


@dataclass
class ProviderAllocation:
```

```
    """Optimal allocation of workload across providers."""
    provider: str
    model: str
    workload_type: str
    monthly_cost_usd: float
    monthly_requests: int
    cost_per_request: float


def optimize_provider_mix(
    config: TokenwatchConfig,
    workloads: list[dict],
) -> dict:
    """Find the optimal provider mix for a set of workloads.

    Args:
        workloads: List of {
            "name": str,
            "daily_requests": int,
            "avg_input_tokens": int,
            "avg_output_tokens": int,
            "min_quality": str,  # "low", "medium", "high"
            "latency_sensitive": bool,
            "batch_eligible": bool,
        }
    """
    allocations = []
    total_current = 0.0
    total_optimized = 0.0

    for workload in workloads:
        daily = workload["daily_requests"]
        monthly = daily * 30
        inp = workload["avg_input_tokens"]
        out = workload["avg_output_tokens"]
        quality = workload.get("min_quality", "medium")

        # Find cheapest model per provider that meets quality
        best_option = None
        best_cost = float("inf")

        for model_id, pricing in config.models.items():
            # Quality filter
            if quality == "high" and "haiku" in model_id.lower():
                continue
            if quality == "high" and "mini" in model_id.lower():
                continue

            # Calculate cost
```

```
            is_batch = workload.get("batch_eligible", False)
            cost_per_req = pricing.cost_for_tokens(
                input_tokens=inp,
                output_tokens=out,
                batch=is_batch,
            )
            monthly_cost = cost_per_req * monthly

            if monthly_cost < best_cost:
                best_cost = monthly_cost
                best_option = {
                    "provider": pricing.provider,
                    "model": model_id,
                    "monthly_cost": monthly_cost,
                    "cost_per_request": cost_per_req,
                }

        if best_option:
            allocations.append(ProviderAllocation(
                provider=best_option["provider"],
                model=best_option["model"],
                workload_type=workload["name"],
                monthly_cost_usd=best_option["monthly_cost"],
                monthly_requests=monthly,
                cost_per_request=best_option["cost_per_request"],
            ))
            total_optimized += best_option["monthly_cost"]

        # Calculate "use default model for everything" baseline
        default_pricing = config.get_model(config.default_model)
        default_cost = default_pricing.cost_for_tokens(
            input_tokens=inp, output_tokens=out
        ) * monthly
        total_current += default_cost

    savings = total_current - total_optimized

    return {
        "allocations": [
            {
                "workload": a.workload_type,
                "provider": a.provider,
                "model": a.model,
                "monthly_cost": a.monthly_cost_usd,
                "requests": a.monthly_requests,
            }
            for a in allocations
        ],
        "total_optimized_monthly": total_optimized,
```

```
        "total_default_monthly": total_current,
        "savings_monthly": savings,
        "savings_pct": (savings / total_current * 100) if total_current > 0 else
0,
    }
```

---

## Committed-use and volume agreements

At scale (typically >$10,000/month), AI providers offer committed-use agreements: you commit to a minimum monthly spend in exchange for a discount. The discount typically ranges from 10-30% depending on the commitment level and term.

### How committed-use agreements work in practice

A committed-use agreement is conceptually simple: you guarantee the provider a minimum monthly spend, and in return they reduce your per-token rate. The mechanics vary by provider, but the general structure includes a commitment level (the minimum monthly spend), a discount rate (the percentage reduction on token pricing), a term (typically 6-12 months), and terms for underuse (what happens if your actual spend falls below the commitment).

The critical variable is the commitment level relative to your forecasted spend. Commit too low, and you leave discount money on the table. Commit too high, and you pay for tokens you do not use. The optimal commitment level is typically 70-80% of your base forecast — high enough to capture meaningful discounts, low enough to avoid underuse penalties in a pessimistic scenario. The `evaluate_commitments` function automates this analysis by projecting spend across the commitment term and flagging months where spend might fall below the commitment.

### Renegotiation timing

Committed-use agreements are not set-and-forget. Three events should trigger a renegotiation review: a significant change in your workload volume (up or down), a provider pricing change that makes your committed rate less competitive than the new on-demand rate, and the approach of your term expiration (typically start discussions 60-90 days before expiration). The forecasting module provides the data for these reviews — run `forecast_monthly` with a horizon that covers your remaining term and compare the projection against your commitment level. If the projection shows consistent underuse, negotiate a lower commitment with a shorter term. If it shows consistent overage, negotiate a higher commitment for a deeper discount.

Provider pricing in the AI space has historically trended downward as competition increases and inference costs decline. This creates a specific risk for long-term commitments: you might lock in a rate that looks attractive today but becomes uncompetitive in 6 months when the provider (or a competitor) drops prices. Shorter terms (6 months rather than 12) provide more flexibility to capture these price reductions, at the cost of a smaller discount. The `evaluate_commitments` function models this tradeoff through the `term_months` parameter — compare 6-month and 12-month scenarios to see whether the deeper discount from a longer term justifies the pricing flexibility you give up.

```python
"""tokenwatch/commitments.py — Committed-use analysis."""

from dataclasses import dataclass


@dataclass
class CommitmentScenario:
    """A committed-use scenario to evaluate."""
    name: str
    monthly_commitment_usd: float
    discount_pct: float
    term_months: int
    penalty_for_underuse_pct: float  # penalty if you don't meet commitment


def evaluate_commitments(
    current_monthly_spend: float,
    projected_monthly_spend: float,
    growth_rate_monthly_pct: float,
    scenarios: list[CommitmentScenario],
) -> list[dict]:
    """Evaluate committed-use scenarios.

    Args:
        current_monthly_spend: Current monthly LLM API spend
        projected_monthly_spend: Forecasted spend at end of term
        growth_rate_monthly_pct: Expected monthly growth rate
        scenarios: Commitment options to evaluate
    """
    results = []

    for scenario in scenarios:
        term = scenario.term_months
        commitment = scenario.monthly_commitment_usd
        discount = scenario.discount_pct / 100

        # Project spend over the term
        total_spend_without_commitment = 0.0
        total_spend_with_commitment = 0.0
        months_under_commitment = 0

        for month in range(1, term + 1):
            projected = current_monthly_spend * (
                (1 + growth_rate_monthly_pct / 100) ** month
            )
            total_spend_without_commitment += projected

            if projected >= commitment:
                # Spend exceeds commitment — get full discount
```

```
                discounted = projected * (1 - discount)
                total_spend_with_commitment += discounted
            else:
                # Spend below commitment — pay commitment minimum
                total_spend_with_commitment += commitment * (1 - discount)
                months_under_commitment += 1

        savings = total_spend_without_commitment - total_spend_with_commitment
        risk = "LOW" if months_under_commitment == 0 else (
            "MEDIUM" if months_under_commitment <= 3 else "HIGH"
        )

        results.append({
            "scenario": scenario.name,
            "commitment_monthly": commitment,
            "discount_pct": scenario.discount_pct,
            "term_months": term,
            "total_without_commitment": total_spend_without_commitment,
            "total_with_commitment": total_spend_with_commitment,
            "total_savings": savings,
            "savings_pct": (savings / total_spend_without_commitment * 100)
                if total_spend_without_commitment > 0 else 0,
            "months_under_commitment": months_under_commitment,
            "risk_level": risk,
            "recommendation": (
                f"RECOMMENDED: saves ${savings:,.0f} over {term} months"
                if savings > 0 and risk != "HIGH"
                else f"RISKY: {months_under_commitment} months under commitment"
                if months_under_commitment > 0
                else f"NOT RECOMMENDED: no net savings"
            ),
        })

    return sorted(results, key=lambda x: x["total_savings"], reverse=True)
```

## Negotiation strategies

When approaching AI provider negotiations, the strongest position comes from data. Here's what to bring to the table:

**1. Detailed usage analytics.** Show your token volume by model, by month, with growth projections. Providers value predictable, growing customers.

**2. Multi-provider usage.** If you're using both Anthropic and OpenAI, each provider knows the other is a credible alternative. This creates competitive tension.

**3. Optimization potential.** Show that you're actively optimizing (caching, routing, batching). This demonstrates that you're a sophisticated customer who will grow usage efficiently.

**4. Commitment readiness.** Come with a specific commitment level in mind, backed by your forecast data. Providers offer better terms when you can commit confidently.

```
"""tokenwatch/negotiation.py — Negotiation preparation toolkit."""

from tokenwatch.store import UsageStore
from tokenwatch.forecast import SpendForecaster
from tokenwatch.provider_mix import optimize_provider_mix
from tokenwatch.config import TokenwatchConfig


def prepare_negotiation_brief(
    config: TokenwatchConfig,
    store: UsageStore,
) -> dict:
    """Prepare a data brief for provider negotiations.

    This generates the key metrics and talking points
    you need when negotiating with AI providers.
    """
    summary = store.summary()
    forecaster = SpendForecaster(store)
    forecast = forecaster.forecast_monthly(months_ahead=6)

    # Current usage profile
    total_cost = summary.get("total_cost_usd", 0)
    total_requests = summary.get("total_records", 0)
    by_model = summary.get("cost_by_model", {})

    # Growth trajectory
    growth_rates = [f.growth_rate_monthly_pct for f in forecast]
    avg_growth = sum(growth_rates) / len(growth_rates) if growth_rates else 0

    # Provider split
    provider_spend: dict[str, float] = {}
    for model, cost in by_model.items():
        provider = "anthropic" if "claude" in model else "openai"
        provider_spend[provider] = provider_spend.get(provider, 0) + cost

    brief = {
        "current_monthly_spend": total_cost,
        "total_requests_period": total_requests,
        "spend_by_provider": provider_spend,
        "spend_by_model": by_model,
        "growth_rate_monthly_pct": avg_growth,
        "six_month_projected_spend": forecast[-1].base_forecast_usd if forecast
else 0,
        "optimization_achievements": [
            f"Cache hit rate: {summary.get('cache_hit_rate', 0)*100:.0f}%",
```

```
            f"Avg cost per request: ${summary.get('avg_cost_per_request',
0):.6f}",
        ],
        "negotiation_points": [
            "Growing, predictable workload with sophisticated cost management",
            "Multi-provider strategy ensures best-of-breed per workload",
            "Committed to long-term AI adoption with responsible spend
practices",
        ],
    }

    return brief
```

I> **Note:** The negotiation brief is a tool for preparation, not a substitute for the actual conversation. Provider relationships are partnerships — the goal is a deal that works for both sides, not squeezing the last cent.

---

## Common mistakes

**1. Committing to more than your forecast supports.** A commitment that exceeds your forecasted spend means paying a guaranteed minimum you won't use. Commit at 80% of your base forecast, not 100%.

**2. Single-provider lock-in without leverage.** If 100% of your workload is on one provider and you have no multi-provider capability, you have zero negotiating leverage. Even a small allocation (10-20%) to an alternative provider creates meaningful competitive pressure.

**3. Forecasting based on features, not tokens.** "We're launching 3 new AI features next quarter" is not a forecast. Estimate each feature's daily request volume, average input/output tokens, and model requirements. Then calculate the cost.

**4. Ignoring pricing model changes.** Providers frequently adjust pricing — often downward. If you locked in a committed-use agreement at old prices and the provider drops prices, you might be paying more than the new on-demand rate. Look for agreements with price-match clauses.

**5. Forecasting without accounting for optimization.** If you plan to implement caching (Chapter 6), routing (Chapter 8), or batch processing (Chapter 7) during the forecast period, your per-token costs will change. A naive forecast that assumes constant per-request costs will overestimate spend after optimizations launch. Model the optimization impact explicitly: if you expect caching to reduce input token costs by 60% on a specific feature, adjust that feature's forecasted cost accordingly. The `planned_features` parameter in `SpendForecaster` can be repurposed for this — treat an optimization as a "feature" with negative estimated monthly cost.

**6. Not reviewing forecast accuracy.** Every month, compare your actual spend against last month's forecast. If the forecast consistently overestimates (your confidence bands are too wide) or underestimates (your growth rate is too low), adjust the model. Forecast accuracy improves with feedback — the `_identify_risks` method flags when recent growth is accelerating relative to the historical average, but it cannot detect slowdowns or seasonal effects without explicit

review. Make forecast-vs-actual comparison a standing item in your monthly AI Cost Review (Chapter 12).

---

## Exercises

**Easy:** Use `SpendForecaster` to project your current usage 3 months ahead. How does changing the growth rate assumption by +/- 5% affect the forecast? What's the difference between optimistic and pessimistic scenarios? Pay attention to how the confidence bands widen with each month — by month 3, the difference between optimistic and pessimistic is significantly larger than at month 1. This widening reflects genuine uncertainty: the further you project, the more things can change. This is why committed-use agreements should be calibrated to the pessimistic bound, not the base forecast.

**Medium:** Configure three commitment scenarios (low: $5K/month at 10% discount, medium: $15K/month at 20% discount, high: $30K/month at 25% discount) and evaluate them against your forecasted spend. Which offers the best risk-adjusted savings?

**Hard:** Build an end-to-end negotiation preparation tool that reads all historical usage data, generates a forecast, runs the provider mix optimization, evaluates commitment scenarios, and produces a formatted one-page brief suitable for sharing with a vendor account manager. Include charts (using matplotlib) showing spend trends, provider allocation, and forecast ranges.

---

## Summary

In this chapter you learned:

- **Forecasting AI spend** requires analyzing historical patterns, growth rates, and planned feature launches — confidence bands should widen with the forecast horizon
- **Provider mix optimization** finds the cheapest model per workload type across all configured providers, revealing savings from multi-provider strategies
- **Committed-use agreements** trade guaranteed spend for discounts — evaluate them against your forecast to ensure the commitment level is achievable
- **Negotiation leverage** comes from data (detailed analytics), alternatives (multi-provider usage), and sophistication (active optimization)
- **Forecast accuracy depends on consistent metering** — the investment in Chapters 3-4 pays off directly in budgeting and negotiation confidence

**Next chapter**

You've built all the pieces: metering, economics, optimization (prompt, cache, batch, routing), governance, and forecasting. Chapter 14 brings it all together into a complete production deployment — the full tokenwatch integration, a real-time cost dashboard, and an operational runbook.

---

A> **Code for this chapter:** `project/cap13/tokenwatch/`

# Chapter 14 — Final Project: Full Tokenwatch Integration

Over the past thirteen chapters, you've built tokenwatch piece by piece: config, token counting, metering, economics, prompt optimization, caching, batching, routing, RAG cost tracking, agent controls, self-hosting analysis, governance, and forecasting. Each module works independently, but the real value emerges when they work together as an integrated system.

This chapter brings everything together. We'll build the final integration layer, create a real-time cost dashboard, write an operational runbook, and deploy tokenwatch as a production-ready toolkit that you can drop into any Python LLM application. By the end, you'll have a complete, working system — and the confidence to operate LLM workloads with full cost visibility and control.

---

## The integrated architecture

Let's start with the architectural overview. tokenwatch's modules compose into a pipeline that every LLM request flows through:

```
Request arrives
    |
[1] Governance check (budget + quota)
    |-- BLOCKED → return error
    |-- DOWNGRADED → switch to cheaper model
    |-- OK → continue
    |
[2] Routing decision (complexity classification)
    |-- Select model based on complexity + policy
    |
[3] Prompt optimization
    |-- Compress system prompt
    |-- Truncate history
    |-- Build cache-friendly structure
    |
[4] Pre-flight estimation
    |-- Count tokens, estimate cost
    |-- Final budget check with estimate
    |
[5] API call (metered)
    |-- Record latency, actual tokens, cost
    |-- Track cache hit/miss
    |
[6] Post-call analytics
    |-- Update usage store
    |-- Update budget tracking
    |-- Check for cost anomalies
    |
Response returned with cost metadata
```

Here's the integration layer that orchestrates all modules:

```
"""tokenwatch/pipeline.py — Integrated request pipeline."""

import time
from dataclasses import dataclass, field
from typing import Any

import anthropic

from tokenwatch.config import TokenwatchConfig
from tokenwatch.middleware import TokenMeter
from tokenwatch.store import UsageStore
from tokenwatch.meter import UsageRecord
from tokenwatch.governance import BudgetManager, BudgetAction
from tokenwatch.quotas import QuotaManager
from tokenwatch.routing_engine import RoutingEngine
from tokenwatch.router import RoutingPolicy, RoutingDecision
from tokenwatch.cache import CacheOptimizer
from tokenwatch.history import truncate_history
from tokenwatch.preflight import preflight_check


@dataclass
class PipelineResult:
    """Complete result from the tokenwatch pipeline."""
    response: anthropic.types.Message
    usage: UsageRecord
    routing: RoutingDecision
    governance_action: str
    model_used: str
    cost_usd: float
    latency_ms: float
    cache_hit: bool
    warnings: list[str] = field(default_factory=list)


class TokenwatchPipeline:
    """The complete tokenwatch pipeline for production use.

    Integrates governance, routing, caching, metering, and
    analytics into a single request flow.
    """

    def __init__(
        self,
        config: TokenwatchConfig,
        store: UsageStore | None = None,
        budget_manager: BudgetManager | None = None,
        quota_manager: QuotaManager | None = None,
```

```
        routing_policy: RoutingPolicy | None = None,
        client: anthropic.Anthropic | None = None,
        environment: str = "production",
    ):
        self.config = config
        self.store = store or UsageStore()
        self.client = client or anthropic.Anthropic()

        self.meter = TokenMeter(config, self.store, environment=environment)
        self.budget_manager = budget_manager or BudgetManager(self.store)
        self.quota_manager = quota_manager or QuotaManager()
        self.cache_optimizer = CacheOptimizer(config)
        self.routing_engine = RoutingEngine(
            config,
            routing_policy or RoutingPolicy.anthropic_cascade(),
            meter=self.meter,
            client=self.client,
        )

    def send(
        self,
        messages: list[dict],
        system: str = "",
        feature: str = "unknown",
        user_id: str = "",
        team: str = "unknown",
        max_tokens: int = 4096,
        tools: list[dict] | None = None,
        enable_cache: bool = True,
        enable_routing: bool = True,
        max_history_tokens: int = 8000,
        **kwargs,
    ) -> PipelineResult:
        """Send a request through the complete tokenwatch pipeline.

        This is the main entry point for production use.
        All governance, routing, optimization, and metering
        happens automatically.
        """
        warnings = []

        # --- Step 1: Governance check ---
        budget_result = self.budget_manager.check_budget(
            feature=feature, team=team, estimated_cost=0.01
        )
        quota_ok, quota_msg = self.quota_manager.check_and_increment(feature,
team)

        if not budget_result.allowed:
```

```
            raise RuntimeError(
                f"Request blocked by governance: {budget_result.message}"
            )
        if not quota_ok:
            raise RuntimeError(f"Request blocked by quota: {quota_msg}")

        # Model override from governance
        model_override = None
        governance_action = "OK"
        if budget_result.action == BudgetAction.DOWNGRADE:
            model_override = budget_result.downgrade_model
            governance_action = "DOWNGRADED"
            warnings.append(f"Downgraded to {model_override}:
{budget_result.message}")

        if budget_result.is_warning:
            warnings.append(f"Budget warning: {budget_result.message}")

        # --- Step 2: Routing ---
        if enable_routing and not model_override:
            routing_decision = self.routing_engine.route(
                messages=messages,
                system=system,
                feature=feature,
                tool_count=len(tools) if tools else 0,
            )
            model = routing_decision.model
        elif model_override:
            from tokenwatch.router import Complexity
            routing_decision = RoutingDecision(
                model=model_override,
                complexity=Complexity.LOW,
                reason="Governance downgrade",
                estimated_cost_ratio=0.1,
            )
            model = model_override
        else:
            from tokenwatch.router import Complexity
            model = self.config.default_model
            routing_decision = RoutingDecision(
                model=model,
                complexity=Complexity.MEDIUM,
                reason="Routing disabled, using default",
                estimated_cost_ratio=1.0,
            )

        # --- Step 3: Prompt optimization ---
        optimized_messages = truncate_history(
            messages, max_tokens=max_history_tokens
```

```
)

# Build cache-friendly payload
if enable_cache and system:
    cached_payload = self.cache_optimizer.build_cached_messages(
        system_prompt=system,
        tool_definitions=tools,
        user_message="",  # already in messages
        conversation_history=optimized_messages,
    )
    api_system = cached_payload["system"]
    api_messages = cached_payload["messages"]
else:
    api_system = system
    api_messages = optimized_messages

# --- Step 4: Pre-flight check ---
estimate = preflight_check(
    self.config, model, system, optimized_messages, max_tokens
)
if "BLOCK" in estimate.recommendation:
    raise RuntimeError(f"Pre-flight blocked: {estimate.recommendation}")
if "WARN" in estimate.recommendation:
    warnings.append(f"Pre-flight warning: {estimate.recommendation}")

# --- Step 5: API call (metered) ---
api_kwargs = {
    "model": model,
    "max_tokens": max_tokens,
    "system": api_system,
    "messages": api_messages,
}
if tools:
    api_kwargs["tools"] = tools
api_kwargs.update(kwargs)

start = time.monotonic()
response = self.client.messages.create(**api_kwargs)
latency_ms = (time.monotonic() - start) * 1000

# --- Step 6: Post-call metering ---
record = self.meter.measure_anthropic(
    response=response,
    latency_ms=latency_ms,
    feature=feature,
    user_id=user_id,
    team=team,
)
```

```
        # Track cache performance
        cache_result = self.cache_optimizer.track_cache_result(response, model)

        return PipelineResult(
            response=response,
            usage=record,
            routing=routing_decision,
            governance_action=governance_action,
            model_used=model,
            cost_usd=record.cost_usd,
            latency_ms=latency_ms,
            cache_hit=cache_result.get("cache_hit", False),
            warnings=warnings,
        )
```

---

## Quick-start setup

For teams adopting tokenwatch, here's the minimal setup that gets you from zero to full cost visibility:

```
"""quickstart.py — Minimal tokenwatch setup for a new project."""

from tokenwatch.config import TokenwatchConfig
from tokenwatch.pipeline import TokenwatchPipeline
from tokenwatch.governance import Budget, BudgetAction, BudgetManager
from tokenwatch.store import UsageStore

# 1. Load pricing configuration
config = TokenwatchConfig.from_yaml("pricing.yaml")

# 2. Initialize the pipeline
pipeline = TokenwatchPipeline(
    config=config,
    environment="production",
)

# 3. Add basic budgets
pipeline.budget_manager.add_budget(Budget(
    name="daily-global",
    limit_usd=500.0,
    period="daily",
    scope_type="global",
    scope_value="*",
    action=BudgetAction.WARN,
))

# 4. Make requests through the pipeline
result = pipeline.send(
    messages=[{"role": "user", "content": "Classify this ticket: 'My payment
```

```
failed'"}],
    system="Classify the support ticket. Respond with one word: billing,
technical, or account.",
    feature="ticket-classifier",
    team="support-platform",
    user_id="user-456",
    max_tokens=10,
)

print(f"Response: {result.response.content[0].text}")
print(f"Model: {result.model_used}")
print(f"Cost: ${result.cost_usd:.6f}")
print(f"Latency: {result.latency_ms:.0f}ms")
print(f"Cache hit: {result.cache_hit}")
print(f"Routing: {result.routing.reason}")

if result.warnings:
    for w in result.warnings:
        print(f"Warning: {w}")

# 5. Generate a cost report
from tokenwatch.dashboard import generate_cost_report
report = generate_cost_report(pipeline.store)
print(report)
```

---

## The operational runbook

Operating tokenwatch in production requires monitoring, alerting, and regular maintenance. Here's the runbook:

```
"""tokenwatch/runbook.py — Operational procedures and health checks."""

from dataclasses import dataclass
from tokenwatch.store import UsageStore
from tokenwatch.config import TokenwatchConfig
from tokenwatch.cache import CacheMonitor
from tokenwatch.economics import EconomicsEngine


@dataclass
class HealthCheck:
    """Result of a system health check."""
    name: str
    status: str  # "OK", "WARNING", "CRITICAL"
    message: str
    action: str  # what to do if not OK


def run_health_checks(
```

```
    store: UsageStore,
    config: TokenwatchConfig,
    cache_monitor: CacheMonitor | None = None,
) -> list[HealthCheck]:
    """Run all tokenwatch health checks.

    Call this periodically (every 5 minutes) or on-demand.
    """
    checks = []

    # Check 1: Data freshness
    records = store.query()
    if not records:
        checks.append(HealthCheck(
            "data_freshness", "CRITICAL",
            "No usage data found",
            "Verify metering middleware is active",
        ))
    else:
        latest = max(r.timestamp for r in records)
        checks.append(HealthCheck(
            "data_freshness", "OK",
            f"Latest record: {latest}",
            "N/A",
        ))

    # Check 2: Cost anomaly detection
    engine = EconomicsEngine(store)
    features = engine.feature_economics()
    for f in features:
        if f.cost_trend_pct > 100:
            checks.append(HealthCheck(
                f"cost_trend_{f.feature}", "WARNING",
                f"{f.feature}: cost up {f.cost_trend_pct:.0f}% vs previous
period",
                f"Review recent changes to {f.feature} prompts and models",
            ))

    # Check 3: Cache health
    if cache_monitor:
        cache_health = cache_monitor.health_report()
        if cache_health.current_hit_rate < 0.5 and cache_health.total_savings_usd
> 0:
            checks.append(HealthCheck(
                "cache_hit_rate", "WARNING",
                f"Cache hit rate: {cache_health.current_hit_rate:.0%}",
                "Check prompt structure — variable content may be before stable
content",
            ))
```

```
        for alert in cache_health.alerts:
            checks.append(HealthCheck(
                "cache_alert", "WARNING", alert,
                "Investigate cache configuration",
            ))

    # Check 4: Pricing config freshness
    checks.append(HealthCheck(
        "pricing_config", "OK",
        "Reminder: verify pricing.yaml is current",
        "Check provider pricing pages and update pricing.yaml quarterly",
    ))

    # Check 5: High-cost requests
    high_cost = [r for r in records if r.cost_usd > 1.0]
    if high_cost:
        checks.append(HealthCheck(
            "high_cost_requests", "WARNING",
            f"{len(high_cost)} requests cost >$1.00 each",
            "Review these requests for optimization opportunities or budget
limits",
        ))

    return checks


def daily_maintenance(store: UsageStore, config: TokenwatchConfig) -> str:
    """Run daily maintenance tasks and return a summary.

    Schedule this as a daily cron job or periodic task.
    """
    lines = []
    lines.append("=== TOKENWATCH DAILY MAINTENANCE ===\n")

    # Health checks
    checks = run_health_checks(store, config)
    critical = [c for c in checks if c.status == "CRITICAL"]
    warnings = [c for c in checks if c.status == "WARNING"]

    lines.append(f"Health checks: {len(checks)} total, "
                 f"{len(critical)} critical, {len(warnings)} warnings\n")

    for c in critical:
        lines.append(f"  CRITICAL: {c.name} — {c.message}")
        lines.append(f"    Action: {c.action}")

    for c in warnings:
        lines.append(f"  WARNING: {c.name} — {c.message}")
        lines.append(f"    Action: {c.action}")
```

```
    # Daily cost summary
    from tokenwatch.dashboard import generate_cost_report
    report = generate_cost_report(store)
    lines.append(f"\n{report}")

    return "\n".join(lines)
```

**Daily procedures**

1. **Morning check (automated):** Run `daily_maintenance()` and review the output. Look for critical alerts and cost trend warnings.
2. **Pricing verification (quarterly):** Visit provider pricing pages (Anthropic, OpenAI) and update `pricing.yaml` if prices have changed. Run `npm run validate` after updating.
3. **Budget review (monthly):** Review budget utilization across teams. Adjust limits based on actual spend patterns and upcoming feature launches.
4. **Optimization review (quarterly):** Run the cost dashboard, identify the top 3 features by cost, and evaluate each for optimization opportunities (caching, routing, prompt compression, batching).

---

## Deployment patterns

**Pattern 1: Centralized gateway**

All LLM requests flow through a centralized tokenwatch gateway. This gives you complete visibility and control, but adds a network hop and requires the gateway to be highly available.

```
"""tokenwatch/gateway.py — Centralized API gateway pattern."""

from tokenwatch.pipeline import TokenwatchPipeline, PipelineResult


class TokenwatchGateway:
    """Centralized gateway for all LLM requests.

    Deploy as a microservice that all teams call
    instead of calling LLM APIs directly.
    """

    def __init__(self, pipeline: TokenwatchPipeline):
        self.pipeline = pipeline

    def handle_request(self, request: dict) -> dict:
        """Handle an incoming request from any team.

        Expected request format:
        {
            "messages": [...],
```

```
    "system": "...",
    "feature": "...",
    "team": "...",
    "user_id": "...",
    "max_tokens": 4096,
    "tools": [...],
}
"""
try:
    result = self.pipeline.send(**request)
    return {
        "status": "success",
        "content": [
            {"type": b.type, "text": getattr(b, "text", "")}
            for b in result.response.content
        ],
        "model": result.model_used,
        "cost_usd": result.cost_usd,
        "latency_ms": result.latency_ms,
        "cache_hit": result.cache_hit,
        "warnings": result.warnings,
    }
except RuntimeError as e:
    return {
        "status": "blocked",
        "error": str(e),
    }
```

**Pattern 2: SDK middleware**

Distribute tokenwatch as a Python SDK that teams install in their services. Each service runs its own metering and governance, with centralized data collection via a shared usage store (database, event stream).

This is simpler to deploy and doesn't require a centralized gateway, but relies on teams correctly integrating the SDK.

**Deployment pattern comparison**

| Dimension | Centralized gateway | SDK middleware |
|---|---|---|
| **Visibility** | Complete — every request passes through one point | Partial — depends on correct integration |
| **Governance enforcement** | Guaranteed — can't bypass | Advisory — teams could call APIs directly |
| **Latency overhead** | +10-50ms network hop | <1ms in-process |
| **Deployment complexity** | High — must be highly available | Low — just a pip install |
| **Team autonomy** | Low — central team controls all config | High — each team configures independently |
| **Single point of failure** | Yes — gateway outage blocks all AI calls | No — failures are isolated per service |
| **Best for** | Regulated environments, cost-sensitive orgs | Fast-moving teams, microservice architectures |

Most organizations start with the SDK pattern for speed, then migrate high-spend features to the gateway pattern once governance maturity reaches Level 3 (see Chapter 12). A hybrid approach is common at scale: the gateway handles the top 5 features by cost, while long-tail features use the SDK with soft budgets.

---

## Worked example: tracing a request through the full pipeline

Let's walk through a single request and calculate the cost impact of each pipeline stage, using `pricing.yaml` rates.

**Scenario:** A customer-facing chat feature receives a message: "Can you explain my last three transactions?" The conversation has 15 prior turns.

**Without tokenwatch (naive approach):**

| Component | Tokens | Rate | Cost |
|---|---|---|---|
| System prompt | 800 input | $3.00/M | $0.0024 |
| Full 15-turn history | 12,000 input | $3.00/M | $0.0360 |
| User message | 200 input | $3.00/M | $0.0006 |
| Output (Sonnet) | 500 output | $15.00/M | $0.0075 |
| **Total per request** | | | **$0.0465** |

At 50,000 requests per day, the naive approach costs **$2,325/day** or roughly **$69,750/month**.

**With tokenwatch pipeline, step by step:**

**Step 1 — Governance check:** Budget is at 60% of daily limit. Action: OK. No cost impact, but the guardrail is in place.

**Step 2 — Routing:** The classifier scores this as LOW complexity (simple transaction lookup, no reasoning required). Route to Haiku instead of Sonnet.

**Step 3 — Prompt optimization:** History truncation reduces 15 turns (12,000 tokens) to the most recent 5 turns (4,000 tokens). Cache-friendly prompt structure places the stable system prompt first.

**Step 4 — Pre-flight check:** Estimated cost $0.0046. Well within budget. Proceed.

**Step 5 — API call (metered):** Cache hits on the system prompt (800 tokens at cached rate).

| Component | Tokens | Rate | Cost |
|---|---|---|---|
| System prompt (cached) | 800 input | $0.08/M | $0.0001 |
| Truncated history | 4,000 input | $0.80/M | $0.0032 |
| User message | 200 input | $0.80/M | $0.0002 |
| Output (Haiku) | 400 output | $4.00/M | $0.0016 |
| **Total per request** | | | **$0.0051** |

**Step 6 — Post-call analytics:** Record logged, budget updated, no anomalies.

**Pipeline savings per request:** $0.0465 - $0.0051 = **$0.0414 (89% reduction)**

At 50,000 requests per day: **$255/day** vs. $2,325/day. Monthly savings: **$62,100**.

The breakdown of savings by technique:

| Technique | Savings contribution |
|---|---|
| Model routing (Sonnet ⟶ Haiku) | 73% of output cost, ~55% of total |
| History truncation (15 ⟶ 5 turns) | 67% of history tokens saved, ~25% of total |
| Prompt caching | 97% off system prompt cost, ~9% of total |
| Combined | 89% total cost reduction |

This example illustrates why the pipeline is more powerful than any single optimization. Routing alone saves ~55%, but combining it with truncation and caching pushes savings to 89%.

---

## Real-world scenario: from prototype to production

At a large fintech, an engineering team built a customer-facing FAQ chatbot using Claude Sonnet for all requests. During the prototype phase with 500 requests per day, the cost was manageable — roughly $23/day. Leadership approved production deployment expecting 10x traffic growth.

**Week 1 (production launch, no tokenwatch):** Traffic ramped to 5,000 requests per day. Cost: $232/day, $6,960/month. Finance flagged the expense.

**Week 2 (metering added):** The team deployed tokenwatch metering (Chapter 3). The cost dashboard revealed that 70% of requests were simple FAQ lookups ("What are your business hours?", "How do I reset my password?") averaging only 150 output tokens. The remaining 30% were complex queries requiring reasoning.

**Week 3 (routing added):** Based on the metering data, the team deployed a two-tier routing cascade (Chapter 8). Simple FAQ queries routed to Haiku, complex queries stayed on Sonnet. Result: cost dropped from $232/day to $112/day — a 52% reduction with no change in user satisfaction scores.

**Week 4 (caching + truncation):** The stable 600-token system prompt was restructured for caching (Chapter 6), and conversation history was truncated to 5 turns (Chapter 5). Additional savings of $38/day, bringing the total to $74/day.

**Month 2 (governance):** Daily budgets were set at $100 (WARN) and $150 (DOWNGRADE). When a marketing campaign drove traffic to 15,000 requests in a single day, governance automatically downgraded overflow requests to Haiku, keeping costs at $142 instead of an estimated $410.

**Final result:** Monthly cost stabilized at $2,220 — 68% below the unoptimized projection of $6,960. The entire optimization effort took 4 weeks of incremental work, following the same sequence this book teaches.

---

## The complete module map

Here's every module in the final tokenwatch package:

```
tokenwatch/
    __init__.py          # Package init, version
    config.py            # Pricing configuration, model registry
    tokens.py            # Token counting and estimation
    context.py           # Context window analysis
    preflight.py         # Pre-request cost estimation
    meter.py             # Usage record data model
    store.py             # Usage record storage (JSONL)
    middleware.py        # Metering middleware (Anthropic + OpenAI)
    client.py            # Metered client wrappers
    attribution.py       # Cost attribution analysis
    economics.py         # Unit economics engine
    margins.py           # Margin analysis
    distribution.py      # Cost distribution analysis
    dashboard.py         # Cost dashboard report generator
    prompt.py            # Prompt compression utilities
    history.py           # Conversation history management
    prompt_test.py       # Prompt A/B testing
    cache.py             # Prompt caching utilities
```

```
cache_economics.py   # Cache break-even analysis
cache_monitor.py     # Cache health monitoring
batch.py             # Batch processing
batch_queue.py       # Automatic batch collection
batch_router.py      # Sync vs. batch routing
batch_analysis.py    # Batch savings analysis
batch_retry.py       # Batch failure handling
router.py            # Model routing data models
classifier.py        # Complexity classification
routing_engine.py    # Main routing engine
cascade.py           # Quality-gated cascades
routing_analytics.py # Routing effectiveness analysis
rag.py               # RAG cost tracking
chunking.py          # Token-aware chunking
rag_optimizer.py     # RAG cost optimization
embedding_costs.py   # Embedding model comparison
agents.py            # Agent budget controls
agent_loop.py        # Budget-aware agent execution
tool_optimizer.py    # Tool call optimization
agent_memory.py      # Cost-aware memory management
selfhost.py          # Self-hosting break-even
quantization.py      # Quantization trade-off analysis
serverless.py        # Serverless inference comparison
governance.py        # Budget enforcement
quotas.py            # Request quota management
showback.py          # Showback/chargeback reports
forecast.py          # Spend forecasting
provider_mix.py      # Multi-provider optimization
commitments.py       # Committed-use analysis
negotiation.py       # Negotiation preparation
pipeline.py          # Integrated request pipeline
gateway.py           # Centralized gateway pattern
runbook.py           # Operational procedures
```

---

## Common mistakes

**1. Deploying without budgets.** It's tempting to "just add metering" and defer governance. Don't. At minimum, deploy with a daily global budget in WARN mode. You'll catch runaway costs immediately.

*Why this fails:* At a large fintech, a team deployed metering and planned to "add budgets next sprint." A prompt regression in week two caused a 4x cost increase that ran for three days before anyone noticed the dashboard. A simple WARN budget would have triggered an alert within hours. The rule of thumb: if you're deploying metering, add `BudgetManager` with at least one global WARN budget in the same pull request.

**2. Not testing the pipeline end-to-end.** Each module works independently, but integration issues appear at the boundaries. Test the full pipeline with realistic traffic before production deployment.

*Why this fails:* The most common integration bug is ordering — for example, applying prompt caching *before* history truncation, which means the cache key includes variable-length history and cache hit rates drop to near zero. Another classic: the routing engine selects Haiku, but the cache optimizer builds cache blocks using Sonnet's token limits. End-to-end tests with 50-100 representative requests will surface these issues before they cost you money.

**3. Over-optimizing early.** Start with metering and basic governance. Add routing, caching, and batching one at a time, measuring the impact of each. Over-optimizing before you understand your cost patterns wastes engineering effort.

*Why this fails:* A team spent two weeks building a sophisticated 3-tier routing cascade before deploying metering. After launch, the dashboard showed that 95% of their requests were in the MEDIUM complexity band, meaning the elaborate LOW and HIGH tiers processed only 5% of traffic. If they had deployed metering first, they would have built a simpler 2-tier cascade targeting the actual traffic distribution, saving a week of engineering time.

**4. Not updating pricing.yaml.** Provider pricing changes. If your pricing.yaml is stale, your cost reports are wrong and your budgets are miscalibrated. Set a quarterly calendar reminder.

*Why this fails:* When providers cut prices (which happens frequently — Anthropic and OpenAI have both reduced prices multiple times), stale pricing means your dashboards over-report costs and your budgets trigger too early. Conversely, if a provider introduces a new pricing tier or increases rates for a model you use, stale pricing under-reports costs and your budgets are too loose. The `run_health_checks()` function includes a pricing freshness reminder, but you should also subscribe to provider announcements.

**5. Treating the pipeline as "done."** The pipeline is not a one-time deployment — it's an ongoing practice. New features change traffic patterns, new models change the cost landscape, and organizational growth changes governance requirements.

*Why this fails:* A team deployed tokenwatch with routing that sent 60% of traffic to Haiku. Six months later, a new Haiku version launched with different pricing, a new feature doubled traffic volume, and two teams that didn't exist at deployment time started using the pipeline. None of the budgets, routing rules, or forecasts reflected these changes. The quarterly optimization review (item 4 in the operational runbook) exists specifically to prevent this drift.

---

## Exercises

**Easy:** Deploy the `TokenwatchPipeline` with the quick-start configuration. Make 20 requests across 3 different features. Generate the cost report and verify that attribution is working correctly.

**Medium:** Set up a realistic governance configuration: daily budgets per team (WARN at 80%, DOWNGRADE at 100%), hourly quotas per feature (to prevent burst abuse), and a monthly global budget (BLOCK). Simulate a day of traffic and verify that all governance controls fire correctly.

**Hard:** Deploy tokenwatch in gateway mode serving traffic from two simulated teams. Each team has different features, models, and cost patterns. Run a full day of simulated traffic, then produce: (1) the daily maintenance report, (2) a showback report per team, (3) a forecast for the next quarter, and (4) a negotiation brief for your largest provider. Present this as a complete FinOps package.

---

## Summary

In this chapter you learned:

- **The tokenwatch pipeline** integrates governance, routing, optimization, metering, and analytics into a single request flow
- **Quick-start setup** gets you from zero to full cost visibility in under 50 lines of configuration code
- **The operational runbook** covers daily health checks, quarterly pricing reviews, monthly budget reviews, and ongoing optimization
- **Deployment patterns** — centralized gateway vs. distributed SDK — each have trade-offs in visibility, complexity, and team adoption
- **The complete module map** shows all 40+ modules that compose the tokenwatch toolkit

**Next steps**

You've built a complete AI cost engineering toolkit from scratch. Here's what to do next:

1. **Deploy metering first.** Get visibility into your current costs before optimizing anything.
2. **Identify your top 3 cost drivers.** Use the economics engine to find the features that account for the most spend.
3. **Apply the highest-leverage optimization.** Usually it's caching or routing — pick the one with the biggest impact on your top cost drivers.
4. **Set up governance.** Even soft budgets (WARN mode) catch problems early.
5. **Review quarterly.** Pricing changes, usage patterns shift, and new features launch. Keep your cost engineering current.

The AI cost landscape will continue to evolve — prices will drop, new model tiers will appear, new optimization techniques will emerge. But the fundamentals you've learned in this book are durable: measure first, optimize informed, govern consistently, and never stop questioning whether you're getting the best value for every token you spend.

---

A> **Code for this chapter:** `project/cap14/tokenwatch/`

# Appendix — Cost Optimization Checklist and Pricing Worksheet

This appendix provides two reference tools for ongoing use: a comprehensive cost optimization checklist that you can run through quarterly, and a pricing worksheet template for tracking provider rates and calculating workload costs.

---

## Cost optimization checklist

Use this checklist as a quarterly review tool. Go through each section, check what's in place, and identify what's missing. The sections are ordered by typical impact — start from the top.

### Visibility (Chapter 3-4)

- ☐ Every LLM API call is metered with actual token counts from the response
- ☐ Every request is tagged with feature name, team, and user ID
- ☐ Usage data is stored in a queryable format (JSONL, database, event stream)
- ☐ A cost dashboard is available showing per-feature and per-team breakdowns
- ☐ Unit economics are calculated: cost per request, cost per user, cost per feature
- ☐ Cost distribution analysis is available (P50/P90/P95/P99 per feature)
- ☐ Cost trends are tracked week-over-week and month-over-month

### Prompt optimization (Chapter 5)

- ☐ System prompts have been reviewed for unnecessary verbosity
- ☐ Role-play preambles ("You are an expert…") have been tested without
- ☐ Structured output (JSON schema) is used for classification/extraction tasks
- ☐ Output tokens are constrained where appropriate (max_tokens, format instructions)
- ☐ Tool definitions are concise (short descriptions, minimal parameter docs)
- ☐ Few-shot examples are tested against zero-shot to verify they're necessary

### Caching (Chapter 6)

- ☐ Prompt caching is enabled for endpoints with >1 request/second
- ☐ Prompts are structured with stable content first, variable content last
- ☐ Cache hit rates are monitored with alerts on drops below 70%
- ☐ Cache break-even analysis has been run for each cacheable endpoint
- ☐ Cache-unfriendly patterns (variable content in system prompt) have been fixed

### Conversation management (Chapter 2, 5)

- ☐ Conversation history is truncated to a maximum token budget
- ☐ Old messages are summarized or dropped, not sent in full
- ☐ System prompts are not re-sent redundantly across turns (rely on caching)

- ☐ The cost of multi-turn conversations is tracked per-conversation, not just per-turn

**Batch processing (Chapter 7)**

- ☐ Latency-tolerant workloads have been identified and catalogued
- ☐ Batch-eligible workloads are processed via the batch API at 50% discount
- ☐ Batch routing is configured per feature (sync vs. batch)
- ☐ Batch failure handling and retry logic is in place
- ☐ Batch savings are measured and reported

**Model routing (Chapter 8)**

- ☐ A model cascade is configured (cheap model for simple tasks, expensive for complex)
- ☐ Complexity classification is in place (rule-based or LLM-based)
- ☐ Routing effectiveness is measured (traffic distribution across models)
- ☐ Escalation rate is monitored (target: <20% escalation from cheapest tier)
- ☐ Quality verification confirms that routing doesn't degrade user experience

**RAG optimization (Chapter 9)**

- ☐ Top-k retrieval is tuned (start with k=3, not k=10)
- ☐ Relevance thresholding filters low-score chunks before injection
- ☐ Chunk size is optimized for the workload (smaller for QA, larger for analysis)
- ☐ Context utilization is measured (are retrieved chunks actually used?)
- ☐ Embedding model cost is compared across providers

**Agent controls (Chapter 10)**

- ☐ Per-request cost budgets are set for agent workloads
- ☐ Iteration limits prevent infinite loops (max 10-15 iterations)
- ☐ Tool call limits prevent excessive tool usage
- ☐ Dynamic tool loading sends only relevant tools per iteration
- ☐ Agent memory is compressed to prevent context growth across iterations
- ☐ Tool results are summarized after processing to reduce context size

**Self-hosting evaluation (Chapter 11)**

- ☐ Break-even analysis has been run for highest-volume workloads
- ☐ Quantization options have been compared (INT8 is the typical sweet spot)
- ☐ Hidden costs (engineering, redundancy, updates) are included in the analysis
- ☐ Serverless inference platforms have been considered as a middle ground
- ☐ Quality comparison has been done on actual tasks, not just benchmarks

### Governance (Chapter 12)

- ☐ Daily budgets are set per feature and per team
- ☐ Monthly global budget has a hard limit (BLOCK action)
- ☐ Budget alerts fire at 80% utilization
- ☐ Request quotas prevent burst abuse
- ☐ Showback reports are generated and shared with team leads monthly
- ☐ An exception process exists for temporary budget increases
- ☐ New features launch with metering and budgets pre-configured

### Forecasting and negotiation (Chapter 13)

- ☐ Monthly spend forecast is generated with confidence bands
- ☐ Planned feature launches are included in forecasts
- ☐ Committed-use agreements are evaluated against forecasts
- ☐ Multi-provider pricing is compared for each workload type
- ☐ Negotiation brief is prepared with detailed usage analytics

### Operational (Chapter 14)

- ☐ pricing.yaml is verified against provider pricing pages quarterly
- ☐ Health checks run periodically (every 5 minutes recommended)
- ☐ Cost anomaly detection alerts on >2x deviation from baseline
- ☐ The operational runbook is documented and accessible to on-call engineers
- ☐ Cache hit rates, routing distributions, and batch savings are in dashboards

---

## Pricing worksheet

Use this worksheet to track current pricing and calculate workload costs. Update it when provider pricing changes.

### Current provider pricing

```
# pricing_worksheet.yaml
# Last updated: YYYY-MM-DD
# Source: [provider pricing page URL]

providers:
  anthropic:
    pricing_url: "https://docs.anthropic.com/en/docs/about-claude/pricing"
    models:
      claude-opus-4-6:
        input_per_million: _____
        output_per_million: _____
        cached_per_million: _____
        batch_input_per_million: _____
```

```
      batch_output_per_million: _____
    claude-sonnet-4-6:
      input_per_million: _____
      output_per_million: _____
      cached_per_million: _____
      batch_input_per_million: _____
      batch_output_per_million: _____
    claude-haiku-4-5:
      input_per_million: _____
      output_per_million: _____
      cached_per_million: _____
      batch_input_per_million: _____
      batch_output_per_million: _____

openai:
  pricing_url: "https://openai.com/pricing"
  models:
    gpt-4o:
      input_per_million: _____
      output_per_million: _____
      cached_per_million: _____
      batch_input_per_million: _____
      batch_output_per_million: _____
    gpt-4o-mini:
      input_per_million: _____
      output_per_million: _____
      cached_per_million: _____
      batch_input_per_million: _____
      batch_output_per_million: _____
```

**Workload cost calculator**

For each AI-powered feature, fill in the parameters and calculate the monthly cost:

```
Feature: _____________________________
Model: ______________________________
Daily requests: _____________________
Avg input tokens: ___________________
Avg output tokens: __________________
Cache hit rate: _____________________%
Batch eligible: _____________________%

Monthly calculation:
  Total monthly requests = daily_requests x 30
  Input token cost = (avg_input x monthly_requests / 1M) x input_rate x (1 -
cache_hit_rate x 0.9)
  Output token cost = (avg_output x monthly_requests / 1M) x output_rate
  Batch discount = (batch_eligible% x total_cost) x 0.5
```

```
Estimated monthly cost = input_cost + output_cost - batch_discount
Cost per request = monthly_cost / monthly_requests
```

**Optimization impact tracker**

Track the impact of each optimization technique:

```
Optimization          | Before $/month | After $/month | Savings | Date applied
----------------------+----------------+---------------+---------+-------------
Prompt compression    | _______________ | _____________ | _______ | ___________
Prompt caching        | _______________ | _____________ | _______ | ___________
Batch processing      | _______________ | _____________ | _______ | ___________
Model routing         | _______________ | _____________ | _______ | ___________
RAG optimization      | _______________ | _____________ | _______ | ___________
Agent controls        | _______________ | _____________ | _______ | ___________
History truncation    | _______________ | _____________ | _______ | ___________
----------------------+----------------+---------------+---------+-------------
Total                 | _______________ | _____________ | _______ |
```

---

## Token counting quick reference

| Content type | Approx. tokens per 1,000 characters |
|---|---|
| English prose | 250 |
| Python code | 350 |
| JSON | 400 |
| XML/HTML | 450 |
| Chinese/Japanese | 500 |
| Markdown | 280 |

| Content type | Approx. tokens per item |
|---|---|
| System prompt (typical) | 200-800 |
| Tool definition (single) | 100-300 |
| Few-shot example (single) | 150-400 |
| RAG chunk (500 chars) | 125 |
| Conversation turn (avg) | 200-500 |

---

## Key formulas

**Cost per request:**

```
cost = (input_tokens / 1M) x input_rate + (output_tokens / 1M) x output_rate
```

**Cache savings per request:**

```
savings = (cached_tokens / 1M) x (input_rate - cached_rate)
```

**Batch savings:**

```
batch_savings = cost_sync x batch_eligible_pct x 0.5
```

**Routing savings:**

```
routing_savings = (expensive_model_cost - cheap_model_cost) x pct_routed_to_cheap
```

**Conversation cost (N turns):**

```
total_input_tokens = sum(system + history_up_to_turn_i for i in 1..N)
```

**Break-even self-hosting volume:**

```
break_even_daily = monthly_selfhost_cost / (managed_cost_per_request x 30)
```

---

## Glossary

**Batch API** — API mode that processes requests asynchronously at a 50% discount.

**Cache hit rate** — Percentage of requests where cached tokens were read from provider cache.

**Cascade** — Model routing pattern that starts with a cheap model and escalates to more expensive models when quality is insufficient.

**Chargeback** — Allocating AI costs to team budgets with financial accountability.

**Context window** — Maximum total tokens (input + output) a model can process per request.

**Extended thinking** — Model feature that generates reasoning tokens before the response.

**Input tokens** — Tokens sent to the model (system prompt, messages, tools, context).

**Output tokens** — Tokens generated by the model in response.

**Prompt caching** — Provider-side cache for repeated prompt prefixes, billed at a discounted rate.

**Quantization** — Reducing model weight precision (FP16 to INT8/INT4) to reduce memory and compute.

**RAG** — Retrieval Augmented Generation: searching a knowledge base and injecting results into the prompt.

**Showback** — Making AI costs visible to teams without financial accountability (precursor to chargeback).

**Token** — The fundamental unit of text processing in LLMs, typically 3-4 characters.

**TTL** — Time To Live: how long cached content persists before expiring.

**Unit economics** — Cost metrics per atomic unit (per request, per user, per feature).