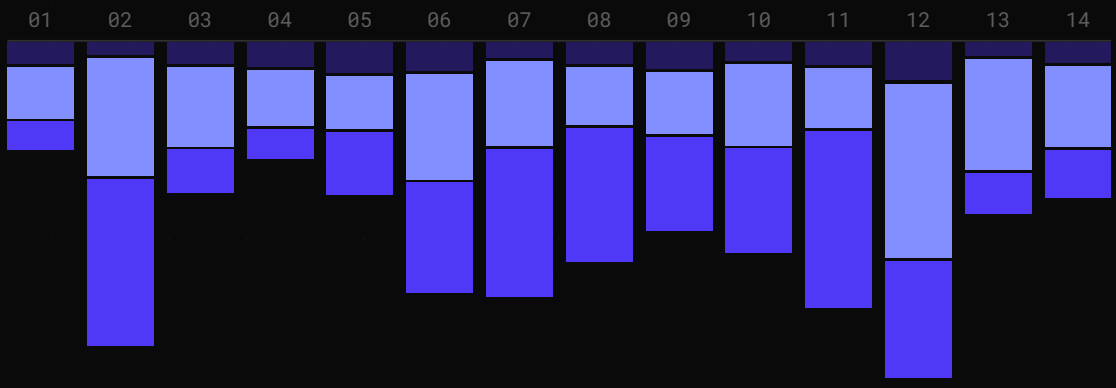
Helmcode

// GUIDE · 14 CHAPTERS · 3 READING DEPTHS

AI Cost Optimization

The Practical Guide

How to reduce LLM and AI infrastructure costs without sacrificing quality.



THE GUIDE, MEASURED

OVERVIEW PRACTICAL DEEP DIVE

EDITION OF 4 NOV 2026

HELMCODE.COM/GUIDES/AI-COST-OPTIMIZATION

// CONTENTS

Contents

00 Introduction
What the guide covers, how to read it and what it does not claim.

PART I · UNDERSTAND THE ECONOMICS

01 The true cost of an AI task 2 min
The price per million tokens tells you less than you think.

02 Build a useful cost baseline 6 min
Tie every model call, tool charge and retry to the feature and task that caused it.

PART II · CHOOSE MODELS AND ROUTING

03 Choose the model the task needs 3 min
Pick a model for each operation by pricing the whole path to an accepted result.

04 Routing, cascades and reasoning budgets 2 min
Send each request to the model it needs, usually with a lookup table rather than another model call.

05 What open-weight models actually change 3 min
Model choice and infrastructure choice are two separate decisions.

PART III · REDUCE REPEATED WORK

06 Tokens, context and prompt caching 5 min
Cut the context a request doesn't use and keep a stable prefix that caching can reuse.

07 Agent memory, response caching and duplicate work 5 min
Keep an agent's working state small and stop paying twice for the same result.

PART IV · MANAGE INFERENCE WORKFLOWS

08 Batch and asynchronous inference 4 min
If the result is needed tomorrow, stop buying a three-second response by default.

09 Agent loops, validation and retries 3 min
Give agents time and spend limits, and match each retry policy to the kind of failure.

10 Tools, search and RAG 4 min
Send the model the passages the answer needs, and count what every extra one costs.

PART V · INFRASTRUCTURE AND PRICING

11 GPU inference and deployment economics 5 min
Size GPU capacity for peak traffic and latency targets before comparing it with a metered API.

PART VI · PUT IT INTO PRACTICE

12 The economics of an AI feature 6 min
An inference call can be cheap and the feature built on top of it can still lose money.

13 The optimization playbook 3 min
Six steps for finding the few changes that will lower your bill the most.

PART VII · PROVIDER NOTE

14 When flat-rate inference makes sense VENDOR 3 min
Provider-specific section. The rest of this guide is vendor-neutral.

REFERENCE

A–Z Glossary
38 terms, each linked to its chapters.

Introduction

Calling an LLM from your application is easy. Knowing what each feature costs you is harder.

Your provider reports which models you used, how many tokens they processed and how much you spent. Whether that spending was needed is left to you. A support agent might search for the same information three times. A document-processing pipeline might extract the same invoice after every retry. You might be using a large reasoning model to return one of five categories.

Each looks cheap on its own and adds up at production volume.

Some optimizations aren't worth making. A cheaper model may need more retries. A shorter prompt may lose the evidence required to answer correctly. Self-hosting can look attractive until you account for idle GPUs and the people operating them.

This guide covers the decisions that change the economics of AI applications: choosing models, reducing unnecessary work, managing context, controlling agents, improving retrieval, and deciding how to buy or operate inference. Where the answer depends on the workload, we explain what you need to know to decide.

To keep the examples connected, the Practical level of each chapter ends with the same fictional company. Ledgerline automates supplier invoices for mid-size businesses and runs three AI operations: invoice extraction, a support assistant and a research agent that writes vendor-risk reports. It starts at $2,600 a month, most chapters change one line of that bill, and the figures carry over from chapter to chapter.

WHAT THE GUIDE COVERS

A production AI bill rarely maps neatly to the feature that generated it. We look at the work behind that bill, from inference and tools to retries, context, infrastructure and subscriptions.

The chapters are grouped by the engineering decision you're making. Read them in order or go straight to a specific problem.

HOW TO READ IT

Each chapter has three reading depths. **Overview** explains what changes and why. **Practical** works through the calculations or implementation patterns. **Deep Dive** covers the mechanics and trade-offs where they matter. Inputs and prices in the examples are illustrative, not measured savings.

WHAT THIS GUIDE DOES NOT CLAIM

There is no universal percentage by which every AI application can cut its costs. Results depend on the workload, its success criteria, its tools and the way inference is purchased. Code examples show patterns to adapt, not production-ready integrations. Check provider pricing and model capabilities against current documentation.

ABOUT THE GUIDE

Written by Helmcode for engineers and teams making decisions about AI models, application workflows and inference infrastructure. The main chapters are provider-neutral. The chapter on flat-rate inference is provider-specific.

Helmcode

// PART I

Understand the economics

01 The true cost of an AI task

OVERVIEW
PRACTICAL
DEEP DIVE

02 Build a useful cost baseline

OVERVIEW
PRACTICAL
DEEP DIVE — Define acceptance before comparing model costs
A calculation that joins the events to the outcomes

The true cost of an AI task

The price per million tokens tells you less than you think.

// LEVEL 1 Overview

Most comparisons start with input and output token prices. Those numbers price a model call, but customers pay for a processed document, a resolved issue or a finished coding task.

A single operation may involve classification, retrieval, two model calls, an external search and a retry. The price of the final model response covers only one of those steps. A successful API response can also hide a failed task: syntactically valid JSON can contain the wrong invoice total.

Measure **cost per accepted task** instead. An accepted task meets the requirements your product sets for that operation.

// LEVEL 2 Practical

Start with:

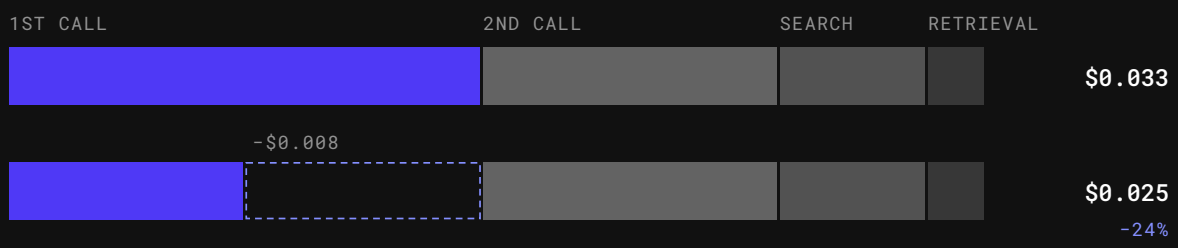
$$\text{Cost per accepted task} = \frac{\text{Total attributable workflow cost}}{\text{Accepted tasks}}$$

Include the costs that apply: inference, embeddings and retrieval, external tools, retries, compute, human correction and a share of fixed capacity. Don’t count a shared platform bill twice, and write down how you allocated it.

A hypothetical document workflow:

EVENT	COST
First model call	$0.016
Second model call	$0.010
Search	$0.005
Retrieval	$0.002
Total	**$0.033**

Halving the first call saves $0.008. The new total is $0.025, about 24% lower, assuming nothing else changes.



One document task, four costs. Halving the first call takes $0.008 off a $0.033 task, which lowers the whole task by 24%.

Run 10,000 tasks at an attributable cost of $330 and accept 9,000, and your cost per accepted task is about $0.0367. An alternative that costs $280 but accepts only 7,000 comes to $0.04 per accepted task. The invoice is smaller, but each accepted task costs more.

The worksheet has a sheet that does this calculation for each of your operations.

You don't need a perfect definition of quality to start. Write down what acceptance means for each operation.

```
// CASE · LEDGERLINE.
```

> Ledgerline's provider bill for October was $2,600. Spread over 72,000 accepted tasks, that is about $0.036 per task, and the average says almost nothing. One extracted invoice costs $0.030, one support answer $0.020 and one vendor-risk report $0.40. Each operation needs its own number before anyone can decide what to change.

// LEVEL 3 Deep Dive

A simple uncached API call costs:

$$C = \frac{\text{input_tokens} \times \text{input_price} + \text{output_tokens} \times \text{output_price}}{1{,}000{,}000}$$

That formula changes when a provider distinguishes cached reads and writes, reasoning tokens, tool fees or batch modes. If reasoning tokens are already included in reported output tokens, don't count them twice.

For fixed infrastructure, separate **marginal** from **allocated** costs. A service costing $1,000 monthly processes 50,000 accepted tasks at an allocated $0.02 per task. While spare capacity remains, one more task that fits into that capacity may cost much less than that at the margin. Once the service is saturated, the next task may require a new capacity tier.

Look beyond averages. A few long-running agents can push the P95 task cost far above the median, so start investigating in the tail.

Build a useful cost baseline

Tie every model call, tool charge and retry to the feature and task that caused it.

// LEVEL 1 Overview

Say one model handles support classification, document extraction and an internal research agent. The provider dashboard shows the model cost $2,600 last month, but probably not which feature spent it or why.

Each feature can have a different problem: oversized prompts in one, repeated paid searches in another, retries in the third. Before switching models, attribute spending to the operation responsible for it.

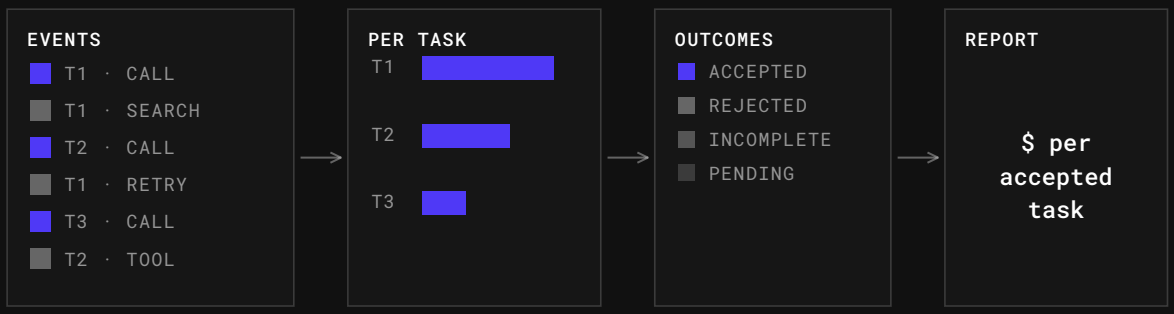
// LEVEL 2 Practical

Give each operation a `task_id` and attach it to every model event, tool invocation and retry. Store the final task outcome separately from individual request outcomes.

A minimal event record needs `task_id`, `operation`, `event_type`, `provider`, `model`, input/output/cached token counts when available, estimated or invoiced cost, attempt number and latency. Add a timestamp and the version of the price table used to compute cost. Metadata is usually enough for cost analysis, so don't store sensitive prompts just to build a dashboard.

The report below counts accepted tasks, so define *accepted* before you count. For extraction, a task might need to pass three checks: the response matches the schema (shape), dates, currencies and totals satisfy business rules (consistency), and the values agree with the source document (correctness). Each

operation needs its own definition, because a support reply or an agent run succeeds in different ways. Record every task as `accepted`, `rejected`, `incomplete` or `pending_review`, so failed and unfinished work stays in the data with its cost.



Every event carries its task ID. Group the events by task, join each task to its outcome, and divide the cost of the whole cohort by the accepted tasks.

From that, produce a short report:

OPERATION	MONTHLY COST	ACCEPTED TASKS	COST PER ACCEPTED TASK
Extraction	$1,200	40,000	$0.030
Support	$600	30,000	$0.020
Research agent	$800	2,000	$0.400

ILLUSTRATIVE DATA. The research agent has the highest unit cost; extraction has the biggest absolute bill. Which to optimize depends on task value, unnecessary work and the feasible saving.

Instead of asking which model is expensive, look for the operation that pays most for retries, keeps resending unchanged context, calls the most external tools or shows the widest spread between ordinary and costly runs.

What you pay is computed from the provider's records, so treat them as the source of truth. Pull usage and cost from the provider usage and cost APIs or billing exports, reconcile them with your event totals every month and investigate any gap. Attribution is easier when spend is separated at the source: use separate API keys or metadata tags per feature, team or environment, so test traffic doesn't land on a production feature's bill. If you already run an LLM observability tool, or emit traces that follow OpenTelemetry's GenAI semantic conventions, send the same `task_id` there instead of building a second pipeline.

Then give the numbers owners. Set budgets and alerts per team. Use showback, where each team sees its own spend, or chargeback, where that spend is billed to its cost center. Keep a simple monthly forecast per operation: expected volume multiplied by the current cost per accepted task. When actual spend drifts from it, check whether volume or unit cost changed.

> // CASE · LEDGERLINE.
>
> The table above is Ledgerline's first report. Adding a `task_id` to every call, search and retry took two days. The result surprised the team: the research agent produces under 3% of accepted tasks (2,000 of 72,000) but 31% of the bill ($800 of $2,600). Most chapters from here on change one of these three lines.

// LEVEL 3 Deep Dive

A simple PostgreSQL event table can support this analysis:

```sql
CREATE TABLE ai_cost_events (
  task_id TEXT NOT NULL,
  operation TEXT NOT NULL,
  event_type TEXT NOT NULL,
  model TEXT,
  input_tokens BIGINT DEFAULT 0,
  output_tokens BIGINT DEFAULT 0,
  cached_tokens BIGINT DEFAULT 0,
  cost_usd NUMERIC(18, 8) NOT NULL,
  attempt INTEGER DEFAULT 1,
  created_at TIMESTAMPTZ NOT NULL DEFAULT now()
);
```

Aggregate events **by task** first, then join to a task-outcomes table. That way one operation with eight calls doesn't count as eight completed tasks. Record excluded fixed costs and incomplete events explicitly, so an incomplete bill is never presented as total cost.

If your task mix changes between months, the average unit cost can move with no change in model efficiency. Compare like with like, or break the metric down by task type and complexity.

Define acceptance before comparing model costs

The word *accepted* needs an operational meaning. Otherwise the denominator becomes a matter of opinion, and the cheapest implementation will often appear to win because it drops the difficult cases.

The three extraction checks need different tools. A schema validator handles **shape**, and application code can handle parts of **consistency**. **Correctness** may need source-grounded comparison or review, because a response can pass the first two checks and still be wrong.

For a documentation assistant, an accepted response might need to answer the question, cite a current authorized source and avoid unsupported claims. For a coding agent, acceptance might require the requested change to be present, no unrelated modifications and the defined acceptance conditions to be satisfied. A classification task may have a deterministic ground-truth label; an open-ended research task may need graded review and explicit unresolved states.

Disclose which of the four task states enter your denominator. For a monthly report, define the cohort and cutoff: tasks created this month, or tasks completed this month? If you mix the two, cost and acceptance rates can disagree only because some runs cross month boundaries.

Report the acceptance rate next to the cost per accepted task. Either number alone can mislead:

OPERATION	STARTED	ACCEPTED	ACCEPTANCE RATE	ATTRIBUTABLE COST	COST / ACCEPTED
Invoice extraction	10,000	9,600	96%	$300	$0.03125
Research synthesis	1,000	850	85%	$425	$0.50

ILLUSTRATIVE SCENARIO. The two operations produce different kinds of outcome, so their unit costs can’t be ranked against each other directly.

A calculation that joins the events to the outcomes

This PostgreSQL query makes the denominator explicit. It takes the tasks **started** in the month as the cohort, so rejected, incomplete and still-pending work keeps its cost in the numerator instead of disappearing from the report. It assumes one row per task with its current state, and that the event table includes all directly attributable variable costs. Shared fixed capacity and human review need their own allocations before this becomes a full TCO report.

```sql
WITH task_cost AS (
  SELECT task_id, SUM(cost_usd) AS cost_usd
  FROM ai_cost_events
  GROUP BY task_id
), cohort AS (
  SELECT t.task_id, t.operation, t.status,
         COALESCE(c.cost_usd, 0) AS cost_usd
  FROM ai_tasks AS t
  LEFT JOIN task_cost AS c USING (task_id)
  WHERE t.created_at >= DATE '2026-10-01'
    AND t.created_at < DATE '2026-11-01'
)
SELECT operation,
       COUNT(*) AS started_tasks,
       COUNT(*) FILTER (WHERE status = 'accepted') AS accepted_tasks,
       COUNT(*) FILTER (WHERE status = 'pending_review') AS pending_tasks,
       SUM(cost_usd) AS variable_cost_usd,
       SUM(cost_usd) / NULLIF(
         COUNT(*) FILTER (WHERE status = 'accepted'), 0
       ) AS cost_per_accepted_task
FROM cohort
GROUP BY operation;
```

The query requires an `ai_tasks` table with `task_id`, `operation`, `created_at` and `status` (`accepted`, `rejected`, `incomplete` or `pending_review`). Treat the month as provisional while `pending_tasks` is still significant: each pending task that ends up accepted lowers the cost per accepted task. In real accounting, keep usage events with a zero or unknown cost distinct from events confirmed as free. Reconcile the sums against invoices, because pricing calculations can miss cached-token discounts, tool charges or delayed usage records.

Document the acceptance definition and the accounting exclusions next to the report, so another engineer can reproduce the numbers.

The table definitions and the monthly query are also in a SQL file you can download, with the task table added.

Helmcode

// PART II

Choose models and routing

03 Choose the model the task needs
OVERVIEW
PRACTICAL
DEEP DIVE

04 Routing, cascades and reasoning budgets
OVERVIEW
PRACTICAL
DEEP DIVE

05 What open-weight models actually change
OVERVIEW
PRACTICAL
DEEP DIVE The cost of switching later

Choose the model the task needs

Pick a model for each operation by pricing the whole path to an accepted result.

// LEVEL 1 Overview

Teams often pick a capable model for the first feature and use it everywhere afterward. Months later, that model is classifying tickets, summarizing conversations, extracting fields and investigating production incidents.

The setup is easy to understand, but some operations end up paying for capabilities they don't need.

A five-category ticket classifier and an agent debugging an intermittent payment failure are different problems, and they rarely need the same model and configuration. For each operation, price the whole path to an accepted result, including validation, retries and escalation, before you pick a model by its token price.

// LEVEL 2 Practical

Group work by what it requires: constrained classification, extraction, summarization, grounded Q&A, multi-document reasoning and tool-using agents. For each group, record the acceptance criteria and the non-negotiable constraints: latency, context size, output format, language, availability and data handling.

Then shortlist models that can meet those requirements, including open-weight candidates where appropriate. An overall benchmark ranking doesn't tell you whether a model suits the task.

To find out, build a small evaluation set from real cases of that operation, including tasks that were accepted and tasks that were rejected. Run each candidate against it before switching, and again after any prompt or model change. Those runs cost tokens and review time, so count them as part of the decision. Look at results by case type and failure mode as well as the overall score, because a model can match the average while failing the cases that matter most.

A hypothetical comparison of two approaches to the same extraction operation:

	LARGER MODEL	SMALLER MODEL + ESCALATION
Initial call	$0.020	$0.006
Validation (same checks on both)	$0.001	$0.001
Escalation	None	15% × $0.020
Modeled cost per request	$0.021	$0.010

Both columns include validation because either model can return a wrong answer. The comparison also assumes equal acceptance rates; if they differ, compare cost per accepted task instead. Whether the cheaper column holds up depends on how reliably the check catches failures, which Routing, cascades and reasoning budgets covers.

Before adding a different model, check whether ordinary code can do the job. Formatting known fields, comparing timestamps and arithmetic usually don't need generative inference.

> // CASE · LEDGERLINE.
>
> Ledgerline built an evaluation set of 300 real invoices, 200 accepted and 100 previously rejected, and ran two candidates against it. Each run of the set costs about $9. A smaller model with 15% escalation to the large one matched the large model's acceptance on the set and brought extraction to $0.018 per accepted invoice: 40,000 × $0.018 = $720 a month, down from $1,200.

// LEVEL 3 Deep Dive

Compare billable token usage as well as published unit prices. Two models can tokenize the same input differently, need different levels of prompt detail, produce outputs of different length and consume different reasoning budgets.

A larger context window doesn't guarantee better use of a long document; retrieval quality and prompt structure may matter more than the headline limit. Every additional model also adds maintenance: provider configurations, prompt variants, fallback behavior and monitoring.

Fine-tuning or distillation is another lever. On a narrow, stable task, such as one extraction schema that rarely changes, a smaller model trained on your accepted outputs or on a larger model's answers can replace a large general one. It brings its own costs: preparing training data, training runs, evaluation,

hosting and retraining when inputs or requirements change. The licenses of the models involved must also allow it, which What open-weight models actually change covers. Spread those costs over the volume you expect before the next retraining and add them to the comparison.

Keep the number of models small, and add one for a distinct workload only when the benefit justifies that maintenance.

Routing, cascades and reasoning budgets

Send each request to the model it needs, usually with a lookup table rather than another model call.

// LEVEL 1 Overview

Routing and cascading are related techniques. **Routing** chooses a model for a request. **A cascade** starts with one model and escalates after a detectable failure or insufficient result.

If your application has separate endpoints for classification and complex analysis, it probably already knows which model each one needs. A deterministic lookup is simpler than asking an LLM to route another LLM.

Dynamic routing makes sense when one endpoint receives requests with very different requirements. Even then, the two kinds of mistake cost different amounts: an easy task sent to a costly model wastes money, while a difficult task sent to an unsuitable one may fail the whole operation.

// LEVEL 2 Practical

Start with static policies:

PYTHON

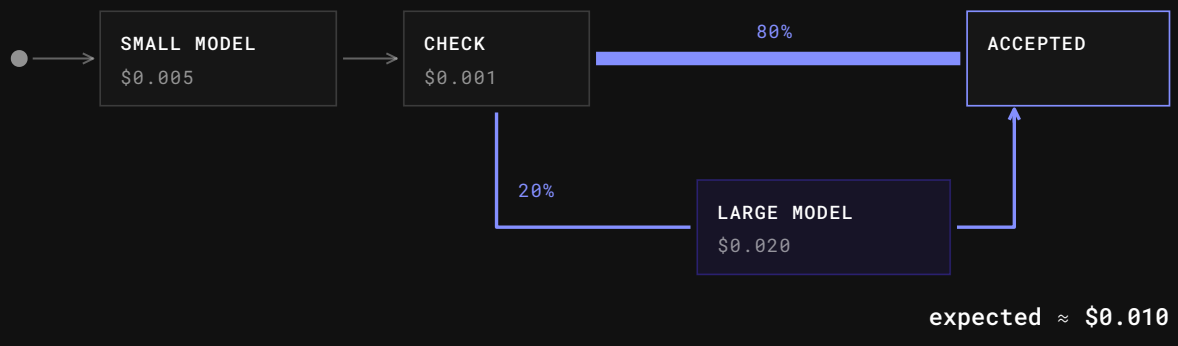
```python
MODEL_BY_OPERATION = {
    "classification": "small-model",
    "extraction": "extraction-model",
    "incident_analysis": "reasoning-model",
}

def choose_model(operation: str) -> str:
    return MODEL_BY_OPERATION.get(operation, "default-model")
```

When a cascade is justified, its simplified expected cost is:

$$C_{cascade} = C_{small} + p_{escalation} \times C_{large} + C_{validation} + C_{other_recovery}$$

For example, a $0.005 initial call, 20% escalation to a $0.020 model and $0.001 validation produces an expected $0.010 before other recovery work. That figure depends on your escalation signal and acceptance requirements.



Every request pays for the small model and the check; only the 20% that fails the check also pays for the large one. Expected cost: about $0.010.

Some reasoning models expose effort or thinking controls, which can cut unnecessary internal generation on constrained tasks without switching models. Supported values, defaults and billing vary by model and provider; don't assume a generic `none`, `low` or `high` setting works everywhere.

Set reasoning effort per operation, next to the model choice. A basic classifier may need little extended reasoning, while a difficult code investigation may justify more. Don't use a low output-token limit as a stand-in for low effort: it can truncate the final answer after internal reasoning has already consumed the budget.

// CASE · LEDGERLINE.

Six in ten Ledgerline support questions ask where an invoice is in the approval flow. The application already knows that from its own API, so those 18,000 questions now go through a lookup and a small model with low reasoning effort at $0.010 each. The other 12,000 keep the current setup at $0.020. Support falls from $600 to 18,000 × $0.010 + 12,000 × $0.020 = $420 a month, with no router model.

// LEVEL 3 Deep Dive

Dynamic routing adds classification cost and latency. If the router costs $0.001 and avoids $0.003 of downstream inference on average, the gross difference is $0.002 per request, before wrong routes and maintenance. At small scale, the architecture may not be worth it.

Cascades depend on a **useful escalation signal**. Schema checks and business-rule violations are cheap to detect; semantic correctness is harder. A model that reports high confidence can still be wrong, and asking another model to judge every answer may erase part of the saving.

Reasoning-effort controls affect the whole workflow. With less reasoning, an agent may make more tool calls or repeat work. On a metered API, fewer billable reasoning tokens can lower a call's charge; on dedicated capacity, the immediate benefit is more likely freed throughput.

What open-weight models actually change

Model choice and infrastructure choice are two separate decisions.

// LEVEL 1 Overview

When someone suggests replacing a proprietary model with Qwen, Gemma, DeepSeek or GLM, the conversation often moves straight to GPUs. But choosing an open-weight model doesn't mean you have to run it yourself.

You can use a metered API, buy dedicated capacity from a provider, or operate the weights on your own infrastructure where the license permits it. This matters for cost, because the organization that develops the model doesn't have to be the one serving every request.

Open weights are free to download, not free to run. Each of those options comes with an inference bill, paid in tokens, committed capacity or GPUs. Nor is an open-weight model necessarily "open source" in the broader sense. Publishing the weights doesn't imply publishing training data, a reproducible training pipeline or unrestricted use rights.

// LEVEL 2 Practical

First establish that the specific model meets the task requirements, then compare ways to deploy it.

OPTION	WHAT YOU PAY FOR	MAIN TRADE-OFF
Metered API	Usage, often input and output tokens	Simple operation; bill follows usage
Dedicated capacity (managed)	Committed service or capacity	More predictable cost; plan and rate limits
Self-hosted	GPUs, supporting systems, operators	Greater control; idle capacity and responsibility

Review licensing per model **release**, not per family: Qwen, Gemma, DeepSeek and GLM have each issued releases with different terms or dependencies. A repository's code license may not cover the weights, and distilled variants may inherit conditions from their base models. Check the exact model card and license text before deploying commercially.

Weights developed in one country can run on infrastructure operated elsewhere, so a model's origin tells you little about where your data is processed. For data handling, look at who operates the service, the region, logging, external tools and the contract. An OpenAI-compatible API doesn't guarantee the same data path or feature behavior across providers.

> // CASE · LEDGERLINE.
>
> For extraction, Ledgerline's shortlist included an open-weight model behind a managed metered API in the EU. Legal read the license of that exact release, the security team checked where the provider processes data, and the evaluation set from chapter 3 decided between it and the proprietary small model. Nobody bought a GPU.

// LEVEL 3 Deep Dive

Parameter counts map poorly to serving cost; GPU inference and deployment economics covers memory sizing for dense and mixture-of-experts models.

Two services exposing the same weight release can still differ in quantization, tokenization or chat templates, tool parsers, reasoning configuration, throughput and availability. Changing a base URL may reduce migration work, but you still have to validate and operate the new setup.

The cost of switching later

Open weights and OpenAI-compatible APIs make it easier to leave a provider that raises prices, retires a model or changes terms. The move still takes work:

- Re-run your evaluation set against the new setup. Choose the model the task needs covers building one.
- Adjust prompts and tool schemas when the new service uses a different chat template, tool parser or structured-output support.
- Re-embed stored documents if the embedding model changes, because vectors from different models can't be compared with each other.
- Export your data, logs and any fine-tuned weights, and budget for egress charges on the transfer.
- Read the contract. Minimum commitments, prepaid credits and notice periods can make leaving expensive before any technical work starts.

You can lower this cost from the start. Keep prompts, the evaluation set and request logging in your own systems, in formats that don’t depend on one provider. Keep a short record of the features you use that only one provider offers, such as a particular caching mode or batch interface, so a migration plan knows what it has to replace.

OFFICIAL MODEL-RELEASE REFERENCES:

Qwen3, Gemma documentation, DeepSeek-R1, DeepSeek-V3, GLM-4.5, GLM-5. Each release carries its own license terms.

Helmcode

// PART III

Reduce repeated work

06 Tokens, context and prompt caching

OVERVIEW
PRACTICAL Images, audio and documents
DEEP DIVE A stable prompt boundary in practice

07 Agent memory, response caching and duplicate work

OVERVIEW
PRACTICAL
DEEP DIVE How transcript growth adds up
One operation, one in-flight result

Tokens, context and prompt caching

Cut the context a request doesn't use and keep a stable prefix that caching can reuse.

// LEVEL 1 Overview

Prompts grow for understandable reasons: a new example fixes a corner case, another instruction improves consistency, and at some point the application starts including the entire conversation to preserve continuity. Eventually every request carries material nobody can clearly justify.

Start by asking why each piece of information is there. Some of it defines the task, some is evidence, and some is repeated because the application never learned to leave it out. Cutting irrelevant context can lower cost and make the model's job clearer, as long as the evidence the answer depends on stays in.

Output counts too. If the product needs a category and priority, an essay explaining the choice may be waste. Specify the exact output format, since "be concise" is too vague.

// LEVEL 2 Practical

Inspect the **actual request** sent to the model: system instructions, examples, tool definitions, retrieved passages and conversation history, not just the visible user message.

For a support assistant, the current question, applicable policy and relevant account facts may be enough, without years of history. For extraction, prefer an explicit schema to free-form explanations the product discards. Keep exceptions, original identifiers and any evidence needed for correctness.

A cost example with hypothetical prices of $1 per million input tokens and $4 per million output tokens:

	BEFORE	AFTER
Input tokens	10,000	6,000
Output tokens	1,000	400
Cost per call	$0.0140	$0.0076

That's a modeled 45.7% reduction **for the call**, assuming both results are equally acceptable and all input uses uncached pricing.

Prompt caching complicates this. When many requests share a long stable prefix, a provider may process and charge for that portion differently, and a shorter prompt with dynamic data near the beginning can lose that benefit. Put reusable instructions and reference material before request-specific content when that matches the provider's caching behavior.

Images, audio and documents

Non-text input follows its own billing rules. Images are usually converted to tokens based on resolution, often by splitting them into fixed-size tiles, so a larger image costs more even when the extra detail doesn't help the task. Audio is typically counted by duration or converted to tokens. PDFs and other documents are often processed page by page, sometimes as both extracted text and a rendered image of each page.

A few habits keep these inputs in check:

- Send the resolution the task needs. Downscale screenshots and photos, and crop to the region that matters, such as one table or one form field.
- Avoid sending the same image or page again on every turn. Conversation history carries attachments forward, so store the first result (a description, extracted fields or a transcript) and reference it in later turns.
- When the task only needs the text, extract it with a cheap OCR tool or a document parser and send that text to the model.
- Check how the provider counts each modality before estimating. Tokens per image or per page vary by provider and model, and an estimate based on text alone can be far off.

> // CASE · LEDGERLINE.
>
> Every Ledgerline support request carried a 6,000-token block of policy text, with the customer's name and a timestamp at the top. Moving the request-specific fields to the end made the policy block a stable prefix that the provider caches, and the answer format went from free prose to a short structured reply. Support drops from $420 to $330 a month.

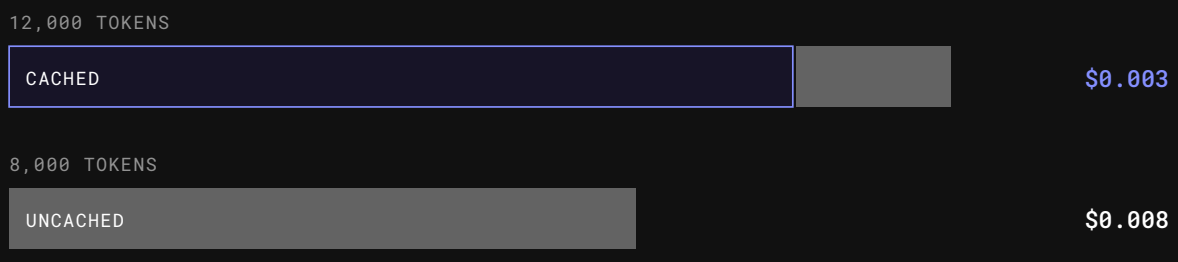
// LEVEL 3 Deep Dive

Three mechanisms are easy to confuse:

MECHANISM	REUSES	WHAT IT SAVES
Provider prompt caching	Repeated input prefix	Eligible input charges and/or processing
Serving-engine prefix caching	Stored KV cache of a shared prefix	Some prefill computation
Response caching	A valid prior output	Potentially the entire inference call

With prompt or prefix caching, the model generally still has to generate a new answer. It helps most when long shared inputs recur often. There is no universal discount. Cache writes, reads, TTLs, minimum sizes and eligibility vary by provider and model.

For illustration: 10,000 cached input tokens at $0.10/M plus 2,000 uncached at $1/M cost $0.003, while a rewritten 8,000-token uncached prompt costs $0.008. In this scenario the shorter prompt costs more.



A longer prompt can cost less when most of it is a cached prefix. Hypothetical rates: $0.10 per million cached input tokens, $1 per million uncached.

Serving-engine prefix caching works one layer down. Every request builds a KV cache, the attention state for the tokens processed so far, and every decode step reads it. Prefix caching, as in vLLM, keeps the KV cache of a shared prefix and reuses it for later requests that start with the same tokens. That saves *prefill* work for the prefix, but the model still generates every output token. On dedicated infrastructure, the benefit may be higher throughput rather than a smaller monthly invoice.

A stable prompt boundary in practice

The simplified Python pattern below targets an API that supports ordinary system and user messages. Shared material stays stable and request-specific content goes last. The code does **not** guarantee a cache hit, which depends on the serving provider, the model, minimum eligible length and prior requests.

PYTHON

```python
SYSTEM = (
    "You answer questions about our public API. "
    "Use only the supplied reference material. "
    "If the source does not answer the question, say what is missing."
)

# Versioned and stable across requests until the documentation changes.
DOCS_VERSION = "api-docs-v17"
REFERENCE = load_documentation(DOCS_VERSION)

def messages_for(question: str) -> list[dict[str, str]]:
    return [
        {"role": "system", "content": SYSTEM},
        {"role": "user", "content": f"Reference ({DOCS_VERSION}):\n{REFERENCE}"},
        {"role": "user", "content": f"Question:\n{question}"},
    ]
```

In a real application, `load_documentation` stands for your own versioned document loader. Don't insert a timestamp or random request ID before the reference material, because it changes the beginning of the request every time. Keep tool definitions and their ordering stable too when the provider includes them in the cacheable prefix.

Check the cached-token fields the API reports before assuming savings. When a service charges for cache writes or storage, work out the effective cost of the complete reuse cycle instead of multiplying a headline cache discount by all input tokens. A stable prefix is worth having even without caching, because it makes prompt changes deliberate and reviewable. Any cache hits are a bonus on top.

REFERENCES:

OpenAI prompt caching, Gemini caching, vLLM prefix caching.

Agent memory, response caching and duplicate work

Keep an agent's working state small and stop paying twice for the same result.

// LEVEL 1 Overview

A coding agent reads files, checks logs and tries approaches that fail. An hour in, its transcript may hold thousands of lines of output. The next step rarely needs all of them, only the objective, the constraints, confirmed findings, failed approaches and where the evidence lives. That smaller set is the agent's working state.

Context compaction, response caching and idempotency each avoid a different kind of repeated work, so treat them as separate problems.

// LEVEL 2 Practical

Maintain three layers: recent conversation for continuity, durable **working state** for decisions and constraints, and evidence stored elsewhere with retrievable references. A compact record might look like this:

JSON

```
{
  "objective": "Fix the webhook retry bug",
  "constraints": ["No duplicate charges", "Rollback required"],
  "findings": ["Worker may time out after downstream success"],
  "failed_approaches": ["Blind retry duplicates the action"],
  "evidence": ["src/webhooks.py", "incident-1042"],
  "next_step": "Check downstream idempotency support"
}
```

Keep exact user constraints and authoritative business state outside lossy prose summaries. Compaction costs tokens and may break prompt-prefix reuse, so compact when the expected reduction in active context justifies it, not automatically after every turn.

For completed operations that can safely produce the same answer, use **exact response caching** keyed by the relevant inputs and versions. A document extraction key might include content hash, schema version, preprocessing version, model configuration and tenant scope. Enforce authorization separately, since a cache key doesn't control access.

When two identical requests arrive at the same time, a finished-result cache alone won't help, because both may miss and start inference. Use in-flight deduplication or a shared operation record so callers wait on the same work.

Measure duplicate work before building a cache. Log a request hash on every model call, computed from the inputs a cache key would use. Each day, report per operation the share of calls that repeat an earlier hash and what those calls cost. Count identical requests that overlapped in time separately, since only in-flight deduplication removes those. That report tells you whether exact caching is worth building.

ILLUSTRATIVE: AN EXTRACTION OPERATION MAKES 40,000 CALLS A DAY, AND 6,000 REPEAT A HASH SEEN EARLIER THAT DAY: 6,000 / 40,000 = 15%. AT $0.004 PER CALL, THE REPEATS COST 6,000 × $0.004 = $24 A DAY, OR $720 OVER 30 DAYS.

> // CASE · LEDGERLINE.
>
> About 10% of invoices reaching Ledgerline are the same file uploaded twice, by the supplier and by the customer. A content hash plus the schema and model version now keys an exact cache, scoped to the customer account, so 4,000 of the 40,000 monthly invoices skip inference. Extraction goes from $720 to $648.

// LEVEL 3 Deep Dive

Response caching asks, "Do we have a still-valid result for these inputs?" **Idempotency** asks, "Has this particular intended operation already been submitted?" An agent that retries sending an email after losing a tool response must not send two emails just because the text is identical. Preventing that takes a durable operation identity.

A distributed lock doesn't prevent duplicate charges either. If a worker crashes or times out after sending a request, the provider may still run it, and the retry runs it again. You can guarantee that only one result is committed and still pay for two upstream calls.

Semantic caching is riskier than exact caching. "Can I cancel?" and "Can I cancel and get a refund?" are similar questions that need different answers. Similarity alone doesn't establish equivalence, freshness or authorization. Start with exact reuse and deduplicated execution; add semantic reuse only when you can define safe boundaries.

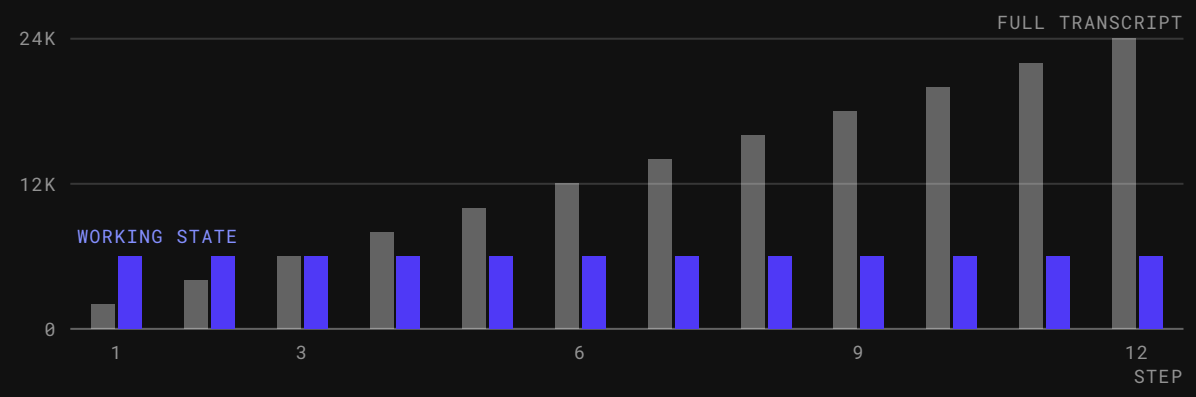
How transcript growth adds up

An agent that resends its whole transcript pays for every earlier step again on each new one. If each step adds `t` tokens, step `k` sends about `k × t` input tokens, and a run of `n` steps sends `t × (1 + 2 + … + n) = t × n(n + 1) / 2`. Total input grows roughly with the square of the number of steps. An agent that sends a capped working state plus the latest step sends about the same amount each time, so its total grows linearly.

ILLUSTRATIVE: EACH STEP ADDS 2,000 TOKENS. THE COMPACT AGENT SENDS A 4,000-TOKEN WORKING STATE PLUS THE LATEST 2,000-TOKEN STEP, 6,000 TOKENS PER STEP.

STEPS (`N`)	FULL TRANSCRIPT: 2,000 × N(N + 1) / 2	COMPACT STATE: 6,000 × N
25	650,000	150,000
50	2,550,000	300,000
100	10,100,000	600,000

Doubling the run from 50 to 100 steps multiplies full-transcript input by about four and compact input by two. The compact column leaves out the tokens spent updating the working state. Prompt caching lowers the price of the repeated prefix (Tokens, context and prompt caching), but the token count grows the same way.



Input tokens sent at each step when every step adds 2,000 tokens. Resending the transcript grows without limit; a compact working state stays at about 6,000.

One operation, one in-flight result

A cache that checks `get(key)` and runs inference on a miss lets two simultaneous requests launch two calls. Coordination has to happen **before** the expensive work starts.

A durable task record can have these fields:

```text
(tenant_id, idempotency_key)  UNIQUE
request_hash                  required
status                        pending | running | completed | failed
result_reference              nullable
lease_owner, lease_expires_at  nullable
```

On arrival, atomically create or retrieve the record. If the key exists with a different `request_hash`, reject the mismatch. If it is completed, return its result; if it is running, return the task identifier or wait for completion. A lease allows controlled recovery when a worker disappears. Publish the final result only if the worker still owns the lease; a database transaction or fencing token can protect that update.

This protects the application's state, but it is **not** an exactly-once guarantee for the inference provider. If the worker loses its lease after dispatching a remote request, a recovery worker may dispatch another one, so record both attempts if they incur charges. For tools that create a payment, send an email or change a remote record, use the downstream system's idempotency contract wherever one exists.

Keep this separate from the finished-result cache. A document hash plus extraction-schema version may justify reusing a result across *separate* legitimate operations, while an idempotency key identifies retries of the *same* intended operation. Mixing the two causes mistakes when a user deliberately asks for fresh processing.

REFERENCES:

AWS: idempotent APIs, Redis: distributed locks, OpenAI conversation state.

Helmcode

// PART IV

Manage inference workflows

08 Batch and asynchronous inference

OVERVIEW
PRACTICAL
DEEP DIVE Running provider batch jobs

09 Agent loops, validation and retries

OVERVIEW
PRACTICAL
DEEP DIVE A budgeted run, without pretending estimates are invoices

10 Tools, search and RAG

OVERVIEW
PRACTICAL
DEEP DIVE Embedding costs

Batch and asynchronous inference

If the result is needed tomorrow, stop buying a three-second response by default.

// LEVEL 1 Overview

A feedback-analysis pipeline classifies thousands of comments every night. Nobody reads the report until morning, yet the application may use the same real-time API mode as a customer-facing assistant.

Some providers discount deferred inference, but keep three things apart: **asynchronous execution**, **provider Batch APIs** and **GPU-level inference batching**.

A background worker calling the standard API is asynchronous to your user but may still pay standard token prices. A provider Batch API can offer a lower pricing tier for nonurgent requests. Continuous batching inside a serving engine improves throughput and has nothing to do with API discounts.

// LEVEL 2 Practical

Start with the product's deadline. Interactive support often needs seconds; historical document processing may tolerate hours. Dependent agent steps limit what you can defer, because a downstream action can't be submitted until the previous step produces its inputs.

Google, OpenAI and Anthropic each document a discounted batch mode for requests that can wait, with results returned within a turnaround window. As of this edition, Google lists a 50% price reduction for eligible Gemini Batch API use and a 24-hour turnaround target. Discounts, windows and eligible models differ by provider and change over time, so check the current terms for each model and endpoint before relying on them.

At a hypothetical standard price of $0.012 per task, 500,000 tasks cost $6,000. An eligible batch mode with equivalent output at half price costs $3,000 in inference charges. Net savings can fall short of $3,000 once you count job management, failed or expired requests, resubmissions and any work that still needs immediate execution.

Track every batch item on its own, so a partly failed job can be repaired without resubmitting the whole batch. The Deep Dive below covers the details.

> // CASE · LEDGERLINE.
>
> A quarter of Ledgerline's invoices arrive in bulk monthly imports that nobody reads until the next morning. Those now go through the provider's batch mode at half price, with each invoice tracked by its own ID. The saving is 25% of $648 at half price, about $81, and extraction settles at $567 a month.

// LEVEL 3 Deep Dive

For self-hosted inference, batching is a throughput technique: the serving engine interleaves requests of different lengths so the GPU stays busy. The gain depends on the workload, KV-cache pressure and latency budget, and GPU inference and deployment economics covers the mechanics.

Better throughput lowers the cost allocated to each task, but **the capacity bill stays the same** unless the change lets you reduce or avoid capacity.

Batching also combines with caching and deduplication. If only 5% of yesterday's documents changed, skipping the unchanged ones may save more than the batch pricing discount.

Running provider batch jobs

Before submitting, write each item to your own store with its application ID, a hash of the input, the model and prompt version, the provider's job ID and a status such as `pending`, `submitted`, `succeeded`, `failed` or `expired`. Put the application ID in the request's custom identifier field when the API offers one, and use it to join results back to your records. A result file may come back in a different order from the input, and some items may be missing from it.

A finished job can contain successful items, items that returned an error and items that expired before the provider processed them. Handle each one on its own. A malformed request or an input over the context limit will fail again unchanged, so fix it or drop it. Expired items, rate-limit errors and transient server errors are worth resubmitting.

Put the retryable items in a new job that carries the original application IDs, and leave the successful ones alone. Cap the attempts per item, for example two resubmissions, then move anything still failing to a review queue. Without a cap, a systematic fault such as a prompt template that always exceeds the context limit keeps generating new jobs.

Make downstream processing idempotent. A worker can crash after reading a result file and read it again, and an item you resubmitted after a timeout can still produce a result from the first attempt. Write results with an upsert keyed on the application ID and prompt version, and have consumers skip IDs they have already handled. A duplicate result then costs tokens but doesn't create a second ticket, email or database row.

Count everything the job cost. Include charges for items whose output you discarded, charges for every resubmission and any items you moved to the real-time API at standard prices. Divide that total by the number of items that produced a usable result. That figure is the one to compare with real-time processing, and with a high failure rate it can sit well above the headline batch price.

Decide what happens to items that miss the deadline before the first job runs. If the morning report is due at 08:00 and the provider's window allows up to a day, a job submitted at midnight may not finish in time. The options are to send late items to the real-time API, publish the report with those items marked as pending, or skip them when the data will be stale anyway. Each option has a different cost, so the product owner should pick one and the pipeline should implement it explicitly.

REFERENCES:

Gemini Batch API, Gemini optimization modes, OpenAI Batch API, Anthropic batch processing.

Agent loops, validation and retries

Give agents time and spend limits, and match each retry policy to the kind of failure.

// LEVEL 1 Overview

An agent's initial request doesn't tell you how much it will cost. It may inspect a repository, call several tools, revisit hypotheses and attempt a fix. Some tasks need that work. Others get stuck repeating the same search or repairing the same invalid response.

A five-step limit on every agent is too blunt to control cost. Ten short calls to a small model cost far less than ten long calls to an expensive reasoning model. You need limits on time and spend, plus a way to spot unproductive work.

Failures also differ. Temporary provider overload, invalid JSON and incorrect extracted facts are separate problems, and retrying all three with the same prompt is rarely the best policy.

// LEVEL 2 Practical

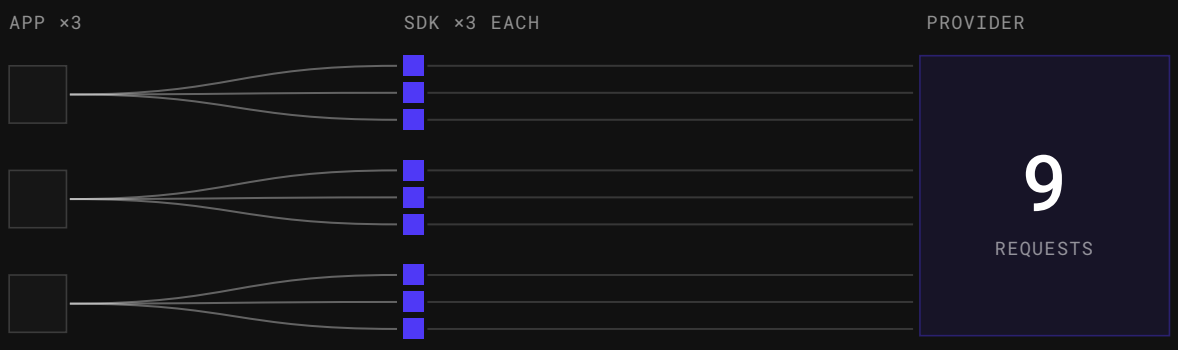
Budget agents by the operation they perform. Track model calls, external tool charges, wall-clock time and an estimated monetary total. Enforce limits in application code, not by asking the model to "spend less than $1." If the budget must be strict, reserve estimated cost before dispatching concurrent work, then reconcile actual usage.

Define what the agent returns when it stops. At the budget limit, it should report confirmed findings, changes made, remaining work and the reason it stopped. A partial result you can trace is worth more than a generic failure that throws the progress away.

Match the recovery policy to the failure:

FAILURE	BETTER FIRST RESPONSE
Transient overload or network error	Bounded retries, backoff and jitter
Invalid schema	Enforce supported structured output and validate deterministically
Missing evidence	Retrieve or provide the missing source
Incorrect semantic interpretation	Reassess model, input and verification approach
Unsupported parameter or exhausted credit	Fix configuration or billing, don't blindly retry

Watch for retry amplification. If an SDK makes up to three attempts and your application wraps it in three attempts, the upstream service may see nine requests. Set an overall deadline and know the retry behavior of each layer.



Three application attempts around an SDK that also retries three times can reach the provider nine times for one task.

// CASE · LEDGERLINE.

Ledgerline's research agent averaged $0.40 per accepted report, but its P95 run cost several dollars: one run in twenty kept searching for filings that didn't exist. A per-run ceiling of $1.00, a check for repeated searches and a stop state that returns the findings so far brought the research line from $800 to $640 a month, with the same number of accepted reports.

// LEVEL 3 Deep Dive

Validation has a cost of its own. A JSON parser is cheap. A second LLM reviewing every response can add substantially to inference spending and may still miss semantic errors. Use deterministic checks where possible: schema constraints, known status codes, arithmetic, identifiers and authoritative data.

For a simple bounded retry model, where each attempt costs `C`, succeeds independently with probability `p > 0` and permits `N` attempts:

$$\text{Expected attempts} = \frac{1 - (1-p)^N}{p}$$

$$\text{Expected cost} = C \times \text{Expected attempts}$$

With `p = 0.8` and `N = 3`, that is 1.24 attempts, so the call costs about 24% more than `C` on average. With `p = 0.4`, it is 1.96 attempts, and 22% of tasks still fail after the third try.

Real retries rarely behave this neatly. Model errors often correlate across attempts, attempts can cost different amounts, and some failures need tool or human recovery. For agents, keep a record of failed approaches so the agent doesn't repeat them after context compaction.

Measure progress by changes in state, such as new evidence, eliminated hypotheses, completed work or a different approach. An agent's own claim that it is making progress doesn't count.

A budgeted run, without pretending estimates are invoices

Keep the agent budget outside the model. The minimal pattern below makes a **reservation** before launching a call. It assumes a single worker updates the local state; production concurrency needs an atomic database reservation or another synchronization mechanism.

PYTHON

```python
from dataclasses import dataclass
from decimal import Decimal

@dataclass
class RunBudget:
    ceiling: Decimal
    reserved: Decimal = Decimal("0")
    actual: Decimal = Decimal("0")
    model_calls: int = 0
    tool_calls: int = 0

    def reserve(self, predicted: Decimal) -> None:
        if self.actual + self.reserved + predicted > self.ceiling:
            raise RuntimeError("Task budget exhausted")
        self.reserved += predicted

    def reconcile(self, predicted: Decimal, charged: Decimal) -> None:
        self.reserved -= predicted
        self.actual += charged
```

The reservation only decides whether a call may start; it doesn't cap the bill. The actual charge can exceed the estimate, providers can bill a timed-out request, and several workers could race unless the reservation is atomic. To limit exposure, bound tokens and tools where supported, set deadlines, prevent unauthorized concurrency, and define what happens at the ceiling.

Paid search, browsing and document-processing calls go into the same budget as model calls.

REFERENCES:

AWS: backoff and jitter, OpenAI rate limits, OpenAI structured outputs.

Tools, search and RAG

Send the model the passages the answer needs, and count what every extra one costs.

// LEVEL 1 Overview

A documentation assistant receives a question, searches a vector database and sends twenty passages to the model. The first two contain the answer; the rest are related but unnecessary. The search may be cheap, but the model still processes every returned token.

RAG costs span ingestion, embeddings, storage, retrieval, reranking, added prompt context and the final model response. Paid web search, OCR and other tools can add separate charges. Optimizing only the vector query or only the generation model misses how the two interact.

// LEVEL 2 Practical

Account for query-time and ingestion-time costs separately. A mostly static knowledge base receiving millions of questions has different economics from a frequently updated archive receiving few queries.

Don't treat `top_k`, the number of retrieved passages sent to the model, as a universal quality knob. At 500 tokens per passage, twenty passages add roughly 10,000 input tokens; five add 2,500. At $2 per million uncached input tokens, the difference is $0.015 per request, or $1,500 across 100,000 requests. If the answer depends on passage eight, though, cutting blindly to five is a false economy.

Another option is to retrieve broadly and then filter with reliable metadata, hybrid search or reranking before sending evidence to the generation model. Reranking has its own cost and pays off only if it cuts enough context or improves quality enough.

Not every task needs retrieval. Reformatting the answer you just produced probably doesn't require another vector search, and a known order status may be better obtained directly from the authoritative application API than inferred from documents.

To set `top_k` from evidence, log two things per request: the passages retrieved, with their rank, and the passages the answer actually used. Citations in the answer give you the second list cheaply; for answers without citations, review a sample by hand or with a grader. From those logs, track the rank of the deepest passage each answer needed, grouped by query type. If policy questions almost always draw on the top

three passages while troubleshooting questions often reach rank eight or nine, give each type its own `top_k` instead of sending ten passages to everything. Recheck the numbers after any change to chunking, the embedding model or the reranker, because rank positions shift.

> // CASE · LEDGERLINE.
>
> The research agent sent 20 retrieved passages with every question. Logging which passages the reports actually cited showed that most query types never used anything past the eighth. With `top_k` set per query type, and the broad ones still allowed 20, research falls from $640 to $560 a month.

// LEVEL 3 Deep Dive

Chunk size affects answer quality as well as cost. Tiny chunks can separate a definition from its exception, and large ones can pull in irrelevant sections. Preserve headings, tables, code blocks and document versions where those boundaries matter.

If the answer is wrong, inspect what was retrieved before upgrading the generation model. A larger model can't reliably reason from a policy paragraph that never reached it. Poor retrieval also tends to trigger extra searches and expensive retries.

A small, stable reference set reused by every request may be cheaper as cached prompt context (Tokens, context and prompt caching) than as a separate retrieval workflow. A large or frequently updated collection usually benefits from selective retrieval.

Embedding costs

Embeddings are paid at two moments. Ingestion embeds every chunk of the corpus; query time embeds each incoming question. Query embeddings are short and grow with traffic. Ingestion embeddings grow with corpus size and with how often you rebuild.

Vectors from different embedding models can't be compared, and new chunk boundaries produce new chunks, so changing either the embedding model or the chunking means re-embedding the whole corpus. Each rebuild is a one-time cost, but it comes back every time you change one of them. Version the index by embedding model and chunking configuration, build the new version alongside the old one, and switch traffic only after it passes your retrieval checks.

ILLUSTRATIVE: A CORPUS OF 2 MILLION CHUNKS AT 400 TOKENS EACH IS 2,000,000 × 400 = 800 MILLION TOKENS TO EMBED ON EVERY FULL REBUILD, BEFORE ANY QUERIES.

Storage and index size grow with chunk count and vector dimensions. A float32 vector takes 4 bytes per dimension.

ILLUSTRATIVE: AT 1,536 DIMENSIONS, ONE VECTOR IS 1,536 × 4 = 6,144 BYTES, AND 2 MILLION CHUNKS NEED 6,144 × 2,000,000 ≈ 12.3 GB OF RAW VECTORS BEFORE INDEX OVERHEAD. AT 768 DIMENSIONS THE RAW SIZE HALVES TO ABOUT 6.1 GB.

Run ingestion embeddings in large batches rather than one request per chunk; if your provider offers discounted asynchronous processing, ingestion is a natural fit (Batch and asynchronous inference). Store a content hash per document or chunk and embed only what changed since the last run, so a daily update doesn't re-embed the unchanged majority.

Smaller embedding models are often enough for retrieval and lower ingestion, query and storage costs together. Don't assume either way: run your own evaluation set (Choose the model the task needs) and compare how often the passages the answer needs land in the top results.

REFERENCES:

OpenAI web search, OpenAI file search, LangChain retrieval.

Helmcode

// PART V

Infrastructure and pricing

11 GPU inference and deployment economics

OVERVIEW
PRACTICAL
DEEP DIVE Comparing GPU purchase options
Capacity break-even as a range

GPU inference and deployment economics

Size GPU capacity for peak traffic and latency targets before comparing it with a metered API.

// LEVEL 1 Overview

A GPU instance can look inexpensive when you divide its hourly price by theoretical token throughput. That calculation assumes the hardware is busy producing useful work most of the time, which real traffic rarely allows.

A model may fit in GPU memory but struggle with concurrent long-context requests. Traffic may arrive in peaks, and strict latency requirements may force you to keep unused capacity available. Self-hosting also makes you responsible for operating the service.

Setting a GPU rental listing next to an API pricing page tells you little. Compare options that deliver the same outcomes at the same service level.

// LEVEL 2 Practical

Describe the workload before selecting infrastructure: input length, output length, concurrency, latency objectives, traffic peaks and availability requirements. Track time to first token, generated-token rate, queue time, tail latency and completed accepted tasks.

For a dense model with 32 billion parameters, raw weight storage is approximately 64 GB at 16-bit precision, 32 GB at 8-bit and 16 GB at 4-bit, before quantization metadata, runtime buffers, KV cache and other overhead. For a mixture-of-experts model, plan memory for the **total** weights, not just the number activated per token.

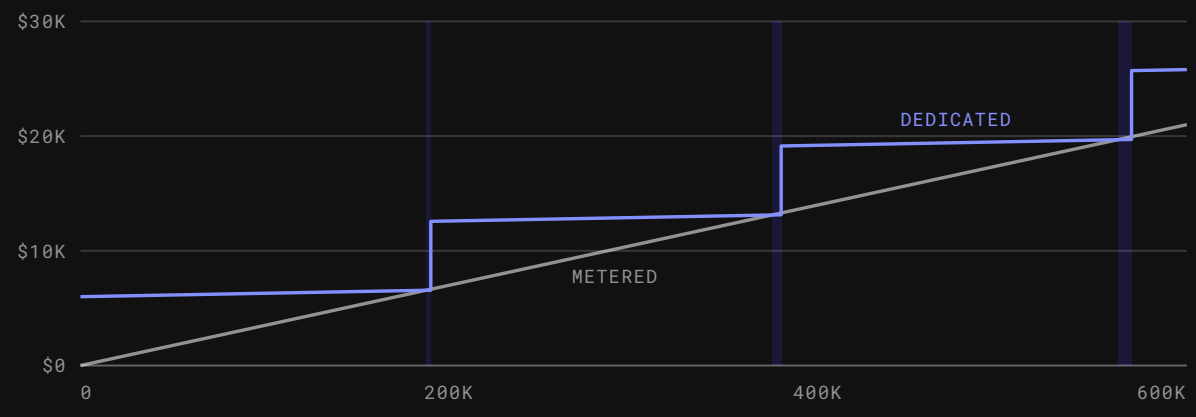
Compare three options:

	METERED API	DEDICATED CAPACITY (MANAGED)	SELF-HOSTED
Price pattern	Follows usage	Recurring or committed	Hardware + operations
Idle-capacity risk	Usually provider-side	Depends on plan	Your side
Serving responsibility	Provider	Provider	Your team
Control	Limited	Contract-dependent	Greater

Say a deployment costs $6,000 monthly and a metered API costs $0.035 per accepted task. If self-hosting adds $0.003 variable cost per task, the simplified break-even volume is:

$$N = \frac{6{,}000}{0.035 - 0.003} = 187{,}500 \text{ accepted tasks/month}$$

This holds only while the deployment can handle that volume without adding capacity and both options deliver equivalent quality and service. Real GPU costs often increase in steps as you add replicas.



Monthly cost against volume with the chapter's worksheet figures. Dedicated capacity is cheaper only in the shaded slivers before each new replica; past 190,000 tasks a second replica doubles the fixed cost.

// CASE · LEDGERLINE.

> Ledgerline priced a dedicated replica for extraction and stopped quickly. At 40,000 invoices a month, the whole extraction line costs $567, while a single replica in the chapter's example costs $6,000 before any traffic. The metered API stays until volume is several times higher, and the worksheet makes it easy to rerun the sums when it is.

// LEVEL 3 Deep Dive

During **prefill**, the model processes the input sequence. During **decode**, it generates output tokens. Scheduling and memory pressure differ between the two phases. Continuous batching can improve useful throughput by coordinating requests of different lengths; paged KV-cache management reduces wasted memory allocation; prefix caching can reuse the KV cache of a shared prefix across requests and skip repeating its prefill (see Tokens, context and prompt caching).

Quantization can reduce weight storage and sometimes improve throughput on supported hardware, but it can change model behavior and doesn't make generation free. A smaller weight footprint helps only if weight memory is what limits the deployment.

On the $6,000 monthly deployment from the Practical example, 100,000 accepted tasks allocate $0.06 per task. Handling 160,000 on the same hardware allocates $0.0375. The invoice doesn't change, but each task costs less. Pushed too far, higher utilization worsens queueing and P95 latency, so a reliable service may need some spare capacity.

Include failover, network, monitoring, upgrades, security and allocated engineering time in total cost. If migration costs $24,000 and saves $2,000 monthly in steady state, simple payback is 12 months before other risks and changes.

Comparing GPU purchase options

Cloud GPU capacity is usually sold in three forms. On-demand capacity costs the most per hour and can be released at any time, though the GPU type you want may not be available when you ask for it. Reserved or committed capacity costs less per hour in exchange for a term, and you pay for it whether traffic arrives or not. Spot or preemptible capacity is cheaper still, but the provider can reclaim it with little notice.

That interruption risk decides where spot fits. Batch jobs, offline evaluation and other retryable work can absorb a reclaimed node by resuming from a checkpoint or requeueing. Latency-critical serving usually can't, unless on-demand or reserved capacity covers the peak by itself.

Price whole nodes. A GPU-hour figure leaves out CPU and system memory, storage for model weights, networking and data transfer. Then convert the node price to cost per accepted task at the throughput you measured for your workload. Two GPUs can swap places in the ranking once throughput and

batching are included. Prices and discount structures change often, so check each provider's current pricing pages, such as the AWS EC2 pricing linked below, before running the comparison.

Capacity break-even as a range

Keep demand shape and financial assumptions in one worksheet:

VARIABLE	ILLUSTRATIVE VALUE	WHY IT MATTERS
Monthly accepted tasks	`180,000`	The denominator is useful work
Metered API cost per accepted task	`$0.035`	Includes relevant calls and recoveries
Dedicated fixed capacity	`$6,000/month`	Paid even during idle periods
Dedicated variable cost per task	`$0.003`	Egress, storage or other metered costs
One-time migration	`$24,000`	Not part of recurring run rate
Available dedicated capacity	`190,000 tasks/month`	Assumed at required peak latency

At 180,000 tasks, the metered alternative is `180,000 × $0.035 = $6,300/month`. Dedicated is `$6,000 + 180,000 × $0.003 = $6,540/month`: **more expensive**, even before migration. At 190,000 tasks, metered is $6,650 and dedicated is $6,570. At 220,000 tasks, though, the fixed deployment in this example runs out of capacity, so you can't extrapolate the apparent saving without adding another replica or changing the service requirement.

Put together, dedicated capacity in this example wins only in a narrow band, from 187,500 to 190,000 tasks a month, before a second replica is needed. Below the band the metered API is cheaper, and above it the second replica adds another $6,000 of fixed cost.

Monthly average traffic doesn't establish capacity on its own. Size for peak arrival rate, input and output length distribution, per-request context, concurrency and P95 latency. A quote for dedicated capacity or a flat-rate plan should spell out throughput and queue behavior as well as monthly token allowances.

When comparing a migration, keep recurring net savings separate from the one-time cost; The optimization playbook covers payback. Treat any break-even point as the edge of one scenario.

Before committing to dedicated capacity, rerun the worksheet with your own volumes, peak load and latency targets. The calculator below starts from the figures in the table; move the volume past one replica's capacity to see the step.

The same calculation, with a volume table, is in the downloadable worksheet.

REFERENCES:

vLLM, TensorRT-LLM, AWS EC2 pricing.

Helmcode

// PART VI

Put it into practice

12 The economics of an AI feature

OVERVIEW

PRACTICAL
- Start with the billable or valuable unit
- A subscription can conceal expensive users
- Decide who carries the usage risk
- A unit-economics worksheet

DEEP DIVE
- The margin changes when the product changes
- Marginal usage still costs capacity on a flat-rate plan
- Avoid using a cost metric as a product KPI
- Run the worksheet through a sensitivity table

13 The optimization playbook

OVERVIEW

PRACTICAL

DEEP DIVE

The economics of an AI feature

An inference call can be cheap and the feature built on top of it can still lose money.

// LEVEL 1 Overview

Take two products. One charges businesses for each document it successfully processes. The other sells a flat monthly subscription and lets customers run an AI assistant as often as they like.

They use the same models at the same negotiated API prices, yet their financial results can differ sharply.

For the document product, revenue grows with completed work, provided price and direct cost stay in proportion. For the subscription product, a few very active accounts may generate most of the inference spend while paying the same as everyone else.

Here cost optimization becomes a product-economics problem as well as an infrastructure one. You need to know **which customer action creates value, what it costs to serve, and how revenue changes as usage grows**.

Keep the gross margin of the whole company separate from the contribution margin of one AI feature. The second is a decision tool: it can show which usage patterns are sustainable before company overhead and other accounting choices are included.

// LEVEL 2 Practical

Start with the billable or valuable unit

For a document processor, the unit might be one accepted document; for a research agent, one completed report; for a coding product, one accepted task. For an AI assistant sold as part of a larger SaaS product, the relevant units may be **cost per active account per month** and **cost per useful interaction**.

Use at least two views of the economics:

> Contribution per task = Net revenue attributable to task − Variable cost of delivering task

> Contribution per account =
>
> Net account revenue − Account-attributable serving and support costs

Net revenue means revenue after applicable discounts, credits and refunds, rather than a marketing list price. Serving cost should include inference, paid tools, variable infrastructure and expected human correction. If you allocate shared fixed capacity, label that allocation separately so the variable margin doesn't get mixed up with fully allocated unit cost.

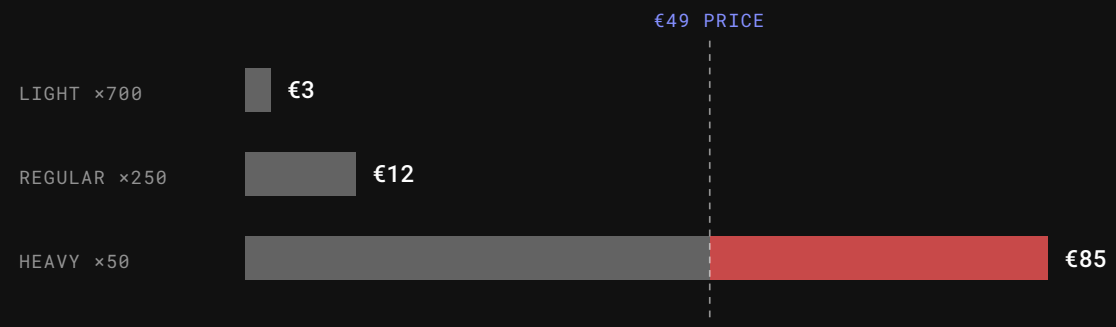
A subscription can conceal expensive users

A hypothetical AI product charges €49 per account per month, excluding taxes. Its usage is distributed like this:

ACCOUNT GROUP	ACCOUNTS	AVERAGE AI SERVING COST/ACCOUNT	TOTAL SERVING COST
Light users	700	€3	€2,100
Regular users	250	€12	€3,000
Heavy users	50	€85	€4,250
Total	**1,000**		**€9,350**

At 1,000 paying accounts, subscription revenue is `1,000 × €49 = €49,000/month`. AI serving costs are €9,350, or about 19.1% of revenue, which looks manageable in aggregate.

The fifty heavy accounts, however, generate €4,250 of cost against €2,450 of revenue. Their AI-serving contribution is **negative €1,800** before customer support and other expenses, and the other accounts subsidize them.



AI serving cost per account per month against a €49 subscription. The 50 heavy accounts cost €36 more each than they pay, and the other accounts cover the gap.

Heavy users can still be worth having, since they may retain longer, refer other customers or upgrade later. Even so, treat the subsidy as an explicit pricing and capacity decision. An average across all accounts hides this tail, as an average across agent tasks does.

Decide who carries the usage risk

Pricing models distribute risk differently:

PRODUCT PRICING	WHAT SCALES WITH USE	ECONOMIC RISK
Per completed task	Revenue often follows useful work	Need a transparent definition of acceptance and cost
Per token or credit	Revenue follows measured consumption	Customers may struggle to predict total spending
Flat subscription	Revenue mainly follows accounts	Heavy-user tail can overwhelm fixed revenue
Subscription + included allowance	Base revenue plus bounded entitlement	Overage rules and user experience need clarity
Seat pricing	Revenue follows licenses	Inference consumption per seat can vary dramatically

Credits need care. One “credit” might represent a model call, a fixed number of tokens or an abstract unit of work. If underlying model costs change, what a credit is worth economically can drift unless the entitlement is clear and versioned.

It often works to meter **valuable work** at the product layer and track tokens underneath as a cost driver. Customers understand a processed document or a completed report more easily than internal reasoning-token accounting. Task size isn’t always stable, though, so the product may need tiers or fair-use rules for unusually expensive work.

A unit-economics worksheet

Suppose a research-report feature has these hypothetical characteristics:

- Net revenue attributable to one completed report: €2.50.
- Average inference and tool cost per submitted report: €0.90.
- Acceptance rate: 80%.
- Additional expected review and delivery cost per accepted report: €0.30.

To deliver 100 accepted reports at an 80% acceptance rate, the simple expected submitted volume is `100 / 0.80 = 125`, assuming the rate is stable.

Estimated inference and tool spend is `125 × €0.90 = €112.50`. Add `100 × €0.30 = €30` in review and delivery costs: €142.50 total, or **€1.425 per accepted report**. Against €2.50 in revenue, contribution is €1.075 per accepted report before other allocated costs.

Comparing the €2.50 price only with the €0.90 per submitted task overstates the margin, because rejected and incomplete attempts are part of the cost of delivering accepted work.

The worksheet reproduces this calculation and the sensitivity table in the Deep Dive.

> // CASE · LEDGERLINE.
>
> Ledgerline sells research reports inside a $299 monthly plan. Each accepted report now costs $0.28 in inference ($560 / 2,000). Thirty of its 400 accounts generate 1,200 of the 2,000 monthly reports, 40 each, or $11.20 of inference per account: well inside the price. To keep that true if heavy accounts automate daily reports, the plan now includes 200 reports a month ($56 of inference) and charges for more.

// LEVEL 3

Deep Dive

The margin changes when the product changes

An AI application's cost distribution is rarely fixed. Adding tool access can turn a cheap assistant into a multi-step workflow. A larger context window may change prefill spending. Better model quality can reduce retries but raise per-call cost. A new enterprise customer may send larger documents or have much higher peak concurrency than existing users.

Track each account cohort on net revenue, accepted work, variable AI serving costs, paid tool usage, the fraction of users triggering fallbacks and P95 account-level spend. Revenue per account alone won't show a cohort drifting into the heavy-user tail.

Marginal usage still costs capacity on a flat-rate plan

Under a flat-rate plan, an extra eligible request may not raise the monthly bill directly, but it still occupies throughput. It could affect latency for other users or require a higher tier. A product with unlimited customer-facing usage can therefore consume capacity quickly while vendor invoicing stays predictable.

Keep a capacity view next to the financial one. The invoice explains current spend but won't tell you whether the operating model can support additional demand.

Avoid using a cost metric as a product KPI

Cost per accepted task can be optimized in ways that make the product worse. Acceptance rules strict enough that almost nothing qualifies make unit economics look terrible, and rules that accept every response make them look wonderful. Neither tells you whether customers got what they wanted.

Tie acceptance to real product outcomes where possible. For a tool used by employees, account for human time saved or lost. For a customer-facing feature, track whether the result was usable, whether another run was needed and whether users still value the feature. Some of these signals won't convert into precise dollar figures; leaving them out makes the numbers look more certain than they are.

Run the worksheet through a sensitivity table

Take the research-report worksheet and vary two inputs: the acceptance rate and the inference and tool cost per submitted report. Net revenue stays at €2.50 and review at €0.30 per accepted report.

$$\text{Cost per accepted report} = \frac{\text{Inference and tool cost per submitted report}}{\text{Acceptance rate} + €0.30}$$

ACCEPTANCE RATE	CONTRIBUTION AT €0.90 PER SUBMITTED REPORT	CONTRIBUTION AT €1.20 PER SUBMITTED REPORT
60%	€0.700	€0.200
80%	€1.075	€0.700
90%	€1.200	€0.867

ILLUSTRATIVE SCENARIO ARITHMETIC BASED ON THE WORKSHEET ABOVE, BEFORE ALLOCATED COSTS.

A cost rise from €0.90 to €1.20 per submitted report cuts contribution at 80% acceptance from €1.075 to €0.700. A drop in acceptance from 80% to 60% does the same at today's price. If both happen, €0.20 is left, 8% of revenue. Add rows for the inputs your product is most exposed to, such as a premium tool's share of tasks, and one for the usage level you are planning for.

Further reading: AWS Well-Architected: Cost Optimization, Stripe: Usage-based billing.

The optimization playbook

Six steps for finding the few changes that will lower your bill the most.

// LEVEL 1 Overview

We don't recommend implementing every technique in this guide; it could take months. Some changes remove a lot of unnecessary work. Others save almost nothing or make the system harder to maintain.

Start with the bill, then look at the workflow. Separate inference, retrieval and tools, infrastructure, subscriptions and operational overhead, attribute those costs to features and accepted tasks, and only then choose what to change.

// LEVEL 2 Practical

Work in this order:

01 **Find the expensive operations.** Identify the largest total costs and the worst cost per accepted task. Look at costly outliers as well as the average.

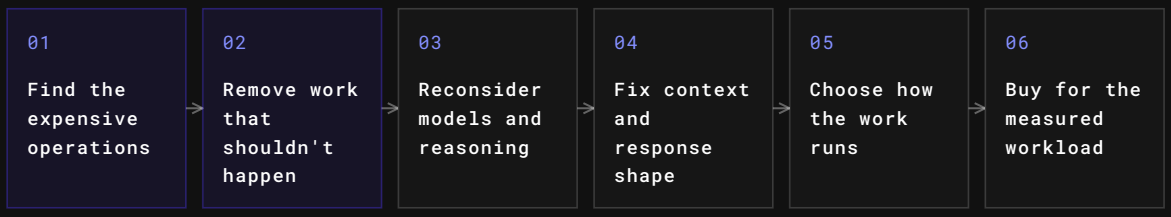
02 **Remove work that shouldn't happen.** Duplicate requests, repeated searches, needless LLM calculations and unused output can often be fixed without changing providers.

03 **Reconsider model and reasoning choices.** Use suitable smaller models for constrained tasks; add routing or cascades only when they make economic sense.

04 **Fix context and response shape.** Keep necessary evidence, remove irrelevant material, preserve reusable prefixes and compact agent state deliberately.

05 **Choose how the work runs.** Batch work that can wait. Compare a metered API, a flat-rate plan and dedicated capacity, managed or self-hosted, against the remaining workload.

06 **Buy for the workload you have measured.** Once volume and request shape are stable, look at committed-use or enterprise discounts, volume tiers, provisioned or reserved throughput, priority versus flexible service tiers, and the higher rates some providers charge for long context. Commit only to usage you have already observed, leave headroom for model and traffic changes, and compare the full cost of the commitment with the metered baseline for the same work.



The order matters: the first two steps remove spending without changing providers or models, so they usually pay back fastest.

Prioritize by net economic impact, confidence, implementation effort and risk. For illustration, removing duplicate extraction might save $400 per month with little engineering work, while a GPU migration promises $1,500 but carries a heavy maintenance load and depends on utilization. The larger headline figure alone shouldn't decide which comes first.

Before approving any optimization, ask:

- Which work does this change remove or make cheaper?
- How much of the total workflow does that work represent?
- What may get worse: correctness, latency, reliability, privacy or maintainability?
- What new ongoing cost does the change introduce?
- Are we comparing equivalent accepted outcomes?

Approve it only if the answers show it lowers the cost of equivalent accepted work once its new costs are counted.

The worksheet has a sheet for ranking candidate changes by saving, payback, confidence, effort and risk, with these questions as columns.

Check AI subscriptions outside your inference bill too: developer assistants, writing and research tools, meeting products, automation seats. Cancelling ten unused $30/month seats on a cancellable contract saves $300 a month, sometimes with less effort than redesigning part of the inference stack.

> // CASE · LEDGERLINE.
>
> Ledgerline's inference lines went from $1,200, $600 and $800 to $567, $330 and $560: $1,457 a month, 44% lower. An observability tool and regular evaluation runs add $140 a month, so the monthly total is $1,597 and the net saving $1,003, or 38.6%. The work took about $8,000 of engineering time, which pays back in eight months. The order followed the playbook: measure, remove duplicate and unnecessary work, right-size models, fix context, then change how work runs.

// LEVEL 3 Deep Dive

Measure **net savings** on the total, so a percentage never lands on the wrong denominator:

$$\text{Monthly net savings} = \text{Baseline monthly total} - \text{Optimized monthly total}$$

$$\text{Payback (months)} = \frac{\text{One-time cost}}{\text{Monthly net savings}}$$

Payback only makes sense when monthly net savings are positive. Keep one-time costs (migration, engineering time, contract exit fees) out of the monthly figure and show them through payback. If inference falls from $4,000 to $2,800 but the change adds $300 of tools and $400 of operations, monthly spending becomes $3,500. The inference line fell 30%; the total fell **12.5%**, a net saving of $500 a month. If the change took $4,000 of engineering work, payback is 8 months.

A budget dashboard can alert you, but agent spending limits have to be enforced in application code, as Agent loops, validation and retries describes.

Keep the routine light. Check unusual runs often, look at cost by feature and provider every month, and reconsider subscriptions, commitments and infrastructure less often. Large changes in traffic, models or product behavior are a reason to revisit your assumptions.

Helmcode

// PART VII

Provider note

14 When flat-rate inference makes sense

OVERVIEW
PRACTICAL
DEEP DIVE

When flat-rate inference makes sense

Provider-specific section. The rest of this guide is vendor-neutral.

VENDOR CHAPTER

// LEVEL 1 Overview

Once you've removed unnecessary calls and chosen suitable models, the remaining question is how to pay for the work your application does.

Metered APIs suit variable workloads. Dedicated capacity can be economical for predictable demand, but someone has to run it. A flat-rate plan charges a fixed monthly fee for a defined service, with model allowances and operating limits.

No inference service has unlimited capacity, whatever a plan says. This chapter uses Helmcode's plans as the worked example: what they include, where they stop, and how to tell whether one would lower your bill.

// LEVEL 2 Practical

Helmcode runs open-weight models on infrastructure it operates in Europe, behind an OpenAI-compatible API. Qwen and Gemma models have no monthly token cap. DeepSeek V4 Flash and GLM 5.3 Flash share a monthly allowance that depends on the plan.

Plans as of November 2026:

PLAN	MONTHLY PRICE	API KEYS	SHARED DEEPSEEK V4 FLASH / GLM 5.3 FLASH ALLOWANCE PER MONTH
Starter	€399	5	5 billion tokens
Growth	€1,299	15	15 billion tokens
Scale	€3,199	40	35 billion tokens
Enterprise	Custom	Custom	Custom terms

Every API key, including GLM 5.3 add-on keys, is limited to **2 million tokens per minute (2M TPM)**, and request-rate and concurrency limits also apply. GLM 5.3 is an add-on at €150 per enabled API key per month, with its own monthly allowance that depends on the plan. **A monthly allowance can't be extended once it runs out**, and there are no overage charges to fall back on. Proprietary models available through the platform are not part of the open-model flat rate.

Take a hypothetical company spending €1,000 a month on metered inference. If €800 of that can move to models covered by Starter and €200 stays with the original provider, the modeled total is €399 + €200 = €599: a gross difference of **€401 per month** before migration, fallback and other costs. It assumes the workload meets its quality and capacity requirements on the new models.

An application spending only €120 a month is unlikely to see a lower invoice on a €399 plan unless the plan replaces other costs or creates value elsewhere.

> // CASE · LEDGERLINE.
>
> Ledgerline's remaining metered spend on extraction and support is $897 a month; the research agent stays on its current proprietary model. Before considering a flat-rate plan, the team would check that the open models it covers pass the extraction and support evaluation sets, that peak traffic stays under 2M tokens per minute per key, and that a €399 Starter plan costs less than the share of that $897 which would actually move.

// LEVEL 3 Deep Dive

The break-even logic from GPU inference and deployment economics applies, with the plan fee as the fixed cost:

$$\text{Break-even volume} = \frac{\text{Monthly fixed fee}}{\text{Current variable cost per task} - \text{Remaining variable cost per task}}$$

At €399 fixed, €0.020 current cost per accepted task and €0.002 remaining variable cost, the simplified break-even is about **22,167 tasks per month**. The calculation no longer holds if the workload needs a larger tier, extra model charges or additional capacity.

A monthly token allowance says little about peak request rate. Check requests per minute, tokens per minute, per-key concurrency and context limits separately, against your busiest hour rather than your monthly average.

Model developer and inference operator are separate roles. Helmcode runs its open models in the EU with zero prompt logging; proprietary models reached through the same API are operated by their providers and can have different processing terms. Check the route for each model instead of assuming one API means one set of residency terms.

A flat-rate plan suits sustained workloads that stay within its limits. It doesn’t remove the need to cut wasted work: unnecessary calls still use allowance and throughput.

SOURCES:

Helmcode pricing, models, rate limits. Prices and limits as of November 2026.

// REFERENCE

Glossary

Acceptance rate

Accepted tasks divided by the tasks in the cohort. Report it next to cost per accepted task, since either number alone hides behavior. Loosening or tightening acceptance can move both.

SEE CH. 02 · CH. 12

Accepted task

A task whose result meets the requirements you defined for that operation. For extraction, for example: the right shape, consistent with business rules and correct against the source. Your product sets that bar. A successful API response alone doesn't meet it.

SEE CH. 01 · CH. 02

Allocated cost

A share of a fixed bill assigned to each task, such as a $1,000 monthly service divided by the tasks it handled. It changes with volume even when the bill does not. Compare with marginal cost.

SEE CH. 01 · CH. 11

Batch API

A provider mode for requests that can wait, often hours, usually at a lower price than real-time calls. This is a pricing tier. Running work asynchronously in your own system and batching on a GPU are separate things.

SEE CH. 08

Break-even volume

The monthly volume at which a fixed-cost option costs the same as a metered one: the fixed fee divided by the variable saving per task. It only holds while capacity, quality and service stay comparable, so treat it as a scenario boundary.

SEE CH. 11 · CH. 14

Cascade

Starting with a cheaper model and escalating to a stronger one when a check finds a failure or a weak result. Worth it if failures are caught reliably and the checks are cheap.

SEE CH. 04 · CH. 03

Chunking

How documents are split before indexing for retrieval. Chunks that are too small separate a rule from its exception; chunks that are too large pull irrelevant text into every request.

SEE CH. 10

Context compaction

Replacing a long transcript with a shorter working state: objective, constraints, findings, failed approaches and where the evidence lives. It costs tokens itself and can break prompt-prefix reuse, so do it when the reduction justifies it.

SEE CH. 07 · CH. 09

Continuous batching

A serving-engine technique that interleaves requests of different lengths on the same GPU to keep it busy. It raises throughput on hardware you run and is unrelated to a provider's batch discount.

SEE CH. 11 · CH. 08

Contribution margin

Net revenue for a task or an account minus the variable cost of serving it. Used per feature, it shows which usage patterns pay for themselves before company overhead is included.

SEE CH. 12

Cost per accepted task

Total attributable workflow cost divided by accepted tasks: inference, retrieval, tools, retries, compute, human correction and a share of fixed capacity. This is the guide's central metric: failed and abandoned work gets charged to the work that succeeded.

SEE CH. 01 · CH. 02

Decode

The phase in which a model generates output tokens one at a time. It stresses the hardware differently from prefill, which is why long inputs and long outputs cost differently to serve.

SEE CH. 11

Dedicated capacity ALSO FIXED CAPACITY

Inference capacity reserved for you and paid for by the month, whether a provider runs it or you run it on your own GPUs. The bill does not follow usage, so idle time is yours to pay for and growth arrives in steps.

SEE CH. 11

Distillation ALSO FINE-TUNING

Training a smaller model to do a narrow, stable task that a larger one handles today. It can lower the cost per call, and it adds training, evaluation, hosting and retraining costs. The license must allow it.

SEE CH. 03 · CH. 05

Escalation rate

The share of requests a cascade sends on to the stronger model. Together with the cost of each check, it decides whether the cascade is cheaper than using the stronger model directly.

SEE CH. 04 · CH. 03

Evaluation set ALSO EVAL SET

A fixed set of real cases with known good outcomes, used to compare models or prompts before switching and to catch regressions after a change. Running it costs money, and that cost belongs in the decision.

SEE CH. 03

Flat-rate plan ALSO FLAT-RATE INFERENCE

A recurring fee for a defined inference service, with model allowances and operational limits. No plan of this kind is unlimited, so check that your workload fits its allowances, rate limits, concurrency and latency.

SEE CH. 14 · CH. 12

Idempotency

The property that repeating the same intended operation has the effect of doing it once. It needs a durable operation identity, and for side effects such as payments or emails, a contract the downstream service supports. Different from response caching.

SEE CH. 07

KV cache

The attention state a request builds while reading its input, which every decode step then reads. It grows with context length and with the number of concurrent requests, which is why long contexts need more GPU memory.

SEE CH. 11 · CH. 06

Marginal cost

What one more task adds to the bill. On fixed capacity with room to spare it can be close to zero; once the service is saturated, the next task may require a new tier.

SEE CH. 01 · CH. 12

Metered API

Inference billed by usage, usually per input and output token. Simple to operate and the bill follows the work, which suits variable traffic.

SEE CH. 11 · CH. 05

Mixture of experts ALSO MOE

A model architecture that activates only part of its parameters for each token. Serving it still needs memory for the total weights, so active-parameter counts understate what it takes to host.

SEE CH. 05 · CH. 11

Open-weight model

A model whose weights are published for others to run, under a license that varies by release. Downloading costs nothing, but you still pay to run it, through an API, managed capacity or your own GPUs.

SEE CH. 05

P95

The value below which 95% of observations fall. Used for latency and for cost per task, because a few long runs can push the tail far above the median, and the average hides them.

SEE CH. 01 · CH. 11

Payback period

One-time cost divided by monthly net saving. It only makes sense when the monthly saving is positive; one-time costs stay out of the monthly figure.

SEE CH. 13 · CH. 11

Prefill

The phase in which a model reads the input before generating. Long prompts and retrieved context make it expensive; prefix caching can avoid repeating it.

SEE CH. 11 · CH. 06

Prefix caching

A serving engine keeping the KV cache of a shared prompt prefix and reusing it for later requests that start the same way. It saves the prefill for that prefix; every output token is still generated. On dedicated hardware the gain shows up as throughput rather than a smaller invoice.

SEE CH. 06 · CH. 11

Prompt caching

A provider feature that charges or processes a repeated input prefix differently. Writes, reads, TTLs and minimum sizes vary by provider, so check the cached-token fields in the response before counting savings.

SEE CH. 06

Quantization

Storing weights at lower precision, for example 8 or 4 bits instead of 16. It reduces memory and can raise throughput on supported hardware, but it can change model behavior.

SEE CH. 11 · CH. 05

RAG ALSO RETRIEVAL-AUGMENTED GENERATION

Retrieving relevant passages and sending them to the model with the question. Its cost spans ingestion, embeddings, storage, retrieval, reranking and every token of context it adds.

SEE CH. 10

Reasoning budget ALSO REASONING EFFORT

A control some reasoning models expose to limit internal generation before the answer. Set it per operation; an output-token limit is not a substitute and can truncate the answer instead.

SEE CH. 04 · CH. 03

Reranking

A second pass that reorders retrieved passages so only the most relevant reach the model. The extra pass has a cost, recovered only if it trims enough context or improves answers enough.

SEE CH. 10

Response caching

Reusing a valid earlier result for the same inputs, keyed by content and versions and scoped by tenant. It can save a whole call. Semantic caching, which matches similar rather than identical inputs, is riskier.

SEE CH. 07 · CH. 06

Retry amplification

Retries stacking across layers: an SDK that tries three times inside an application loop that tries three times can send nine requests upstream. Set one overall deadline and know each layer's policy.

SEE CH. 09

Routing

Choosing a model for each request. When the application already knows the operation, a static lookup is simpler than asking a model to route another model.

SEE CH. 04

Showback ALSO CHARGEBACK

Reporting AI spend to the team or feature that caused it. Chargeback goes one step further and bills it to their budget. Either needs costs tagged by team or feature first.

SEE CH. 02

Tokens per minute ALSO TPM

A rate limit on how many tokens a key or account can process each minute. A monthly allowance says little about it, so check TPM, request rate and concurrency separately.

SEE CH. 14

Total cost of ownership ALSO TCO

Everything it costs to run a service: inference or hardware, plus failover, network, monitoring, upgrades, security and engineering time, with one-time migration shown separately.

SEE CH. 11 · CH. 13

Helmcode

// READ IT ONLINE

Read it online, with the three reading depths, the capacity calculator and every update.

`helmcode.com/guides/ai-cost-optimization`

The online edition is the reference. This PDF is the edition of 4 Nov 2026.

Worksheet: `helmcode.com/downloads/ai-cost-optimization-worksheet.xlsx`

Baseline SQL: `helmcode.com/downloads/ai-cost-optimization-baseline.sql`

Edition of 4 Nov 2026: First edition: introduction and 14 chapters at three reading depths, the Ledgerline case, nine figures, a capacity calculator, a 38-term glossary, a worksheet, a SQL baseline and the PDF. Prices and limits as of November 2026.

AI Cost Optimization: The Practical Guide. Figures are illustrative unless a source is given; prices and limits change, so check them before you rely on them. Helmcode S.L.