What Does an AI Analysis Actually Cost?

How I turned OpenRouter usage into a credit ledger for Jiddu without pretending every token is worth the same

Rafael M. Ehlers · July 28, 2026
rafaelmehlers.com.br/essays/what-does-an-ai-analysis-actually-cost

---

When Jiddu was open to anyone, an innocent **Analyze** button had a direct line to my OpenRouter balance. A short fallacy scan might make one inexpensive model call. A deep adversarial review of a paper could call several models, use reasoning tokens, consult an independent critic, and retry a stage that returned invalid data.

Calling both of those actions "one analysis" would hide nearly everything that determines their cost.

I built Jiddu Credits to account for that shared spend and, eventually, let users buy credit packs instead of giving unlimited access to the platform account. The billing system then had to answer a less obvious question: how many credits should one analysis consume when its true cost is known only after it finishes?

The system I built has three layers:

1. OpenRouter reports the real cost of every completed model call.
2. Jiddu converts the total into an integer number of credits.
3. A rate card protects the account before execution and fills the gaps when usage data is missing.

Those layers do different jobs. The rate card estimates the cost before execution. OpenRouter's reported cost settles the charge afterward. The ledger keeps enough detail to explain the result later.

Tokens describe the work; cost prices it

The first version of a credit system is tempting to write as a token formula:

```
credits = input tokens × input rate + output tokens × output rate
```

That formula works only when the application controls one model with one stable price. Jiddu does not.

Different models charge different amounts for input and output. Some expose separate prices for cached input or reasoning tokens. The provider selected behind a model route may matter. Tokenizers also differ, so identical text does not necessarily produce the same token count across models. A paper review can mix cheaper and more expensive models inside the same job.

Copying that pricing logic into Jiddu would create a stale version of a catalog that changes outside the application and make historical charges hard to defend after a provider updates a price.

OpenRouter returns usage accounting with each response. Alongside prompt and completion token counts, the response can contain:


```
{
  "usage": {
    "prompt_tokens": 10000,
    "completion_tokens": 2000,
    "total_tokens": 12000,
    "cost": 0.0165,
    "prompt_tokens_details": {
      "cached_tokens": 0
    },
    "completion_tokens_details": {
      "reasoning_tokens": 400
    },
    "cost_details": {
      "upstream_inference_cost": 0.015
    }
  }
}
```


The important field for settlement is **usage.cost**: the amount charged to the OpenRouter account for that generation. OpenRouter has already applied the selected model's current prices to its input, output, cached, and reasoning usage.

Jiddu still records the token counts for audits, debugging, cost analysis, and future product decisions. It does not reprice them.

The conversion formula

Jiddu uses two environment variables:

```
JIDDU_CREDIT_MARGIN_MULTIPLIER=1.5
JIDDU_CREDIT_USD_VALUE=0.01
```

The settlement is:

```
provider cost = sum of usage.cost for every call in the operation

billable cost = provider cost × margin multiplier

credits = ceil(billable cost ÷ dollar value per credit)
```

With the current defaults:

```
credits = ceil(OpenRouter cost × 1.5 ÷ 0.01)
```

One Jiddu Credit represents one cent of internal billable value. It is not a claim that one cent was paid to OpenRouter or a retail promise. Stripe is not connected to this system at the time of writing.

The multiplier covers expenses beyond inference: payment fees, taxes, failed requests, infrastructure, customer support, and price movement. It is configurable because those economics should not be buried in application code.

A complete example

Suppose one model call returns:

```
prompt tokens:      10,000
completion tokens:   2,000
OpenRouter cost:   $0.0165
```

Jiddu calculates:

```
$0.0165 × 1.5 = $0.02475

$0.02475 ÷ $0.01 = 2.475 credits

ceil(2.475) = 3 credits
```

The ledger records three credits, plus the original token counts, provider cost, billed cost, model, generation identifier, and conversion settings.

The stored cost uses integer millionths of a dollar rather than a floating-point value:

```
$0.0165 = 16,500 micros
```

Using an integer avoids accumulating binary floating-point errors in financial records. The displayed dollar value can always be reconstructed by dividing the stored integer by one million.

One feature can make several calls

A single feature can produce several provider generations.

A long document is split into chunks so it can fit within a model's context window. Each chunk can produce a separate call. Fact-checking adds one or more verification searches. An adversarial paper review coordinates several specialist stages and may use a different model for an independent critique.

Jiddu adds the cost of every call in the operation and rounds only once:

```
call 1: $0.012
call 2: $0.035
call 3: $0.004
----------------
total:  $0.051
```

The credit charge becomes:

```
$0.051 × 1.5 = $0.0765

$0.0765 ÷ $0.01 = 7.65

ceil(7.65) = 8 credits
```

Rounding each call separately would overcharge any operation made of many small calls. Aggregating first keeps the calculation tied to the cost of the whole action the user requested.

A 1.5 multiplier is not a 50% margin

This trips people up because markup and gross margin use different denominators.

If the provider cost is one dollar:

```
provider cost: $1.00
multiplier:       1.5
billable value: $1.50
gross profit:   $0.50
```

The markup on cost is 50 percent:

```
$0.50 ÷ $1.00 = 50%
```

The gross margin on revenue is 33.3 percent:

```
$0.50 ÷ $1.50 = 33.3%
```

A true 50 percent gross margin would require a multiplier of 2.0 before payment fees, taxes, infrastructure, and other expenses.

At the current settings, 1,000 credits carry ten dollars of internal billable value and cover approximately $6.67 of provider cost:

```
$10.00 ÷ 1.5 = $6.67
```

I kept the multiplier configurable because a number that looks generous in a formula may be much thinner after the rest of the business reaches it.

Integer credits create a minimum charge

Credits are integers. Any positive metered cost consumes at least one:

```
provider cost: $0.001
billable cost: $0.0015
raw credits:   0.15
final charge:  1 credit
```

That makes the effective markup on very small calls much higher than 1.5. The rounding effect falls as the operation gets larger.

I accept the one-credit floor for now because fractional credits would complicate the interface, ledger, and future credit packs. The system is proportional at normal analysis sizes, but not for very small calls.

If real usage produces many sub-cent operations, I can make the credit unit smaller, aggregate charges over time, or store fractional internal units while showing rounded balances to the user.

Reserve first, settle after

The exact cost is available only when the provider responds. Jiddu still needs to know that a user can afford an operation before starting an expensive review.

The rate card handles that check before execution. Jiddu estimates the maximum likely charge from the feature, model chain, number of chunks, and possible fallback calls, then reserves that many credits on the account.

A reservation leaves the balance untouched but reduces the amount available to other concurrent operations:

```
balance:   100 credits
reserved:   20 credits
available:  80 credits
```

After the operation finishes, Jiddu settles against actual usage.

If the reservation was 20 credits and the real charge is seven, Jiddu debits seven and releases the remaining 13. If the real charge is 24, it debits 24. The reservation is a solvency check, not a billing ceiling.

An overrun can leave the account negative. Future reservations are blocked until the balance is topped up. Allowing the overrun is more accurate than discarding provider cost because the estimate was wrong.

Pending reservations expire after two hours. This prevents a browser tab, network failure, or abandoned job from locking credits forever.

The rate card is also a fallback

OpenRouter normally returns **usage.cost**, but the billing system still needs a deterministic fallback.

A response may contain token counts but omit cost. A call can fail after the provider has performed work. An integration can return incomplete usage metadata. Jiddu needs a deterministic result in those cases.

The current rate card assigns a fixed estimate to each known model:

Model	Fallback credits per call
MiniMax M3	1
GPT-5.4 mini	2
Perplexity Sonar Reasoning Pro	2
GPT-5.5	10
Claude Opus 4.8	18
GPT Live Transcribe	20 per session
Unknown model	20

These numbers are versioned, so a ledger row can show which rate card was used even after the current table changes.

Jiddu applies the fallback only to calls that cannot be matched to reported cost. Consider an operation with two calls:

```
GPT-5.4 mini
reported cost: $0.012
converted charge: ceil($0.012 × 1.5 ÷ $0.01) = 2 credits
```

```
MiniMax M3
reported cost: missing
fallback charge: 1 credit

total: 3 credits
```

An explicit `cost: 0` is different from a missing cost. Zero means OpenRouter reported that the generation was free, so Jiddu charges no fallback credit for it. Missing means the price is unknown, so the rate card applies.

The same table provides the reservation estimate before execution. Actual provider accounting replaces that estimate once the operation completes.

Retries and failed stages still have a cost

A model can return syntactically valid text that fails Jiddu's schema validation. A request can be interrupted after the provider has already generated tokens. A fallback model can then complete the same stage.

One user action can consume paid inference across several attempts.

When provider usage is available, Jiddu records and charges each attempt even if its output could not be used. The successful retry is recorded separately. This keeps the ledger aligned with the account that paid the upstream bill.

Those failed attempts also give me evidence to improve the product. If one model causes frequent paid retries, the problem is visible in its completion rate, usage records, and cost per successful analysis. Hiding those calls inside a flat feature price would conceal an engineering problem behind the customer's bill.

What the ledger remembers

A balance alone can tell a user that credits disappeared. It cannot explain why.

Each usage transaction records:

- the feature and resulting resource;
- the credit debit and balance after settlement;
- the reservation estimate and any overrun;
- prompt, completion, total, reasoning, and cached tokens;
- provider cost and billable cost in micros;
- the margin multiplier and dollar value per credit;
- model and generation identifiers for each call;
- whether the call completed;
- any fallback models and fallback credits;
- the version of the rate card.

That record makes a charge reproducible. Given the provider cost and conversion settings, I can calculate the debit again months later. The per-call data shows whether the expense came from a long prompt, heavy reasoning, an expensive critic, or a retry.

Manual grants and usage debits remain separate. When an administrator adds 500 credits, that is one transaction. When an analysis consumes eight, that is another. Adjusting the balance directly without recording both events would make reconciliation impossible.

Live transcription is the current exception

Jiddu's live mode sends audio from the browser directly to OpenAI's Realtime API using `gpt-live-transcribe`. That traffic bypasses OpenRouter, so there is no OpenRouter `usage.cost` to settle.

The current implementation charges a fixed 20 credits for a transcription session capped at 60 minutes. Live analysis windows still use OpenRouter and settle from actual usage like other model calls.

The fixed transcription price is conservative. OpenAI currently lists `gpt-live-transcribe` at $0.017 per minute, so a more proportional calculation could be:

```
transcription cost = billable duration × $0.017 per minute

credits = ceil(transcription cost × margin multiplier ÷ value per credit)
```

I have not implemented that calculation yet. It requires a trustworthy server-side record of session start, end, reconnection, and maximum duration. A browser timer would be easy to manipulate and hard to audit.

Until that measurement exists, the fixed rate remains separate and visible instead of pretending to be usage-based.

What happens when Stripe arrives

Jiddu Credits are still an internal accounting unit. One credit represents $0.01 of billable value during settlement, but users cannot buy a pack yet.

When Stripe arrives, the sale price and the settlement value should remain separate. A pack can include a volume discount, a promotion, tax, or a payment fee without changing how much provider usage one credit covers inside the ledger.

Keeping those concepts separate also prevents a future pricing page from rewriting historical usage. The transaction records the multiplier, credit value, provider cost, and rate-card version that applied when it happened.

Before choosing retail packs, I want real distributions: cost per feature, cost per successful result, retry spend, reasoning-token share, reservation overruns, and the size of the rounding effect. The ledger now collects those numbers.