Optimize cost per accepted result

Key takeaways

1. Optimize dollars per accepted result, not price per request.
2. Routing, prefix caching, and batching each cut a different part of the bill.
3. Throughput is a token-budget problem before it is a worker-count problem.

Common misunderstandings of LLM costs

Which claim would you defend right now? Choose before the evidence.

"The cheapest model minimizes total cost."

"Increasing worker concurrency always increases throughput."

"Configuring prompt caching proves the request received a discount."

Keep your choice. Look for the mechanism or counterexample that would make it fail.

The workload: 1,000 code reviews a day

30k	1k	1,000	?
input tokens / review	**output tokens / review**	**reviews / day**	**accepted reviews**
25k stable + 5k diff	structured findings	steady production load	the denominator that matters

A cheaper call that fails the quality gate is not a saving. It is paid rework, and often a second call.

Pennies per review become a budget at scale

Illustrative Fall 2026 Sonnet list price: $3 per million input tokens and $15 per million output tokens. Write the arithmetic before optimizing.

Baseline receipt × workload

```
input_cost  = 30_000 / 1_000_000 * $3   = $0.090
output_cost =  1_000 / 1_000_000 * $15  = $0.015
request_cost                              = $0.105

daily  = 1_000 * $0.105 = $105
annual = 365 * $105     = $38,325

# before retries, rejected output, storage, or human review
```

source: platform.claude.com/docs/en/about-claude/pricing (verify current prices)

Put accepted results in the denominator

The optimization target

```
total_cost = api + tools + queue + human_review + rework
cost_per_accepted = total_cost / count(results_that_pass_quality_gate)
acceptance_rate = accepted / attempted
```

metric	failure it exposes
$/request	hides rejects and repeated attempts
tokens/request	hides model price and non-token tools
acceptance rate	hides how expensive each attempt was
$/accepted result	forces quality and spend into one outcome

Route model and effort with evals, not vibes

A versioned routing policy

```
def route(change: Change) -> Lane:
    risky = (
        change.touches_auth
        or change.changed_lines > 500
        or change.language not in CALIBRATED
    )
    return Lane(
        model=STRONG if risky else FAST,
        effort="high" if risky else "low",
    )
```

Evidence required for each lane

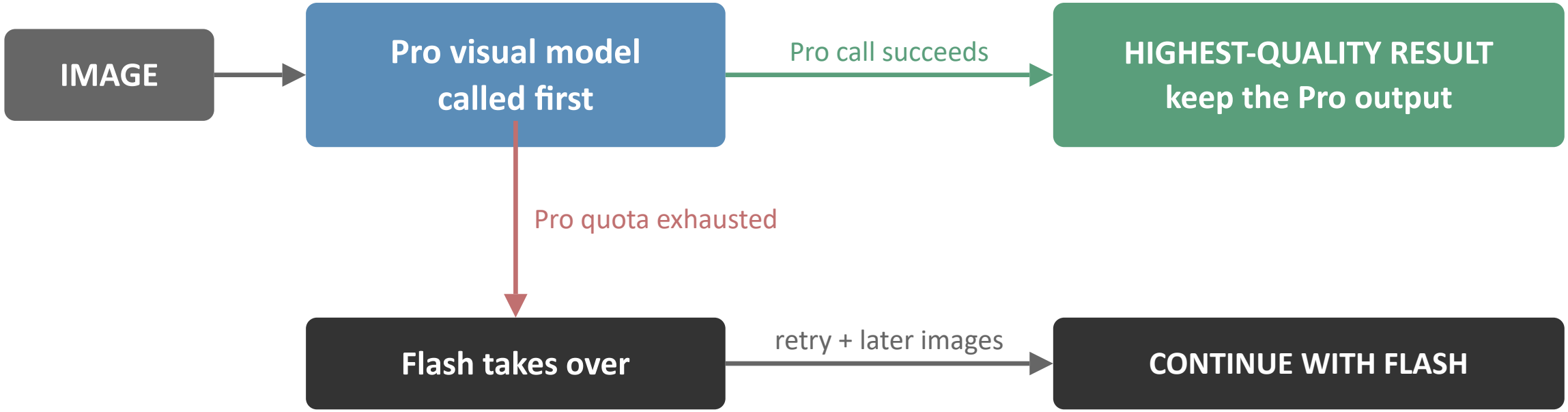
```
cohort = evals.for_lane(lane)
assert cohort.n >= 100
assert cohort.acceptance >= TARGET
assert cohort.false_negative <= MAX_FN

# Re-evaluate after model, prompt,
# schema, or policy changes.
```

Effort trades token spend for thoroughness inside one model. Routing is an experimental claim that a cheaper lane still meets the task's bar.

source: platform.claude.com/docs/en/build-with-claude/effort

Best-first maximizes quality, not cost

Pro goes first; Flash runs only after the Pro quota is exhausted



This is quota failover: it preserves progress, but it spends Pro quota first.

Pro cost >$100—and quota still forced Flash

The first cascade preserved progress, but it did not control spend

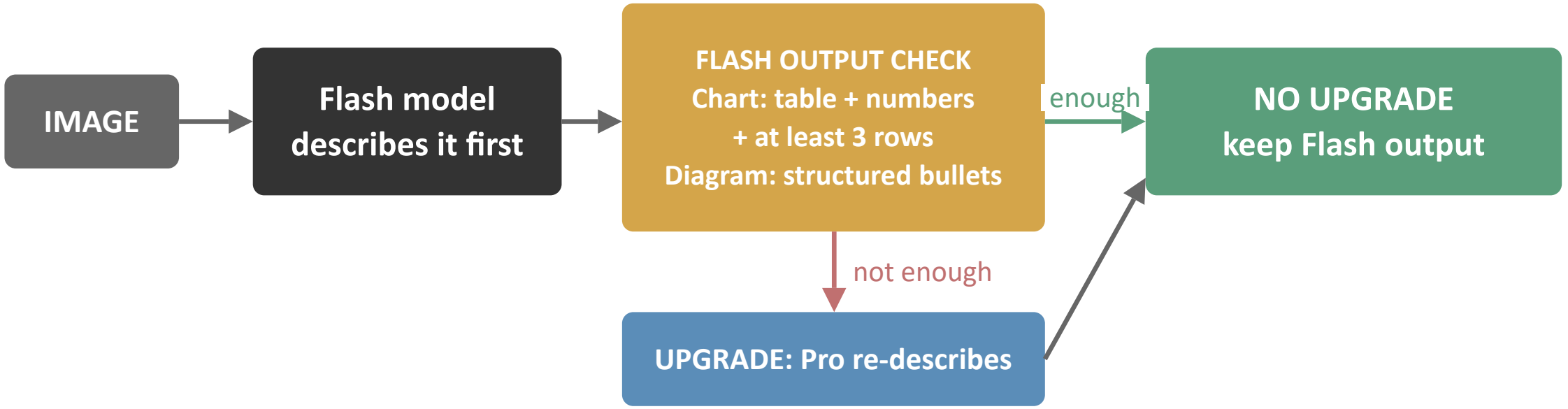
Observation	What happened in the bulk run
Workload	179 documents • 2,077 images
Pro spend	> $100 on successful Pro calls
Quota	Daily Pro quota exhausted within hours
Cascade down	Flash retried quota failures and handled later images

Quota failover keeps the batch moving. It cannot undo the cost of every successful Pro call made before the quota runs out.

Historical run, March 2026. The repository records the scope, quota failure, and ≈10× Pro/Flash rate gap; >$100 is the project owner's remembered bill.

Flash output signals when Pro is needed

Flash runs first; its output structure determines whether to pay for Pro.



Benchmark: Flash output had enough structure on 116/120 images (96.7%). In practice, only 5–10% of images needed Pro.

Cache the stable prefix—not the answer

Stable prefix — 25k tokens

```
system = [
  {"type": "text", "text": POLICY},
  {"type": "text", "text": REPO_GUIDE},
  {"type": "text", "text": EXAMPLES,
   "cache_control": {"type": "ephemeral"}},
]

# breakpoint ends at the last byte-identical
block
```

Varying suffix — 5k tokens

```
messages = [{
  "role": "user",
  "content": diff_and_task,
}]

# timestamps or per-job data before the
# breakpoint destroy prefix equality
```

Anthropic reports up to 90% lower input cost and 85% lower latency for long prompts. ‘Up to’ applies to the cached input portion, not total job cost.

source: anthropic.com/news/token-saving-updates; platform.claude.com/docs/en/build-with-claude/prompt-caching

Cache hits cut this review from 10.5¢ to 3.75¢

Fall 2026 price example; M = 1,000,000 tokens

```
baseline = 30_000*$3/M + 1_000*$15/M
         = $0.1050

cache_hit = 5_000*$3/M + 25_000*$0.30/M + 1_000*$15/M
          = $0.015  + $0.0075       + $0.015
          = $0.0375                 # 64.3% lower total cost

first_write = 25_000*$3.75/M = $0.09375
# Check usage.cache_read_input_tokens; no hit means no saving.
```

Caching pays when requests share a byte-identical, sufficiently large prefix within its TTL. Receipts, not configuration, prove a hit.

Batch when latency is not part of the product

Best fit: backfills, nightly evals, corpus review, offline extraction

```
requests = [
    {"custom_id": job.id, "params": make_request(job)}
    for job in overnight_reviews
]
batch = client.messages.batches.create(
    requests=requests,
)

# Persist custom_id -> job; reconcile every returned result.
# The create call does not mean the jobs are complete.
```

Use batch when the deadline is minutes or hours. Persist custom_id, poll, and account for missing, expired, and failed results.

source: platform.claude.com/docs/en/build-with-claude/batch-processing

1,000 reviews cost $105, $52.50, or $18.75

design	cost / review	cost / day	cost / year
standard	$0.1050	$105.00	$38,325
cache hits	$0.0375	$37.50	$13,688
batch only	$0.0525	$52.50	$19,163
cache + batch hits	$0.01875	$18.75	$6,844

- cache rows show steady-state hits; first write adds about 9¢ standard or 5¢ batch
- combined discounts require actual cache hits during batch execution — verify receipts
- prices and model behavior change; rerun this table from current price + eval data

source: Fall 2026 example from Claude pricing, prompt-caching, and batch-processing docs

Token budgets—not workers—set throughput

More concurrent workers cannot beat the token budget. They reach it sooner—or create a burst of 429s.

Theoretical floor before service latency and queue overhead

```
rpm  = 1_000 / 1_000              # 1.0 min
itpm = 30_000_000 / 2_000_000     # 15.0 min
otpm = 1_000_000 / 400_000         # 2.5 min
minutes = max(rpm, itpm, otpm)     # ITPM wins

# With 25k cached input tokens per job:
cached = max(1.0, 2.5, 2.5)        # 2.5 min
```

Quota sets gate width; workers set arrival rate.

source: platform.claude.com/docs/en/api/rate-limits

Separate worker and token limits

Worker bound

```
workers = asyncio.Semaphore(32)

async def run(job):
    async with workers:
        return await call(job)

# limits connections and memory,
# not tokens per minute
```

Token admission bound

```
await input_budget.acquire(
    estimate_uncached_input(job)
)
await output_budget.acquire(
    job.max_tokens
)
queue.submit(job)

# reconcile estimates with receipts
```

The queue owns admission. Workers execute admitted jobs. Receipts update the estimator so a bad token forecast does not become permanent policy.

You vs. the Agent: LLM workload economics

Your Responsibility	Agent's Responsibility
Define the quality gate and acceptable false-accept rate	Estimate tokens and attach usage receipts
Choose latency, budget, and throughput objectives	Route jobs under the versioned policy
Approve routing cohorts and re-evaluation triggers	Reconcile batches and surface rejected/ missing results
Decide when cached or batch processing fits the product	Report cost, latency, cache-hit, and acceptance metrics

Spend follows evidence

1. Optimize dollars per accepted result; raw token price ignores quality and rework.
2. Model/effort routing, stable-prefix caching, and batches solve different parts of the bill.
3. Throughput is a token-budget problem first and a worker-count problem second.