

lab_answer

April 3, 2026

1 Lab 14 — Cost, Caching & Rate Limits (ANSWERS)

Explore prompt caching, cost estimation, and the Batch API.

```
[ ]: # Setup
import json
import time
from dotenv import load_dotenv
import anthropic

load_dotenv()
client = anthropic.Anthropic()
MODEL = "claude-sonnet-4-20250514"
print("Setup complete.")
```

```
[ ]: # Long system prompt for caching exercises
# This generates a ~2000 token system prompt to test caching

SYSTEM_PROMPT = """You are an expert technical writing assistant specialising
    ↪in software documentation. Your role is to help developers write clear,
    ↪accurate, and comprehensive documentation for their projects.

## Core Principles

1. Clarity First: Every sentence should convey exactly one idea. Avoid
    ↪ambiguity. If a term could be interpreted multiple ways, define it
    ↪explicitly.

2. Audience Awareness: Always consider who will read the documentation. API
    ↪references are for experienced developers; tutorials are for beginners.
    ↪Adjust vocabulary, detail level, and examples accordingly.

3. Structure: Use consistent heading hierarchies. Start with a brief
    ↪overview, then dive into details. Group related concepts together. Provide a
    ↪table of contents for documents longer than 3 sections.
```

4. ****Code Examples****: Every concept should include a working code example.□
↳Examples should be:
- Complete (can be copied and run without modification)
 - Minimal (no unnecessary code that distracts from the concept)
 - Annotated (comments explaining non-obvious lines)
 - Tested (verified to work with the current version)
5. ****Error Documentation****: Document common errors and their solutions.□
↳Developers spend more time debugging than writing new code, so good error□
↳documentation is extremely valuable.

Writing Style Guide

- Use active voice: "The function returns a list" not "A list is returned by□
↳the function"
- Use present tense: "This method creates a new instance" not "This method will□
↳create a new instance"
- Use second person for tutorials: "You can configure the timeout by..."
- Use imperative mood for instructions: "Set the environment variable" not "You□
↳should set the environment variable"
- Keep paragraphs short (3-5 sentences maximum)
- Use bullet points for lists of 3 or more items
- Use numbered lists for sequential steps
- Use tables for comparing features or options

Documentation Types

API Reference

- Document every public function, class, and method
- Include: signature, parameters (with types), return value, exceptions,□
↳examples
- Use consistent formatting across all entries

Tutorials

- Start with what the reader will build/learn
- List prerequisites upfront
- Provide step-by-step instructions
- Include checkpoints where the reader can verify progress
- End with next steps and further reading

Conceptual Guides

- Explain the "why" before the "how"
- Use analogies to connect new concepts to familiar ones
- Include diagrams where they add clarity
- Link to API reference for implementation details

```

### Changelog
- Group changes by category (features, fixes, breaking changes)
- Use semantic versioning
- Include migration guides for breaking changes

## Quality Checklist
Before finalising any documentation, verify:
- All code examples run without errors
- Links point to valid destinations
- Technical terms are used consistently
- No placeholder text remains (TODO, TBD, FIXME)
- Formatting renders correctly in the target platform
- Screenshots are current and annotated if necessary

Remember: good documentation is an investment that pays dividends in reduced_
↳support burden, faster onboarding, and higher developer satisfaction. Treat_
↳documentation with the same rigour as production code."""

print(f"System prompt length: {len(SYSTEM_PROMPT)} characters")

```

1.1 Exercise 1 — Prompt Caching

```

[ ]: # Pricing constants (Sonnet 4, per token)
INPUT_PRICE = 3.0 / 1_000_000      # $3 per MTok
OUTPUT_PRICE = 15.0 / 1_000_000   # $15 per MTok
CACHE_WRITE_PRICE = INPUT_PRICE * 1.25  # 25% premium
CACHE_READ_PRICE = INPUT_PRICE * 0.10   # 90% discount

def calc_cost(usage):
    """Calculate the cost of a request from its usage object."""
    cost = 0
    cost += getattr(usage, 'input_tokens', 0) * INPUT_PRICE
    cost += getattr(usage, 'output_tokens', 0) * OUTPUT_PRICE
    cost += getattr(usage, 'cache_creation_input_tokens', 0) * CACHE_WRITE_PRICE
    cost += getattr(usage, 'cache_read_input_tokens', 0) * CACHE_READ_PRICE
    return cost

def print_usage(label, usage, cost):
    print(f"\n{label}:")
    print(f"  input_tokens: {usage.input_tokens}")
    print(f"  output_tokens: {usage.output_tokens}")
    print(f"  cache_creation: {getattr(usage, 'cache_creation_input_tokens', 0)}")
    print(f"  cache_read: {getattr(usage, 'cache_read_input_tokens', 0)}")
    print(f"  Cost: ${cost:.6f}")

```

```

question = "What are your top 3 tips for writing good API documentation?"

# Call 1: Cache write
r1 = client.messages.create(
    model=MODEL, max_tokens=256,
    system=[{"type": "text", "text": SYSTEM_PROMPT, "cache_control": {"type": "ephemeral"}}],
    messages=[{"role": "user", "content": question}],
)
c1 = calc_cost(r1.usage)
print_usage("Call 1 (cache write)", r1.usage, c1)

# Call 2: Cache read (same prompt)
r2 = client.messages.create(
    model=MODEL, max_tokens=256,
    system=[{"type": "text", "text": SYSTEM_PROMPT, "cache_control": {"type": "ephemeral"}}],
    messages=[{"role": "user", "content": question}],
)
c2 = calc_cost(r2.usage)
print_usage("Call 2 (cache read)", r2.usage, c2)

# Call 3: No caching
r3 = client.messages.create(
    model=MODEL, max_tokens=256,
    system=SYSTEM_PROMPT,
    messages=[{"role": "user", "content": question}],
)
c3 = calc_cost(r3.usage)
print_usage("Call 3 (no cache)", r3.usage, c3)

# Summary
print(f"\n{'='*50}")
print(f"Cache write cost: ${c1:.6f}")
print(f"Cache read cost:  ${c2:.6f}")
print(f"No cache cost:    ${c3:.6f}")
print(f"Savings (read vs no-cache): {(1 - c2/c3)*100:.1f}%")

```

1.2 Exercise 2 — Token Budget Checker

```

[ ]: # Pricing per token
PRICING = {
    "claude-sonnet-4-20250514": {"input": 3.0 / 1e6, "output": 15.0 / 1e6},
    "claude-haiku-3-5-20241022": {"input": 0.80 / 1e6, "output": 4.0 / 1e6},

```

```

}

def estimate_cost(model: str, input_tokens: int, max_output_tokens: int) -> float:
    """Estimate the maximum cost of an API call."""
    prices = PRICING.get(model, PRICING["claude-sonnet-4-20250514"])
    return input_tokens * prices["input"] + max_output_tokens * prices["output"]

def check_budget(model: str, messages: list, system_prompt: str,
                 max_cost_usd: float, max_output_tokens: int = 1024) -> dict:
    """Check if a request is within budget. Returns status dict."""
    # Count input tokens
    count_result = client.messages.count_tokens(
        model=model,
        system=system_prompt,
        messages=messages,
    )
    input_tokens = count_result.input_tokens

    est_cost = estimate_cost(model, input_tokens, max_output_tokens)

    return {
        "input_tokens": input_tokens,
        "estimated_max_cost": est_cost,
        "budget": max_cost_usd,
        "within_budget": est_cost <= max_cost_usd,
        "usage_pct": (est_cost / max_cost_usd) * 100,
    }

# Test
messages = [{"role": "user", "content": "Write a 500-word essay about Python."}]

result = check_budget(MODEL, messages, SYSTEM_PROMPT, max_cost_usd=0.05)
print(f"Input tokens: {result['input_tokens']}")
print(f"Estimated max cost: ${result['estimated_max_cost']:.4f}")
print(f"Budget: ${result['budget']:.4f}")
print(f"Within budget: {result['within_budget']}")
print(f"Usage: {result['usage_pct']:.1f}%")

# Test with a very low budget
result2 = check_budget(MODEL, messages, SYSTEM_PROMPT, max_cost_usd=0.001)
print(f"\nTight budget test:")
print(f"Within budget: {result2['within_budget']} (usage: {result2['usage_pct']:.1f}%")

```

1.3 Exercise 3 — Batch API

```
[ ]: SAMPLE_TEXTS = [
    "The new iPhone features a faster processor and improved camera system.",
    "Scientists discovered a new species of deep-sea fish near hydrothermal_
    ↪vents.",
    "The stock market rallied today on positive earnings reports from tech_
    ↪companies.",
    "A new study links regular exercise to improved cognitive function in older_
    ↪adults.",
    "The latest Python release includes pattern matching and improved error_
    ↪messages.",
    "Climate change is causing more frequent and severe weather events globally.
    ↪",
    "The restaurant received a Michelin star for its innovative fusion cuisine.
    ↪",
    "SpaceX successfully launched another batch of Starlink satellites into_
    ↪orbit.",
    "A major cybersecurity breach exposed millions of customer records.",
    "The new electric vehicle can travel 400 miles on a single charge.",
    "Researchers developed a more efficient solar cell using perovskite_
    ↪materials.",
    "The housing market shows signs of cooling after two years of rapid growth.
    ↪",
    "A breakthrough in quantum computing achieved a new record for qubit_
    ↪stability.",
    "The film received critical acclaim at the Cannes Film Festival.",
    "New regulations require companies to disclose AI usage in customer_
    ↪interactions.",
    "The World Health Organization issued guidelines for a new respiratory_
    ↪virus.",
    "An archaeological dig uncovered a 3000-year-old temple in southern Egypt.",
    "The central bank raised interest rates to combat persistent inflation.",
    "A new programming language aims to combine Rust safety with Python_
    ↪simplicity.",
    "The Olympics committee announced the addition of three new sports for 2028.
    ↪",
]

CLASSIFY_PROMPT = "Classify this text into exactly one category: Technology,_
    ↪Science, Finance, Health, Entertainment, Politics, Sports. Respond with only_
    ↪the category name."

# Create batch requests
batch_requests = []
for i, text in enumerate(SAMPLE_TEXTS):
```

```

batch_requests.append({
    "custom_id": f"text-{i:02d}",
    "params": {
        "model": MODEL,
        "max_tokens": 20,
        "messages": [{"role": "user", "content":
↪f"{CLASSIFY_PROMPT}\n\nText: {text}"}],
    }
})

# Submit batch
batch = client.messages.batches.create(requests=batch_requests)
print(f"Batch submitted: {batch.id}")
print(f"Status: {batch.processing_status}")

# Poll for completion
while True:
    batch = client.messages.batches.retrieve(batch.id)
    counts = batch.request_counts
    print(f"  Status: {batch.processing_status} | "
          f"succeeded={counts.succeeded} processing={counts.processing} "
          f"errored={counts.errored}")
    if batch.processing_status == "ended":
        break
    time.sleep(5)

print(f"\nBatch completed!")

```

```

[ ]: # Process batch results
results = []
for result in client.messages.batches.results(batch.id):
    custom_id = result.custom_id
    idx = int(custom_id.split("-")[1])
    if result.result.type == "succeeded":
        category = result.result.message.content[0].text.strip()
        results.append({"id": idx, "text": SAMPLE_TEXTS[idx][:60], "category":
↪category})
    else:
        results.append({"id": idx, "text": SAMPLE_TEXTS[idx][:60], "category":
↪"ERROR"})

# Display results
print(f"{'ID':<4} {'Category':<15} {'Text'}")
print("-" * 80)
for r in sorted(results, key=lambda x: x["id"]):
    print(f"{r['id']:<4} {r['category']:<15} {r['text']}\n...")

```

```

# Cost comparison
batch_input_price = 1.50 / 1e6 # 50% of Sonnet input
batch_output_price = 7.50 / 1e6 # 50% of Sonnet output
realtime_input_price = 3.0 / 1e6
realtime_output_price = 15.0 / 1e6

est_input_per_task = 50 # rough estimate
est_output_per_task = 5
n_tasks = len(SAMPLE_TEXTS)

batch_cost = n_tasks * (est_input_per_task * batch_input_price +
    ↪est_output_per_task * batch_output_price)
realtime_cost = n_tasks * (est_input_per_task * realtime_input_price +
    ↪est_output_per_task * realtime_output_price)

print(f"\nCost comparison (estimated):")
print(f"  Real-time: ${realtime_cost:.4f}")
print(f"  Batch:      ${batch_cost:.4f}")
print(f"  Savings:     ${realtime_cost - batch_cost:.4f} ({(1 - batch_cost/
    ↪realtime_cost)*100:.0f}%)")

```