

lab

April 3, 2026

1 Lesson 01 — How LLMs Work: Lab

In this lab you will: 1. Use `tiktoken` to explore how text is tokenised 2. Call the Claude API at different temperatures and compare outputs 3. Build a cost calculator function for API usage planning

Instructions: Fill in each # TODO cell. Run cells top to bottom. Check your work against `lab_guide.md`.

1.1 Setup

Run this cell first. It imports required libraries and loads your API key from `.env`.

```
[ ]: import os
import tiktoken
import anthropic
from dotenv import load_dotenv

# Load API key from .env file
load_dotenv()
api_key = os.getenv("ANTHROPIC_API_KEY")

if not api_key:
    raise ValueError("ANTHROPIC_API_KEY not found. Create a .env file with your API key.")

# Mask the key for display
masked = api_key[:8] + "..." + api_key[-4:]
print(f"API key loaded: {masked}")

# Initialise the Anthropic client
client = anthropic.Anthropic(api_key=api_key)
print("Anthropic client ready.")
```

1.2 Exercise 1: Tokenisation with tiktoken

Tokens are the atomic unit of text for LLMs. Understanding tokenisation helps you: - Predict API costs before making calls - Avoid accidentally exceeding context windows - Understand why some

prompts behave unexpectedly

We'll use `tiktoken`, OpenAI's open-source tokeniser library. The `cl100k_base` encoding is used by GPT-4-class models; we use it here as a practical proxy since Claude's tokeniser is not publicly released as a standalone library.

```
[ ]: # Load the cl100k_base encoding
# TODO: Use tiktoken.get_encoding() to load "cl100k_base" and assign it to `enc`
enc = None # replace with your code

print(f"Vocabulary size: {enc.n_vocab:,}")
```

```
[ ]: # Encode a simple string and inspect the token IDs
text = "Hello, world!"

# TODO: Use enc.encode() to get the token IDs for `text`
token_ids = None # replace with your code

print(f"String: {text!r}")
print(f"Token IDs: {token_ids}")
print(f"Number of tokens: {len(token_ids)}")

# TODO: Decode each token ID individually to see what text each token represents
# Hint: use enc.decode([token_id]) for each id in token_ids
tokens = None # replace with list comprehension

print(f"Tokens: {tokens}")
```

```
[ ]: # Compare token counts across different types of strings
strings_to_test = [
    "The quick brown fox",
    "def calculate_fibonacci(n):",
    "    ,    !", # Russian: "Hello, world!"
    "Hello",
    " Hello",      # note the leading space
    "2024",
    "20240101",
]

print(f"{'String':<35} {'Tokens':>8}")
print("-" * 45)

for s in strings_to_test:
    # TODO: Count the tokens for each string and print in the formatted table
    count = None # replace with enc.encode(s) and len()
    print(f"{s!r:<35} {count:>8}")
```

```
[ ]: # Dig deeper: show each token in a string as (id, text) pairs
def show_tokens(text: str, encoding) -> None:
    """Print each token in text as (id, decoded_text) pairs."""
    # TODO: Encode the text, then for each token ID print (id, decoded text)
    ids = encoding.encode(text)
    print(f"Input: {text!r} → {len(ids)} tokens")
    pairs = None # TODO: build list of (id, enc.decode([id])) tuples for each
    ↪ id in ids
    # Guard so students get a clear message if they run without completing the
    ↪ TODO:
    if pairs is None:
        raise NotImplementedError("Complete the TODO above before running this
    ↪ cell")
    for token_id, token_text in pairs:
        print(f" {token_id:>7} → {token_text!r}")

show_tokens("unbelievable", enc)
print()
show_tokens("GPT-4", enc)
```

1.3 Exercise 2: Temperature Comparison

Temperature controls how deterministic or creative the model's outputs are: - **Temperature 0**: always picks the most probable next token (greedy decoding) - **Temperature 1.0**: samples from the full distribution — more varied and creative

We'll send the same creative prompt at both temperatures and observe the difference.

```
[ ]: # Define a creative prompt that has no single "correct" answer
CREATIVE_PROMPT = "Write the opening sentence of a noir detective novel set in
    ↪ a cyberpunk city."

MODEL = "claude-3-5-haiku-20241022" # fast and cheap for experiments
MAX_TOKENS = 150
```

```
[ ]: def call_claude(prompt: str, temperature: float, model: str = MODEL, max_tokens:
    ↪ int = MAX_TOKENS) -> str:
    """Call the Claude API and return the response text."""
    # TODO: Use client.messages.create() to call the API
    # Parameters: model, max_tokens, temperature, messages=[{"role": "user",
    ↪ "content": prompt}]
    # Return: response.content[0].text
    pass # replace with your implementation
```

```
[ ]: # Call at temperature 0 - run this cell multiple times and note the consistency
print("=" * 60)
```

```
print("Temperature: 0 (deterministic)")
print("=" * 60)

# TODO: Call call_claude() with temperature=0
output_t0 = None # replace with function call
print(output_t0)
```

```
[ ]: # Call at temperature 1.0 - run this cell multiple times and note the variation
print("=" * 60)
print("Temperature: 1.0 (creative/varied)")
print("=" * 60)

# TODO: Call call_claude() with temperature=1.0
output_t1 = None # replace with function call
print(output_t1)
```

```
[ ]: # Run temperature=1.0 three more times to see the range of variation
print(f"Prompt: {CREATIVE_PROMPT!r}")
print()

# TODO: Loop 3 times, call the API at temperature=1.0, and print each output
↳with a run number
for i in range(3):
    # TODO: call and print
    pass
```

1.3.1 Reflection

In the cell below, write your observations (edit the markdown):

- Did temperature=0 produce identical outputs on repeated runs?
- How did the temperature=1.0 outputs differ from each other?
- For which tasks would you choose temperature=0? Temperature=1.0?

Your observations here:

(double-click to edit this cell)

1.4 Exercise 3: Cost Calculator

API costs are real and can escalate quickly. Building cost estimation into your workflow from the start is a good habit.

Pricing is per million tokens (input and output are priced separately).

```
[ ]: # Prices last verified 2024-10-22 - check https://www.anthropic.com/pricing for
↳updates
```

```
# Current pricing (USD per million tokens) - check docs.anthropic.com for
↳updates
MODEL_PRICING = {
    "claude-3-5-sonnet-20241022": {"input": 3.00, "output": 15.00},
    "claude-3-5-haiku-20241022": {"input": 0.80, "output": 4.00},
    "claude-3-opus-20240229": {"input": 15.00, "output": 75.00},
}
```

```
[ ]: def count_tokens(text: str) -> int:
    """Count the number of tokens in a string using cl100k_base encoding."""
    # TODO: Use enc.encode() and return the count
    pass # replace with your implementation
```

```
[ ]: def estimate_cost(
    prompt: str,
    expected_output_tokens: int,
    model: str = "claude-3-5-sonnet-20241022"
) -> dict:
    """
    Estimate the API cost for a single call.

    Returns a dict with keys:
        input_tokens, output_tokens,
        input_cost_usd, output_cost_usd, total_cost_usd
    """
    # TODO:
    # 1. Count input tokens using count_tokens()
    # 2. Look up the model's pricing from MODEL_PRICING
    # 3. Calculate input_cost = input_tokens / 1_000_000 * input_price
    # 4. Calculate output_cost = expected_output_tokens / 1_000_000 *
    ↳output_price
    # 5. Return the dict
    pass # replace with your implementation
```

```
[ ]: # Test with a realistic summarisation scenario
sample_article = """
Artificial intelligence researchers have long sought to create systems that can
↳reason
about the world in ways that mirror human cognition. Recent advances in large
↳language
models have brought this goal closer to reality, with systems like GPT-4,
↳Claude, and
Gemini demonstrating impressive capabilities across a wide range of tasks.
↳However,
significant challenges remain, including issues of hallucination, bias, and the
↳energy
```

```

cost of training and running these models at scale. This article examines the_
↳current
state of the field and the road ahead.
"""

# TODO: Call estimate_cost() with the article, 100 expected output tokens, and_
↳the Sonnet model
result = None # replace with function call

print("=" * 45)
print("Cost Estimate")
print("=" * 45)
# TODO: Print input_tokens, output_tokens, input_cost_usd, output_cost_usd,_
↳total_cost_usd
# Format costs to 6 decimal places

```

```

[ ]: def monthly_cost(
    prompt: str,
    expected_output_tokens: int,
    model: str,
    calls_per_day: int
) -> dict:
    """
    Calculate monthly API cost given a daily call volume.

    Returns a dict with keys:
        calls_per_month, monthly_cost_usd, cost_per_call_usd
    """
    # TODO:
    # 1. Call estimate_cost() to get per-call cost
    # 2. Calculate calls_per_month = calls_per_day * 30
    # 3. Calculate monthly_cost_usd = cost_per_call * calls_per_month
    # 4. Return the dict
    pass # replace with your implementation

```

```

[ ]: # Compare monthly cost across all models for the same workload
CALLS_PER_DAY = 100

print(f"Workload: {CALLS_PER_DAY} calls/day, same prompt, 100 output tokens")
print()
print(f"{'Model':<40} {'Monthly Cost':>14}")
print("-" * 56)

# TODO: Loop over MODEL_PRICING.keys(), call monthly_cost() for each,
# and print the model name and monthly cost formatted as $X.XX
for model_name in MODEL_PRICING:
    pass # replace with your implementation

```

1.5 Wrap-up

Congratulations on completing the lab! Before finishing:

1. Run **Kernel** → **Restart & Run All** to verify everything executes without errors.
2. Compare your implementations with `lab_answer.ipynb`.
3. Review the self-check criteria in `lab_guide.md`.
4. When you're ready, work through `homework.md`.