

lab_answer

April 3, 2026

1 Lesson 01 — How LLMs Work: Lab (Answer Key)

Note: This is the fully solved version of `lab.ipynb`. Try the exercises yourself first!
All TODO cells are filled in with working, commented code.

1.1 Setup

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

# Load API key from .env file in project root
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_
↳key.")

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

# Initialise the Anthropic client once; reuse throughout the notebook
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 costs, avoid context window overflow, and debug unexpected model behaviour.

```
[ ]: 
```

```

# Load the cl100k_base encoding - this is an OpenAI encoding (used by
↳GPT-4-class models).
# Claude's tokeniser is not publicly released as a standalone library, so we
↳use cl100k_base
# as a practical proxy. Token counts for Claude may differ slightly in practice.
# This encoding uses Byte-Pair Encoding (BPE) with a 100k vocabulary.
enc = tiktoken.get_encoding("cl100k_base")

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

```

```

[ ]: text = "Hello, world!"

# encode() converts the string to a list of integer token IDs
token_ids = enc.encode(text)

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

# Decode each token ID individually to see the text it represents
# Note: enc.decode() expects a LIST even for a single token
tokens = [enc.decode([tid]) for tid in token_ids]

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 - different token!
    "2024",
    "20240101",
]

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

for s in strings_to_test:
    # enc.encode() returns the token ID list; len() gives the count
    count = len(enc.encode(s))
    print(f"{s!r:<35} {count:>8}")

```

```

[ ]: def show_tokens(text: str, encoding) -> None:
    """Print each token in text as (id, decoded_text) pairs."""
    ids = encoding.encode(text)

```

```

print(f"Input: {text!r} → {len(ids)} tokens")
# Build a list of (token_id, decoded_text) tuples
pairs = [(tid, enc.decode([tid])) for tid in ids]
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 - **Temperature 1.0:** samples from the full probability distribution

```

[ ]: 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."""
    response = client.messages.create(
        model=model,
        max_tokens=max_tokens,
        temperature=temperature,
        messages=[
            {"role": "user", "content": prompt}
        ]
    )
    # response.content is a list of content blocks; [0] is the first (and_
    ↪ usually only) block
    return response.content[0].text

```

```

[ ]: # Run this cell multiple times - output should be consistent at temperature=0
print("=" * 60)
print("Temperature: 0 (deterministic)")
print("=" * 60)

output_t0 = call_claude(CREATIVE_PROMPT, temperature=0)
print(output_t0)

```

```

[ ]: # Run this cell multiple times - output should vary at temperature=1.0
print("=" * 60)

```

```

print("Temperature: 1.0 (creative/varied)")
print("=" * 60)

output_t1 = call_claude(CREATIVE_PROMPT, temperature=1.0)
print(output_t1)

```

```

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

for i in range(3):
    result = call_claude(CREATIVE_PROMPT, temperature=1.0)
    print(f"--- Run {i + 1} ---")
    print(result)
    print()

```

1.3.1 Observations

- **Temperature 0:** Produces consistent output on repeated runs (though not guaranteed to be 100% identical due to hardware floating-point differences, it will be very close or identical).
- **Temperature 1.0:** Each run explores different creative directions — different word choices, sentence structures, narrative perspectives.
- **When to use temperature 0:** Code generation, data extraction, factual Q&A, structured output, any task where consistency is important.
- **When to use temperature 1.0:** Creative writing, brainstorming, generating multiple variations, any task where diversity is valuable.

1.4 Exercise 3: Cost Calculator

API costs are priced per million tokens. Output tokens are significantly more expensive than input tokens (typically 5x). Building a cost calculator helps you make informed decisions about model selection and prompt design.

```

[ ]: # 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."""
    # enc is the cl100k_base encoding loaded in Exercise 1

```

```
return len(enc.encode(text))
```

```
[ ]: 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
    """
    if model not in MODEL_PRICING:
        raise ValueError(f"Unknown model: {model}. Available: {
↪list(MODEL_PRICING.keys())}")

    # Count input tokens using our tokeniser
    input_tokens = count_tokens(prompt)

    # Retrieve per-million-token pricing for the chosen model
    pricing = MODEL_PRICING[model]

    # Calculate cost: (tokens / 1_000_000) * price_per_million
    input_cost = (input_tokens / 1_000_000) * pricing["input"]
    output_cost = (expected_output_tokens / 1_000_000) * pricing["output"]

    return {
        "input_tokens": input_tokens,
        "output_tokens": expected_output_tokens,
        "input_cost_usd": input_cost,
        "output_cost_usd": output_cost,
        "total_cost_usd": input_cost + output_cost,
    }
```

```
[ ]: 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.
"""

result = estimate_cost(
    prompt=sample_article,
    expected_output_tokens=100,
    model="claude-3-5-sonnet-20241022"
)

print("=" * 45)
print("Cost Estimate")
print("=" * 45)
print(f"Model:          claude-3-5-sonnet-20241022")
print(f"Input tokens:    {result['input_tokens']}")
print(f"Output tokens:    {result['output_tokens']} (estimated)")
print(f"Input cost:       ${result['input_cost_usd']:.6f}")
print(f"Output cost:      ${result['output_cost_usd']:.6f}")
print(f"Total per call:   ${result['total_cost_usd']:.6f}")

```

```

[ ]: 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
    """

    # Get the per-call cost breakdown
    per_call = estimate_cost(prompt, expected_output_tokens, model)

    # Approximate month as 30 days
    calls_per_month = calls_per_day * 30
    total_monthly = per_call["total_cost_usd"] * calls_per_month

    return {
        "calls_per_month": calls_per_month,
        "monthly_cost_usd": total_monthly,
        "cost_per_call_usd": per_call["total_cost_usd"],
    }

```

```
[ ]: CALLS_PER_DAY = 100

print(f"Workload: {CALLS_PER_DAY} calls/day, same prompt_
↳ (~{count_tokens(sample_article)} tokens input), 100 output tokens")
print()
print(f"{'Model':<40} {'Cost/Call':>12} {'Monthly Cost':>14}")
print("-" * 68)

for model_name in MODEL_PRICING:
    mc = monthly_cost(
        prompt=sample_article,
        expected_output_tokens=100,
        model=model_name,
        calls_per_day=CALLS_PER_DAY
    )
    print(
        f"{'model_name':<40} "
        f"${mc['cost_per_call_usd']:>10.6f} "
        f"${mc['monthly_cost_usd']:>12.2f}"
    )
```

1.5 Wrap-up

You've completed all three exercises. Key takeaways:

1. **Tokenisation:** Text is not characters or words — it's subword tokens. English prose tokenises at ~1.3 tokens/word; code and non-Latin scripts tokenise less efficiently.
2. **Temperature:** Use 0 for tasks needing consistency (code, structured data, facts) and ~1.0 for tasks needing creative variation. Most production use cases sit around 0–0.7.
3. **Cost:** Output tokens are significantly more expensive than input. The choice between models can mean a 20x cost difference on the same task. Always build cost estimation into your planning.

Next: **Lesson 02 — Prompt Engineering Fundamentals**