

lab

April 3, 2026

1 Lesson 04 — Advanced Prompting

1.1 Lab Notebook

Exercises: 1. Chain-of-Thought — compare CoT vs direct answer on a multi-step word problem
2. Few-Shot Classifier — classify support tickets by urgency using 4 labeled examples
3. XML Extraction — extract structured info from meeting notes using XML tags

```
[ ]: # Setup - run this cell first
import json
from dotenv import load_dotenv
import anthropic

load_dotenv()
client = anthropic.Anthropic()
MODEL = "claude-3-5-haiku-20241022"

def call_claude(prompt: str, system: str = None, max_tokens: int = 1024) -> str:
    kwargs = dict(model=MODEL, max_tokens=max_tokens,
                  messages=[{"role": "user", "content": prompt}])
    if system:
        kwargs["system"] = system
    msg = client.messages.create(**kwargs)
    return msg.content[0].text

print("Setup complete.")
```

1.2 Exercise 1 — Chain-of-Thought vs Direct Answer

Compare Claude's accuracy on a multi-step word problem with and without chain-of-thought prompting.

```
[ ]: # Word problem - do not change
WORD_PROBLEM = """A train leaves City A at 9:00 AM traveling at 75 mph toward
↳City B.
A second train leaves City B at 10:30 AM traveling toward City A at 55 mph.
The distance between City A and City B is 280 miles.
```

```

At what time do the two trains meet, and how far from City A is the meeting_
↳point?"""

print("Problem loaded.")
print("Expected answer: They meet at 12:15 PM, approximately 243.75 miles from_
↳City A.")

```

```

[ ]: # TODO: implement solve_with_cot(problem) and solve_without_cot(problem)

def solve_without_cot(problem: str) -> str:
    """Ask Claude for a direct, concise answer with no step-by-step reasoning.
    ↳"""
    # TODO: implement this function
    # Hint: append something like "Answer concisely with just the final answer."
    raise NotImplementedError("Complete this TODO")

def solve_with_cot(problem: str) -> str:
    """Ask Claude to think step by step before giving the answer."""
    # TODO: implement this function
    # Hint: append "Let's think step by step." to the problem
    raise NotImplementedError("Complete this TODO")

```

```

[ ]: # Run comparison 3 times each
print("=" * 60)
print("WITHOUT CoT (3 runs)")
print("=" * 60)
for i in range(3):
    print(f"\nRun {i+1}:")
    print(solve_without_cot(WORD_PROBLEM))

print("\n" + "=" * 60)
print("WITH CoT (3 runs)")
print("=" * 60)
for i in range(3):
    print(f"\nRun {i+1}:")
    print(solve_with_cot(WORD_PROBLEM))

```

Reflection:

1. Did CoT produce more accurate answers? On which runs?
(your answer here)
2. Did CoT ever produce a longer but still incorrect answer?
(your answer here)

1.3 Exercise 2 — Few-Shot Urgency Classifier

Build a few-shot classifier for customer support ticket urgency: - **CRITICAL** — service down or data loss risk - **HIGH** — major feature broken, significant business impact
- **MEDIUM** — partial impact, workaround available - **LOW** — cosmetic issue or feature request

```
[ ]: # TODO: build FEW_SHOT_EXAMPLES string with 4 labeled examples (one per urgency,
      ↪level)
      # Format:
      #   Ticket: [text]
      #   Urgency: [CRITICAL/HIGH/MEDIUM/LOW]

FEW_SHOT_EXAMPLES = None # TODO: 4 labeled ticket examples as a string

if FEW_SHOT_EXAMPLES is None:
    raise NotImplementedError("Complete this TODO")
```

```
[ ]: # TODO: implement classify_ticket(text)
      # Must return exactly one of: "CRITICAL", "HIGH", "MEDIUM", "LOW"

def classify_ticket(text: str) -> str:
    """Classify a support ticket by urgency using few-shot prompting.
    Returns one of: CRITICAL, HIGH, MEDIUM, LOW
    """
    # TODO: build prompt combining FEW_SHOT_EXAMPLES and the new ticket
    # Ask Claude to output ONLY the urgency label, no explanation
    raise NotImplementedError("Complete this TODO")
```

```
[ ]: # Test tickets with expected labels
TEST_TICKETS = [
    "The entire API is returning 500 errors for all customers since 14:00 UTC.↵
    ↪Production is down.",
    "Our dashboard export to CSV is producing garbled characters for non-ASCII↵
    ↪names.",
    "Would be great if the dark mode also applied to the mobile app - currently↵
    ↪only on desktop.",
    "I forgot my password and the reset email is not arriving. I need to log in↵
    ↪for a meeting in 10 minutes.",
    "Some users report occasional slow loading of the reports page - not↵
    ↪consistent, maybe 1 in 10 loads.",
]

EXPECTED = ["CRITICAL", "HIGH", "LOW", "HIGH", "MEDIUM"]

print("Running classifier on 5 test tickets...\n")
correct = 0
for ticket, expected in zip(TEST_TICKETS, EXPECTED):
```

```

result = classify_ticket(ticket).strip().upper()
match = result == expected
correct += int(match)
status = "CORRECT" if match else f"WRONG (expected {expected})"
print(f"[{result:8}] {status}")
print(f"  {ticket[:80]}...")
print()

print(f"Accuracy: {correct}/{len(TEST_TICKETS)}")

```

1.4 Exercise 3 — XML-Structured Extraction

Extract structured information from meeting notes using XML tags to clearly separate document content from instructions and output schema.

```

[ ]: # Meeting notes document - do not change
MEETING_NOTES = """Quick sync - Product & Engineering
Tuesday, March 19th, 2024, 10am

Present: Priya (PM), Jake (Eng Lead), Mei (Design), Carlos (Backend)

We kicked off with Priya walking through the Q2 roadmap. The team agreed to
prioritize the mobile redesign over the analytics dashboard - that's decided.
Jake will need to prepare a technical spec document for the mobile work by next
Friday (March 29). Mei is already working on mockups and will share them in_
↳Figma
by end of this week (March 22). Carlos pointed out we have a performance issue
in the search endpoint and he'll fix it by March 21 before it becomes_
↳customer-facing.

We also discussed the API rate limiting feature. No decision yet - needs more
scoping. Priya to write a one-pager on requirements by March 25.

The team agreed to move the weekly standup from Monday to Tuesday going forward.
"""

print("Document loaded.")

```

```

[ ]: # TODO: implement extract_with_xml(document_text)
# The prompt must use XML tags: <document>, <instructions>, <output_format>
# Return a raw JSON string matching:
# {
#   "meeting_date": "string",
#   "attendees": ["string"],
#   "action_items": [{"owner": "string", "task": "string", "deadline": "string_
↳or null"}],

```

```

# "decisions": ["string"]
# }

def extract_with_xml(document_text: str) -> str:
    """Extract structured info using XML-tagged prompt. Returns raw JSON string.
    ↪ """
    # TODO: build prompt with <document>, <instructions>, <output_format> XML
    ↪ tags
    raise NotImplementedError("Complete this TODO")

```

```

[ ]: # Run extraction and validate
raw_result = extract_with_xml(MEETING_NOTES)
print("Raw response:")
print(raw_result)
print()

try:
    result = json.loads(raw_result)
    print("JSON parse: SUCCESS")
    print(f" Meeting date: {result.get('meeting_date')}")
    print(f" Attendees: {result.get('attendees')}")
    print(f" Action items ({len(result.get('action_items', []))}):")
    for item in result.get('action_items', []):
        print(f" - {item.get('owner')}: {item.get('task')} (by {item.
    ↪ get('deadline')})")
    print(f" Decisions: {result.get('decisions')}")
    assert len(result.get('action_items', [])) >= 1, "Expected at least 1
    ↪ action item"
    print("\nValidation: PASSED")
except json.JSONDecodeError as e:
    print(f"JSON parse FAILED: {e}")

```