

lab

April 3, 2026

1 Lesson 05 — Context Windows & Long Documents

In this lab you will implement three practical skills for working with documents that are long or expensive to process in a single API call:

1. **Chunking** — split text into overlapping fixed-size pieces
2. **Map-Reduce Summarisation** — summarise each chunk then combine
3. **Cost Comparison** — compare token usage between approaches

Before you start: make sure `demo/sample_document.txt` exists in the lesson folder (it should be a ~4000-word history of AI).

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

load_dotenv()

client = anthropic.Anthropic()

print('Setup complete. Client ready.')
```

```
[ ]: # Load the sample document
text = pathlib.Path('demo/sample_document.txt').read_text(encoding='utf-8')
print(f'Document loaded: {len(text)} characters, {len(text.split())} words')
```

1.1 Exercise 1 — Chunking

Implement `chunk_text` which splits a document into overlapping word-based chunks.

Requirements: - Split text into words - Create chunks of approximately `chunk_size` words - Consecutive chunks should share approximately `overlap` words at their boundaries - Return a list of strings (each string is one chunk's text)

Example: with `chunk_size=5`, `overlap=2` and input "a b c d e f g h i j" you should get roughly: - chunk 0: "a b c d e" - chunk 1: "d e f g h" (starts 3 words after chunk 0) - chunk 2: "g h i j" (starts 3 words after chunk 1)

```
[ ]: def chunk_text(text: str, chunk_size: int = 500, overlap: int = 50) -> list[str]:
    """Split text into overlapping chunks of approximately chunk_size words."""
    # TODO: split text into words, create chunks with overlap
    # Each chunk should have ~chunk_size words, consecutive chunks share ~overlap words
    raise NotImplementedError("Implement chunk_text")

chunks = chunk_text(text)
print(f"Created {len(chunks)} chunks")
print(f"First chunk preview: {chunks[0][:200]}...")
```

1.2 Exercise 2 — Map-Reduce Summarisation

Implement a two-stage summarisation pipeline:

1. **Map:** `summarize_chunk(chunk)` — summarise a single chunk in 2-3 sentences using the Claude API
2. **Reduce:** `map_reduce_summarize(chunks)` — summarise all chunks (the map step), then call Claude once more to combine the chunk summaries into a final coherent summary

Use `claude-haiku-4-5` for cost efficiency. Set `max_tokens=150` for chunk summaries and `max_tokens=400` for the final combine step.

```
[ ]: def summarize_chunk(chunk: str) -> str:
    """Summarise a single chunk in 2-3 sentences."""
    # TODO: call Claude API to summarize the chunk
    raise NotImplementedError("Implement summarize_chunk")

def map_reduce_summarize(chunks: list[str]) -> str:
    """Summarise all chunks then combine into final summary."""
    # TODO: 1) summarize each chunk, 2) combine chunk summaries, 3) final summary pass
    raise NotImplementedError("Implement map_reduce_summarize")

final_summary = map_reduce_summarize(chunks)
print(final_summary)
```

1.3 Exercise 3 — Cost Comparison

Calculate and compare the token cost of: - **Single-pass:** sending the full document + a summarise instruction in one API call - **Map-reduce:** the sum of all chunk summarisation calls plus the final combine call

Use `tiktoken (cl100k_base)` to count tokens. Use these Haiku pricing figures: - Input: \$0.25 per 1 000 000 tokens - Output: \$1.25 per 1 000 000 tokens

Print a comparison table showing: Approach | Input Tokens | Output Tokens | Estimated Cost (\$)

```
[ ]: enc = tiktoken.get_encoding("cl100k_base") # proxy tokenizer

def count_tokens(text: str) -> int:
    return len(enc.encode(text))

# TODO: calculate token count for single-pass approach (full text + prompt +
#       ↪overhead)
# TODO: calculate total token count for map-reduce approach (sum of all chunk +
#       ↪calls + combine call)
# TODO: print comparison table: approach / input tokens / output tokens /
#       ↪estimated cost
raise NotImplementedError("Complete the token comparison")
```