

lab

April 3, 2026

1 Lab 11 — Multi-Agent Systems

Build a multi-agent pipeline with an orchestrator, researcher, and writer.

Exercises: 1. Build an Agent class and create Researcher + Writer agents 2. Implement an orchestrator function 3. End-to-end test with message traces

```
[ ]: # Setup
import json
from dataclasses import dataclass, field
from dotenv import load_dotenv
import anthropic

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

```
[ ]: # Mock Data

MOCK_SEARCH_DB = {
    "python async": [
        {"title": "Python asyncio Guide", "url": "https://example.com/asyncio",
         "snippet": "asyncio is a library for writing concurrent code using
↪ async/await syntax."},
        {"title": "Async Patterns in Python", "url": "https://example.com/
↪ async-patterns",
         "snippet": "Common patterns include gather, TaskGroup, and async
↪ generators."}
    ],
    "python type hints": [
        {"title": "Type Hints in Python", "url": "https://example.com/
↪ type-hints",
         "snippet": "Type hints improve code readability and enable static
↪ analysis with mypy."},
    ],
    "python lists vs tuples": [
        {"title": "Lists vs Tuples", "url": "https://example.com/lists-tuples",
```

```

        "snippet": "Lists are mutable; tuples are immutable and hashable."}
    ],
    "python web frameworks": [
        {"title": "Python Web Frameworks", "url": "https://example.com/
↳web-frameworks",
        "snippet": "Django, Flask, and FastAPI are the most popular Python web
↳frameworks."},
    ],
}

MOCK_URL_DB = {
    "https://example.com/asyncio": "Python asyncio Guide\n\nasyncio is Python's
↳built-in library for async programming.\nKey concepts: coroutines, event
↳loop, tasks, gather(), async with/for.",
    "https://example.com/async-patterns": "Common Async Patterns\n\n1. Fan-out/
↳fan-in with asyncio.gather()\n2. Structured concurrency with TaskGroup\n3.
↳Async generators for streaming",
    "https://example.com/type-hints": "Type Hints in Python\n\ndef greet(name:
↳str) -> str\nOptional, container types, TypeAlias, NewType.",
    "https://example.com/lists-tuples": "Lists vs Tuples\n\nLists: mutable, []
↳syntax. Tuples: immutable, () syntax, hashable.\nTuples use less memory.",
    "https://example.com/web-frameworks": "Python Web Frameworks\n\nDjango:
↳batteries-included. Flask: lightweight. FastAPI: modern, async-native.",
}

print(f"Mock DB: {len(MOCK_SEARCH_DB)} search entries, {len(MOCK_URL_DB)} URLs")

```

```

[ ]: # Shared data classes

@dataclass
class AgentResult:
    status: str # "success", "partial", "error"
    content: str
    error: str = ""
    steps_used: int = 0

print("AgentResult defined.")

```

1.1 Exercise 1 — Agent Class + Two Agents

Build a reusable Agent class with a ReAct loop, then create Researcher and Writer instances.

```

[ ]: # TODO: Implement the Agent class
# - __init__(self, name, system_prompt, tools, tool_executor)
# - run(self, task, max_steps=10) -> AgentResult

```

```

# The run method should implement a ReAct loop using Claude's tool-use API.
# It should return an AgentResult with status, content, and steps_used.

# TODO: Define RESEARCHER_TOOLS (search, read_url schemas)
# TODO: Define researcher_execute_tool(name, input) -> str
# TODO: Create researcher = Agent("researcher", ..., RESEARCHER_TOOLS,
    ↪ researcher_execute_tool)

# TODO: Create writer = Agent("writer", ..., tools=[], tool_executor=None)
# The writer has no tools - it just generates text from a prompt.

raise NotImplementedError("Exercise 1: implement Agent class and create two
    ↪ agents")

```

```

[ ]: # Test Exercise 1
result = researcher.run("Find information about Python async programming.")
print(f"Researcher: status={result.status}, steps={result.steps_used}")
print(f"Content: {result.content[:200]}...")

result = writer.run("Write a short paragraph about Python based on these facts:
    ↪ Python is a popular language.")
print(f"\nWriter: status={result.status}, steps={result.steps_used}")
print(f"Content: {result.content[:200]}...")

```

1.2 Exercise 2 — Orchestrator Function

Implement `orchestrate(task, researcher, writer)` that delegates research then writing.

```

[ ]: # TODO: Implement orchestrate(task, researcher, writer) -> dict
# Steps:
# 1. Send task to researcher with instruction to gather facts
# 2. Pass researcher's output to writer with instruction to produce polished
    ↪ text
# 3. Return {"final_answer": str, "trace": list[dict]}
# where trace entries have: agent, instruction, status, steps

raise NotImplementedError("Exercise 2: implement orchestrate function")

```

```

[ ]: # Test Exercise 2
result = orchestrate(
    "Write a technical overview of Python async programming.",
    researcher, writer
)
print("FINAL ANSWER:")
print(result["final_answer"][:500])
print("\nTRACE:")

```

```
for entry in result["trace"]:
    print(f"  {entry['agent']}: status={entry['status']},  
    ↳steps={entry['steps']}")
```

1.3 Exercise 3 — End-to-End Test

Run 3 tasks through the pipeline and compare traces.

```
[ ]: # TODO: Run 3 tasks through orchestrate() and collect results
# Tasks:
# 1. "Write a technical overview of Python async programming."
# 2. "Explain the differences between Python lists and tuples for a beginner,  
    ↳audience."
# 3. "Create a guide to choosing a Python web framework."
#
# For each task, print the trace, a snippet of the final answer, and total,  
    ↳steps.

raise NotImplementedError("Exercise 3: run end-to-end tests")
```