

lab_answer

April 3, 2026

1 Lab 11 — Multi-Agent Systems (ANSWERS)

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

```

[ ]: # Agent class

class Agent:
    """Reusable LLM agent with optional tools and a ReAct loop."""

```

```

def __init__(self, name: str, system_prompt: str, tools: list = None,
              tool_executor=None):
    self.name = name
    self.system_prompt = system_prompt
    self.tools = tools or []
    self.tool_executor = tool_executor

def run(self, task: str, max_steps: int = 10) -> AgentResult:
    messages = [{"role": "user", "content": task}]
    steps = 0

    for step in range(max_steps):
        kwargs = dict(
            model=MODEL,
            max_tokens=1024,
            system=self.system_prompt,
            messages=messages,
        )
        if self.tools:
            kwargs["tools"] = self.tools

        response = client.messages.create(**kwargs)

        # Collect text and tool calls
        tool_results = []
        final_text = None

        for block in response.content:
            if hasattr(block, "text") and block.text:
                final_text = block.text
            elif block.type == "tool_use" and self.tool_executor:
                steps += 1
                result = self.tool_executor(block.name, block.input)
                tool_results.append({
                    "type": "tool_result",
                    "tool_use_id": block.id,
                    "content": result,
                })

        if response.stop_reason == "end_turn":
            return AgentResult(
                status="success",
                content=final_text or "",
                steps_used=steps
            )

    messages.append({"role": "assistant", "content": response.content})

```

```

        messages.append({"role": "user", "content": tool_results})

    return AgentResult(
        status="partial",
        content=final_text or "",
        error="max steps reached",
        steps_used=steps
    )

# Tool schemas

RESEARCHER_TOOLS = [
    {
        "name": "search",
        "description": "Search for information on a topic.",
        "input_schema": {
            "type": "object",
            "properties": {
                "query": {"type": "string", "description": "The search query."}
            },
            "required": ["query"]
        }
    },
    {
        "name": "read_url",
        "description": "Read the full content of a URL.",
        "input_schema": {
            "type": "object",
            "properties": {
                "url": {"type": "string", "description": "The URL to read."}
            },
            "required": ["url"]
        }
    },
]

def researcher_execute_tool(tool_name: str, tool_input: dict) -> str:
    if tool_name == "search":
        query = tool_input["query"].lower()
        results = []
        for key, entries in MOCK_SEARCH_DB.items():
            if any(word in key for word in query.split()):
                results.extend(entries)
        return json.dumps(results, indent=2) if results else "No results found."
    elif tool_name == "read_url":

```

```

        url = tool_input["url"]
        return MOCK_URL_DB.get(url, f"URL not found: {url}")
    return f"Unknown tool: {tool_name}"

# Create agents

researcher = Agent(
    name="researcher",
    system_prompt="You are a research agent. Use your search and read_url tools,
↳to find factual information. Be thorough and cite your sources.",
    tools=RESEARCHER_TOOLS,
    tool_executor=researcher_execute_tool,
)

writer = Agent(
    name="writer",
    system_prompt="You are a technical writer. Given research notes or a brief,
↳write clear, accurate, and well-structured prose. Include specific details
↳and examples from the provided material.",
)

print("Agents created: researcher, writer")

```

```

[ ]: # 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[:300]}...")

print()
result = writer.run("Write a short paragraph about Python based on these facts:
↳Python is a popular language used for web dev, data science, and automation.
↳")
print(f"Writer: status={result.status}, steps={result.steps_used}")
print(f"Content: {result.content[:300]}...")

```

1.2 Exercise 2 — Orchestrator Function

```

[ ]: def orchestrate(task: str, researcher: Agent, writer: Agent) -> dict:
    """Orchestrate a research-then-write pipeline."""
    trace = []

    # Step 1: Research
    research_instruction = f"Research the following topic and provide detailed
↳facts and findings:\n\n{task}"

```

```

research_result = researcher.run(research_instruction)
trace.append({
    "agent": "researcher",
    "instruction": research_instruction[:100] + "...",
    "status": research_result.status,
    "steps": research_result.steps_used,
})

# Step 2: Write
write_instruction = (
    f"Using the following research, write a polished response to this task:␣
↪{task}\n\n"
    f"Research findings:\n{research_result.content}"
)
write_result = writer.run(write_instruction)
trace.append({
    "agent": "writer",
    "instruction": write_instruction[:100] + "...",
    "status": write_result.status,
    "steps": write_result.steps_used,
})

return {
    "final_answer": write_result.content,
    "trace": trace,
}

print("orchestrate() defined.")

```

```

[ ]: # 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

```
[ ]: tasks = [
    "Write a technical overview of Python async programming.",
    "Explain the differences between Python lists and tuples for a beginner_
    ↪audience.",
    "Create a guide to choosing a Python web framework.",
]

all_results = []
for i, task in enumerate(tasks):
    print(f"\n{'#' * 60}")
    print(f"  TASK {i+1}: {task[:60]}")
    print(f"{'#' * 60}")
    result = orchestrate(task, researcher, writer)
    all_results.append(result)
    print(f"\nAnswer preview: {result['final_answer'][:200]}...")
    print("Trace:")
    total_steps = 0
    for entry in result["trace"]:
        print(f"  {entry['agent']}: {entry['status']} ({entry['steps']} steps)")
        total_steps += entry["steps"]
    print(f"  Total steps: {total_steps}")

# Summary
print("\n" + "=" * 60)
print("SUMMARY")
print("=" * 60)
for i, r in enumerate(all_results):
    total = sum(e["steps"] for e in r["trace"])
    statuses = [e["status"] for e in r["trace"]]
    print(f"Task {i+1}: steps={total}, statuses={statuses},_
    ↪answer_len={len(r['final_answer'])}")
```