

lab__answer

April 3, 2026

1 Lab 10 — Building Agents (ANSWERS)

Build a research agent with mock tools using the ReAct pattern.

Exercises: 1. Build 4 mock tools (search, read_url, save_note, get_notes) 2. Implement the full ReAct agent loop 3. Test with 3 different task types and analyse step counts

```
[ ]: # Setup
import json
from dotenv import load_dotenv
import anthropic

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

```
[ ]: # Mock Data (provided)

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."},
        {"title": "Advanced Typing", "url": "https://example.com/
↪ advanced-typing",
         "snippet": "Protocol, TypeVar, and ParamSpec enable advanced generic
↪ patterns."}
    ]
}
```

```

],
    "python lists vs tuples": [
        {"title": "Lists vs Tuples", "url": "https://example.com/lists-tuples",
         "snippet": "Lists are mutable sequences; tuples are immutable. Tuples
are hashable and slightly faster."}
    ],
    "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."},
        {"title": "FastAPI Tutorial", "url": "https://example.com/fastapi",
         "snippet": "FastAPI is a modern, high-performance framework for
building APIs with Python 3.7+."}
    ],
    "python decorators": [
        {"title": "Python Decorators", "url": "https://example.com/decorators",
         "snippet": "Decorators modify function behavior using the @decorator
syntax."}
    ],
}

```

```

MOCK_URL_DB = {
    "https://example.com/asyncio": """Python asyncio Guide\n\nasyncio is
Python's built-in library for asynchronous programming. Key concepts:\n-
Coroutines: defined with async def, paused with await\n- Event loop: runs
coroutines concurrently\n- Tasks: wrap coroutines for concurrent
execution\n- gather(): run multiple coroutines concurrently\n- async with /
async for: async context managers and iterators""",
    "https://example.com/async-patterns": """Common Async Patterns\n\n1.
Fan-out/fan-in with asyncio.gather()\n2. Structured concurrency with
TaskGroup (Python 3.11+)\n3. Async generators for streaming data\n4.
Semaphores for rate limiting\n5. Queue-based producer/consumer patterns""",
    "https://example.com/type-hints": """Type Hints in Python\n\nBasic
annotations: def greet(name: str) -> str\nOptional types: Optional[str] or
str | None (3.10+)\nContainer types: list[int], dict[str, Any]\nCustom types:
TypeAlias, NewType\nRuntime checking: isinstance() still works, hints are
not enforced""",
    "https://example.com/advanced-typing": """Advanced Typing
Patterns\n\nProtocol: structural subtyping (duck typing with type
safety)\nTypeVar: generic functions and classes\nParamSpec: preserving
parameter types in decorators\nConcatenate: adding parameters to callable
types\nOverload: multiple function signatures""",
}

```

```

    "https://example.com/lists-tuples": """Lists vs Tuples in Python\n\nLists:
↳mutable, use [] syntax, support append/extend/pop\nTuples: immutable, use ()
↳syntax, hashable, can be dict keys\nPerformance: tuples are slightly faster
↳for iteration and access\nMemory: tuples use less memory than equivalent
↳lists\nUse lists when you need to modify the collection; use tuples for
↳fixed data."""
    "https://example.com/web-frameworks": """Python Web Frameworks
↳Overview\n\nDjango: full-featured, batteries-included, ORM, admin
↳panel\nFlask: lightweight, flexible, great for microservices\nFastAPI:
↳modern, async-native, auto-generated OpenAPI docs\n\nChoose Django for large
↳apps, Flask for simplicity, FastAPI for APIs."""
    "https://example.com/fastapi": """FastAPI Tutorial\n\nFastAPI uses Python
↳type hints for request validation.\nKey features: automatic docs, dependency
↳injection, async support.\nInstall: pip install fastapi uvicorn\nRun:
↳uvicorn main:app --reload"""
    "https://example.com/decorators": """Python Decorators\n\nA decorator is a
↳function that takes a function and returns a modified function.\nCommon uses:
↳ logging, caching, authentication, retry logic.\n@functools.wraps preserves
↳the original function's metadata.\nClass-based decorators use __call__
↳method."""
}

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

```

1.1 Exercise 1 — Build Mock Tools

```

[ ]: # Agent notes storage
agent_notes = {}

# Tool schemas for Claude's API
TOOLS = [
    {
        "name": "search",
        "description": "Search for information on a topic. Returns a list of
↳results with titles, URLs, and snippets.",
        "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. Returns the page text.",
        "input_schema": {
            "type": "object",
            "properties": {
                "url": {"type": "string", "description": "The URL to read."}
            },
            "required": ["url"]
        }
    },
    {
        "name": "save_note",
        "description": "Save a note with a key for later reference.",
        "input_schema": {
            "type": "object",
            "properties": {
                "key": {"type": "string", "description": "A short label for the ↵
note."},
                "content": {"type": "string", "description": "The note content ↵
to save."}
            },
            "required": ["key", "content"]
        }
    },
    {
        "name": "get_notes",
        "description": "Retrieve all saved notes.",
        "input_schema": {
            "type": "object",
            "properties": {},
            "required": []
        }
    }
]

```

```

def execute_tool(tool_name: str, tool_input: dict) -> str:
    """Execute a mock tool and return the result as a string."""

    if tool_name == "search":
        query = tool_input["query"].lower()
        # Search by substring match across all keys
        results = []
        for key, entries in MOCK_SEARCH_DB.items():
            if any(word in key for word in query.split()):
                results.extend(entries)
        if results:
            return json.dumps(results, indent=2)

```

```

        return "No results found for: " + tool_input["query"]

    elif tool_name == "read_url":
        url = tool_input["url"]
        if url in MOCK_URL_DB:
            return MOCK_URL_DB[url]
        return f"URL not found: {url}"

    elif tool_name == "save_note":
        key = tool_input["key"]
        content = tool_input["content"]
        agent_notes[key] = content
        return f>Note saved under key '{key}' ({len(content)} chars)"

    elif tool_name == "get_notes":
        if not agent_notes:
            return "No notes saved yet."
        parts = []
        for k, v in agent_notes.items():
            parts.append(f"[{k}]: {v}")
        return "\n\n".join(parts)

    return f"Unknown tool: {tool_name}"

```

```

[ ]: # Test Exercise 1
print(execute_tool("search", {"query": "python async"}))
print()
print(execute_tool("save_note", {"key": "test", "content": "hello world"}))
print(execute_tool("get_notes", {}))
# Clean up test note
agent_notes.clear()

```

1.2 Exercise 2 — The ReAct Agent Loop

```

[ ]: SYSTEM_PROMPT = """You are a research agent. You have access to tools for_
    ↪ searching,
    reading URLs, and saving notes. Follow this process:
    1. Think about what information you need
    2. Use tools to gather information
    3. Save important findings as notes
    4. When you have enough information, provide your final answer

    Be systematic and thorough."""

```

```

def run_agent(task: str, tools: list, max_steps: int = 10) -> str:
    """Run a ReAct agent loop. Returns the final answer text."""
    messages = [{"role": "user", "content": task}]
    step_count = 0

    print(f"\n{'='*60}")
    print(f"TASK: {task}")
    print(f"{'='*60}")

    for step in range(max_steps):
        response = client.messages.create(
            model=MODEL,
            max_tokens=1024,
            system=SYSTEM_PROMPT,
            tools=tools,
            messages=messages,
        )

        print(f"\n Step {step + 1} | stop_reason={response.stop_reason} ")

        # Process content blocks
        tool_results = []
        final_text = None

        for block in response.content:
            if hasattr(block, "text") and block.text:
                print(f" THOUGHT: {block.text[:200]}{'...' if len(block.text) > 200 else ''}")
                final_text = block.text
            elif block.type == "tool_use":
                step_count += 1
                print(f" ACTION: {block.name}({json.dumps(block.input)[:100]})")
                result = execute_tool(block.name, block.input)
                print(f" OBSERVATION: {result[:150]}{'...' if len(result) > 150 else ''}")
                tool_results.append({
                    "type": "tool_result",
                    "tool_use_id": block.id,
                    "content": result,
                })

        # If done, return final text
        if response.stop_reason == "end_turn":
            print(f"\n Agent completed in {step + 1} rounds ({step_count} tool calls)")
            return final_text or "(no text)"

```

```

    # Feed tool results back
    messages.append({"role": "assistant", "content": response.content})
    messages.append({"role": "user", "content": tool_results})

    print(f"\n Max steps ({max_steps}) reached. Returning partial result.")
    return f"(max steps reached after {step_count} tool calls)"

```

```

[ ]: # Test Exercise 2
agent_notes.clear() # Reset notes
result = run_agent(
    "Research Python async programming. What are the key concepts? Save your_
    findings.",
    TOOLS
)
print("\nFINAL ANSWER:", result[:300])

```

1.3 Exercise 3 — Multi-Task Testing

```

[ ]: tasks = [
    "Research Python type hints. Find key concepts and save a summary.",
    "Compare Python lists vs tuples. Search for information about both and_
    explain the differences.",
    "Create a study plan for learning Python web development. Research the key_
    frameworks and tools needed.",
]

results = []
for i, task in enumerate(tasks):
    agent_notes.clear() # Fresh notes per task
    print(f"\n{'#' * 60}")
    print(f" TASK {i+1} of {len(tasks)}")
    print(f"{'#' * 60}")
    answer = run_agent(task, TOOLS, max_steps=10)
    results.append({
        "task": task,
        "answer": answer,
        "notes_saved": dict(agent_notes),
    })

# Summary table
print("\n" + "=" * 60)
print("SUMMARY")
print("=" * 60)
for i, r in enumerate(results):

```

```
completed = "(max steps)" not in r["answer"]
print(f"Task {i+1}: {'Completed' if completed else 'Incomplete'} | "
      f"Notes saved: {len(r['notes_saved'])} | "
      f"Answer length: {len(r['answer'])} chars")
```