2026 EDITION · PDF GUIDE

Claude Certified Architect Foundations Hands-On Labs

Hands-On Labs, Agentic AI Systems, MCP & Tool Integration Projects

PassITExams
https://passitexams.com/

★ FREE DOWNLOAD — UPDATED SEPTEMBER 2026

Table of Contents

Everything you need to build, ship, and certify.

SECTION 01

01 Introduction

• *Note: This is the complete hands-on lab content from the article you are reading + extra resources.*

The 2026 Agentic AI Shift

We are no longer in the era of "chatbots that answer questions." In 2026, the center of gravity in AI engineering has moved decisively toward **agentic systems** — Claude-powered architectures that plan, use tools, call other agents, write and execute code, and operate autonomously across multi-step workflows. The rise of the **Model Context Protocol (MCP)**, native tool-use, and multi-agent orchestration frameworks has turned "prompting" into a genuine engineering discipline, complete with its own patterns, anti-patterns, and certification paths.

This shift is why the **Claude Certified Architect Foundations** track exists in the first place. Employers are no longer hiring people who can write a clever prompt — they're hiring architects who can design systems: agents that hand off work to sub-agents, tool servers that expose safe and well-scoped capabilities, and orchestration layers that keep everything observable, testable, and production-ready.

Why This Guide Exists

Most exam-prep material is theory-heavy and lab-light. You read about agent architecture, memorize a few definitions, and then sit an exam without ever having actually **built** the thing you're being tested on. This guide flips that model.

Every section here is a **hands-on lab** — real code, real repos, real architecture diagrams — designed to take you from "I understand the concept" to "I have built and deployed this exact pattern." By the time you finish all four labs, you won't just be exam-ready. You'll have a working portfolio of multi-agent systems, MCP servers, and production agent deployments that you can show in interviews.

Before You Start

To get the most out of these labs, set up the following before Lab 1:

- **A Claude API key** — Sign up at the Anthropic Console and generate a key. Store it as an environment variable (`ANTHROPIC_API_KEY`), never hard-code it in your source files.
- **The starter repo** — Clone the companion starter repository referenced in each lab (see Section 6 for the full repo list). Each lab has a corresponding `/labs/lab-N` folder with scaffolding, so you're not starting from a blank file.
- **Python 3.10+ or Node.js 18+** — Labs are written with both stacks in mind; pick whichever you're more comfortable with. Code samples default to Python with Node equivalents noted.

- **A virtual environment** — `python -m venv .venv && source .venv/bin/activate` (or `npm init` for Node) so dependencies stay isolated.
- **Claude Code CLI installed** — Lab 2 assumes Claude Code is installed and authenticated. If not, set that up first — instructions are in Lab 2.

With that in place, let's build.

SECTION 02 · LAB 1

L1

Building Your First Multi-Agent System

★ **Lab Goal:** Build a working orchestrator + worker multi-agent system where a "manager" agent decomposes a task and delegates to two specialized "worker" agents, then synthesizes their outputs.

Step 1 — Define the Agent Roles

In a multi-agent system, the most common failure mode is giving every agent the same generic instructions. Instead, each agent needs a **narrow, well-defined role**, a **limited toolset**, and a **clear handoff contract** describing what it receives and must return.

For this lab, we'll build three agents:

- **Manager Agent** — receives the user's task, breaks it into sub-tasks, and delegates.
- **Research Agent** — has web-search-style tool access, gathers information.
- **Writer Agent** — takes research output and produces a final polished answer.

Step 2 — Project Structure

```
lab1-multi-agent/
├── agents/
│   ├── manager.py
│   ├── researcher.py
│   └── writer.py
├── orchestrator.py
├── tools.py
└── requirements.txt
```

Step 3 — Define Each Agent

AGENTS/MANAGER.PY

```
MANAGER_SYSTEM_PROMPT = """
You are the Manager Agent in a multi-agent system.
Your job is ONLY to decompose the user's request into:
1. A research sub-task (facts, data, sources needed)
2. A writing sub-task (structure, tone, target audience)

Return your output as strict JSON with keys: "research_task" and "writing_task".
Do not attempt to answer the user's question yourself.
"""

def build_manager_request(client, user_task: str):
    return client.messages.create(
```

```
        model="claude-sonnet-4-6",
        max_tokens=500,
        system=MANAGER_SYSTEM_PROMPT,
        messages=[{"role": "user", "content": user_task}],
    )
```

AGENTS/RESEARCHER.PY

```
RESEARCHER_SYSTEM_PROMPT = """
You are the Research Agent. You receive a narrow research task.
Use the provided search_tool to gather 3-5 relevant facts.
Return findings as a bulleted list with source attribution.
Do not draft prose — that is the Writer Agent's job.
"""

def build_researcher_request(client, research_task: str, tools: list):
    return client.messages.create(
        model="claude-sonnet-4-6",
        max_tokens=800,
        system=RESEARCHER_SYSTEM_PROMPT,
        tools=tools,
        messages=[{"role": "user", "content": research_task}],
    )
```

AGENTS/WRITER.PY

```
WRITER_SYSTEM_PROMPT = """
You are the Writer Agent. You receive research findings and a writing brief.
Produce a clear, well-structured final answer for the end user.
Do not invent facts not present in the research findings.
"""

def build_writer_request(client, writing_task: str, research_findings: str):
    prompt = f"Writing brief: {writing_task}\n\nResearch findings:\n{research_findings}"
    return client.messages.create(
        model="claude-sonnet-4-6",
        max_tokens=1000,
        system=WRITER_SYSTEM_PROMPT,
        messages=[{"role": "user", "content": prompt}],
    )
```

Step 4 — The Orchestrator

The orchestrator is the glue: it calls the Manager, parses its JSON output, routes sub-tasks to the Researcher and Writer, and returns the final synthesized answer.

ORCHESTRATOR.PY

```
import json
import anthropic
from agents.manager import build_manager_request
from agents.researcher import build_researcher_request
from agents.writer import build_writer_request
```

```
from tools import SEARCH_TOOL_SCHEMA

client = anthropic.Anthropic()

def run_pipeline(user_task: str) -> str:
    # Step 1: Manager decomposes the task
    manager_response = build_manager_request(client, user_task)
    plan = json.loads(manager_response.content[0].text)

    # Step 2: Researcher gathers facts
    research_response = build_researcher_request(
        client, plan["research_task"], tools=[SEARCH_TOOL_SCHEMA]
    )
    findings = research_response.content[0].text

    # Step 3: Writer synthesizes final output
    writer_response = build_writer_request(
        client, plan["writing_task"], findings
    )
    return writer_response.content[0].text

if __name__ == "__main__":
    result = run_pipeline("Explain the pros and cons of serverless architecture for a
beginner audience")
    print(result)
```

Step 5 — Test & Validate

Run the orchestrator and check three things:

1. Does the Manager output valid JSON every time? (Add a retry-with-repair loop if not.)
2. Does the Researcher's output stay factual and cite sources?
3. Does the Writer avoid introducing facts the Researcher didn't provide?

> ⚠ **Common Pitfall:** Skipping a strict output schema for the Manager Agent. Freeform text between agents breaks orchestration. Always enforce JSON (or a structured tool call) at agent-to-agent handoffs.

★ Lab 1 Complete! You've built a functioning orchestrator pattern — the foundation of every multi-agent architecture on the exam.

SECTION 03 · LAB 2

L2

Mastering Claude Code Workflows

★ **Lab Goal:** Learn to drive Claude Code through a real refactor-and-test workflow, using custom slash commands and a project-level CLAUDE.md for persistent context.

Step 1 — Install & Authenticate

BASH

```
npm install -g @anthropic-ai/claude-code
claude auth login
```

Verify the install:

BASH

```
claude --version
```

Step 2 — Create a Project Memory File

Claude Code reads a `CLAUDE.md` file at the root of your repo to persist project conventions across sessions. This is one of the most under-used features candidates get tested on.

CLAUDE.MD

```
# CLAUDE.md

## Project Conventions
- Use TypeScript strict mode everywhere.
- All new functions require unit tests in the same PR.
- Prefer functional components with hooks over class components.
- Run `npm run lint` before considering any task complete.

## Architecture Notes
- `/src/agents` contains all agent orchestration logic.
- `/src/tools` contains MCP tool definitions — do not put business logic here.
```

Step 3 — Build a Custom Slash Command

Custom commands let you package a repeatable workflow (e.g., "run the full test-and-refactor loop") into a single invocation.

.CLAUDE/COMMANDS/REFACTOR-AND-TEST.MD

```
Refactor the file at $ARGUMENTS to reduce cyclomatic complexity below 10.
After refactoring:
1. Run the existing test suite.
2. If any test fails, fix the refactor — do not modify the test.
3. Report a before/after complexity score.
```

Invoke it from the Claude Code terminal:

```bash
/refactor-and-test src/services/paymentProcessor.ts
```

Step 4 — The Plan → Execute → Verify Loop

The exam places heavy emphasis on **agentic coding discipline** — not just "ask Claude to write code," but a structured loop:

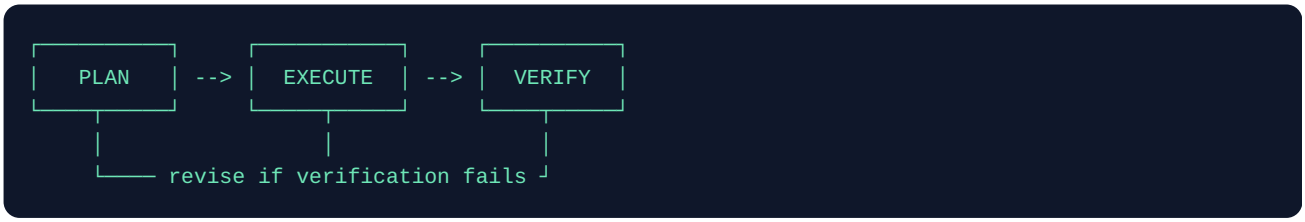

Diagram description

A three-stage loop. Plan stage: Claude proposes a step-by-step approach before touching code. Execute stage: Claude makes the edits or runs commands. Verify stage: tests, linters, or type-checkers confirm correctness. If verification fails, control returns to Plan with the failure context appended, rather than blindly retrying Execute.

In practice, ask Claude Code to explicitly separate these phases:

```bash
claude "Plan a fix for the failing test in auth.test.ts, but do not write any code yet — show me the plan first."
```

Then, once you approve:

```bash
claude "Proceed with the plan and run the test suite to verify."
```

Step 5 — Multi-File Refactor with Checkpoints

For larger refactors, use checkpointed commits so you can roll back cleanly if Claude's changes drift:

```bash
```

```
git checkout -b refactor/agent-orchestrator
claude "Refactor src/agents/orchestrator.py to use async/await instead of blocking calls.
Commit after each file is updated and passing tests."
```

⚠ **Common Pitfall:** Letting Claude Code run open-ended for large, multi-file changes without checkpoints. Always request incremental commits so you have rollback points.

★ Lab 2 Complete! You now know how to structure Claude Code sessions the way the exam expects: memory files, custom commands, and a disciplined plan-execute-verify loop.

SECTION 04 · LAB 3

L3

MCP Integration Claude Architect

★ **Lab Goal:** Build a minimal MCP server exposing a custom tool, then connect it to a Claude-powered client.

Step 1 — What MCP Actually Solves

Before MCP, every integration between an LLM and an external system (a database, a ticketing tool, a file system) required custom, one-off glue code. **MCP (Model Context Protocol)** standardizes this: any MCP-compliant client can talk to any MCP-compliant server, the same way any web browser can talk to any web server via HTTP.

An MCP server exposes three kinds of primitives:

- ● **Tools** — functions the model can call (e.g., `create_ticket`, `query_database`)
- ● **Resources** — data the model can read (e.g., a file, a document)
- ● **Prompts** — reusable prompt templates the server provides

Step 2 — Scaffold a Minimal MCP Server

MCP_SERVER.PY

```
from mcp.server import Server
from mcp.types import Tool, TextContent
import asyncio

app = Server("inventory-server")

@app.list_tools()
async def list_tools():
    return [
        Tool(
            name="check_stock",
            description="Check current stock level for a given SKU",
            inputSchema={
                "type": "object",
                "properties": {
                    "sku": {"type": "string", "description": "Product SKU"}
                },
                "required": ["sku"],
            },
        )
    ]

@app.call_tool()
```

```
async def call_tool(name: str, arguments: dict):
    if name == "check_stock":
        sku = arguments["sku"]
        # In production, replace with a real DB lookup
        stock_levels = {"SKU-001": 42, "SKU-002": 0}
        level = stock_levels.get(sku, "unknown")
        return [TextContent(type="text", text=f"Stock for {sku}: {level} units")]
    raise ValueError(f"Unknown tool: {name}")

if __name__ == "__main__":
    from mcp.server.stdio import stdio_server
    async def main():
        async with stdio_server() as (read, write):
            await app.run(read, write, app.create_initialization_options())
    asyncio.run(main())
```

Step 3 — Register the Server with a Claude Client

CLAUDE_DESKTOP_CONFIG.JSON

```
{
  "mcpServers": {
    "inventory-server": {
      "command": "python",
      "args": ["mcp_server.py"]
    }
  }
}
```

Step 4 — Call the Tool from a Claude Conversation

Once registered, you can simply ask:

Sample prompt

"Check the stock level for SKU-001 and let me know if I need to reorder."

The client-side model will invoke `check_stock` automatically, receive the `TextContent` result, and reason over it in its final answer — no manual function-calling glue required on your end.

Step 5 — Scope Permissions Tightly

MCP tools run with whatever permissions the server process has. The exam tests your ability to **scope tools narrowly**:

PYTHON — BAD VS. GOOD

```
# BAD: overly broad tool
Tool(name="run_sql", description="Run any SQL query against the database")
```

```
# GOOD: narrowly scoped tool
Tool(name="check_stock", description="Read-only stock lookup by SKU")
```

⚠ **Common Pitfall:** Exposing a generic "run_sql" or "execute_shell" tool for convenience. This is a top exam anti-pattern — always expose the narrowest tool that satisfies the use case.

★ Lab 3 Complete! You've built and registered a real MCP server — the exact skill tested in the "Tool Integration" domain of the exam.

SECTION 05 · LAB 4

L4

Build Production AI Agents with Claude

★ **Lab Goal:** Harden the multi-agent system from Lab 1 into something production-ready: retries, observability, guardrails, and cost controls.

Step 1 — Add Structured Logging

OBSERVABILITY.PY

```
import logging
import time
import json

logger = logging.getLogger("agent_pipeline")
logging.basicConfig(level=logging.INFO)

def log_agent_call(agent_name: str, input_tokens: int, output_tokens: int, latency_ms:
float):
    logger.info(json.dumps({
        "agent": agent_name,
        "input_tokens": input_tokens,
        "output_tokens": output_tokens,
        "latency_ms": latency_ms,
        "timestamp": time.time(),
    }))
```

Step 2 — Wrap Every Agent Call with Retries

RESILIENCE.PY

```
import time
from anthropic import APIError, APIStatusError

def call_with_retry(fn, max_retries=3, backoff_base=2):
    for attempt in range(max_retries):
        try:
            return fn()
        except (APIError, APIStatusError) as e:
            if attempt == max_retries - 1:
                raise
            wait = backoff_base ** attempt
            time.sleep(wait)
```

Step 3 — Add Guardrails at Handoff Boundaries

Every agent-to-agent handoff is a potential failure point. Validate structured output before passing it downstream:

GUARDRAILS.PY

```
from pydantic import BaseModel, ValidationError

class ManagerPlan(BaseModel):
    research_task: str
    writing_task: str

def validate_plan(raw_json: str) -> ManagerPlan:
    try:
        return ManagerPlan.model_validate_json(raw_json)
    except ValidationError as e:
        raise ValueError(f"Manager output failed validation: {e}")
```

Step 4 — Cost & Token Budgeting

Production agent systems need a hard ceiling on spend per task, or a runaway loop can burn through budget fast.

BUDGET.PY

```
class BudgetExceededError(Exception):
    pass

class TokenBudget:
    def __init__(self, max_tokens: int):
        self.max_tokens = max_tokens
        self.used = 0

    def consume(self, tokens: int):
        self.used += tokens
        if self.used > self.max_tokens:
            raise BudgetExceededError(f"Budget exceeded: {self.used}/{self.max_tokens}")
```

Step 5 — Human-in-the-Loop Escape Hatch

For any action with real-world side effects (sending an email, charging a card, deleting data), require explicit human confirmation:

CONFIRMATION.PY

```
SENSITIVE_ACTIONS = {"send_email", "charge_card", "delete_record"}

def requires_confirmation(tool_name: str) -> bool:
    return tool_name in SENSITIVE_ACTIONS

def gate_tool_call(tool_name: str, confirm_callback):
```

```
    if requires_confirmation(tool_name):
        approved = confirm_callback(tool_name)
        if not approved:
            raise PermissionError(f"User did not approve: {tool_name}")
    return True
```

Step 6 — Deploy Checklist

	Item
☐	Structured logging on every agent call
☐	Retry logic with exponential backoff
☐	Pydantic (or equivalent) validation at every handoff
☐	Hard token/cost budget per task
☐	Human confirmation gate on side-effecting tools
☐	Timeout on every agent call (no infinite loops)
☐	Staging environment tested before production rollout

⚠ **Common Pitfall:** Shipping a multi-agent system with no budget ceiling. A single mis-behaving loop between two agents can generate thousands of unnecessary API calls in minutes.

★ Lab 4 Complete! You now have a hardened, production-grade version of your multi-agent pipeline — exactly what the exam's "Production Readiness" domain evaluates.

SECTION 06

Must-Fork GitHub Repos

Building from scratch is great for learning, but forking well-structured starter projects will save you hours and show you idiomatic patterns. Below is a curated list of repo **categories** worth searching for and forking on GitHub — use these as search terms since specific repos and their popularity shift constantly:

Category	Search Terms	Why Fork It
Multi-agent starter kits	`claude multi-agent orchestrator template`	Pre-built manager/worker scaffolding like Lab 1
MCP server examples	`model context protocol server example`	Reference implementations of tools/ resources/prompts
Claude Code workflow configs	`claude code CLAUDE.md examples`	Real-world project memory files and slash commands
Agent evaluation harnesses	`LLM agent eval harness`	Test suites for measuring agent reliability
Production agent templates	`production ready LLM agent boilerplate`	Logging, retries, guardrails already wired in
Tool-use cookbook repos	`anthropic tool use cookbook`	Copy-paste tool schemas for common integrations

Contribution Guide (For When You Fork One)

1. **Read the CONTRIBUTING.md first.** Most well-maintained repos reject PRs that don't follow their commit message and branch-naming conventions.
2. **Open an issue before a large PR.** Confirm the maintainer wants the feature before you build it — this avoids wasted work.
3. **Add tests with every change.** Especially for agent logic — a passing test is your proof the orchestration still behaves correctly.
4. **Keep PRs small and scoped.** One logical change per PR rather than bundling five unrelated changes.
5. **Document new tools/agents in the README.** Update the architecture diagram/description in the docs.
6. **Use Conventional Commits.** e.g. `feat(agents): add validation guardrail to manager handoff` — many repos enforce this in CI.

★ **Tip:** Star and fork 3–5 repos in each category above before your exam. Reading real, messy, production code is one of the fastest ways to internalize the architecture patterns the exam tests.

SECTION 07

Claude Architect Architecture Patterns

Pattern Reference Table

Pattern	When to Use	Key Risk	Mitigation
Orchestrator–Worker	Task naturally decomposes into independent sub-tasks	Manager becomes a bottleneck / single point of failure	Add manager output validation + fallback routing
Reflection Loop	Output quality matters more than latency	Infinite self-critique loops	Cap reflection iterations (e.g., max 3 passes)
Tool-Use Router	Many possible tools, only 1–2 needed per request	Wrong tool selected, or overly broad tool exposed	Narrow tool scopes + descriptive docstrings
Planner–Executor	Multi-step tasks needing explicit sequencing	Plan drifts from execution reality	Re-plan on verification failure
Parallel Fan-Out/Fan-In	Independent sub-tasks can run concurrently	Race conditions merging results	Deterministic merge function + timeout per branch
Human-in-the-Loop Gate	Side-effecting or high-stakes actions	Agent takes irreversible action autonomously	Explicit confirmation gate
Memory-Augmented Agent	Long-running or multi-session tasks	Context window overflow, stale memory	Summarize + prune memory on a schedule

Reflection-Loop Code Example

The **Reflection Loop** pattern has an agent critique and revise its own output before returning it — commonly tested in the exam's "Quality & Reliability" domain.

REFLECTION_LOOP.PY

```
import anthropic

client = anthropic.Anthropic()

DRAFT_PROMPT = "Write a concise technical explanation of: {topic}"
CRITIQUE_PROMPT = """
Critique the following draft for accuracy, clarity, and completeness.
If it is already excellent, respond with exactly: "APPROVED"
Otherwise, list specific, actionable revisions.

Draft:
{draft}
"""
```

```
REVISE_PROMPT = """
Revise the draft based on this critique. Return only the revised text.

Original draft:
{draft}

Critique:
{critique}
"""

def reflection_loop(topic: str, max_iterations: int = 3) -> str:
    draft = client.messages.create(
        model="claude-sonnet-4-6",
        max_tokens=500,
        messages=[{"role": "user", "content": DRAFT_PROMPT.format(topic=topic)}],
    ).content[0].text

    for i in range(max_iterations):
        critique = client.messages.create(
            model="claude-sonnet-4-6",
            max_tokens=400,
            messages=[{"role": "user", "content": CRITIQUE_PROMPT.format(draft=draft)}],
        ).content[0].text

        if critique.strip() == "APPROVED":
            break

        draft = client.messages.create(
            model="claude-sonnet-4-6",
            max_tokens=500,
            messages=[{"role": "user", "content": REVISE_PROMPT.format(draft=draft,
critique=critique)}],
        ).content[0].text

    return draft

if __name__ == "__main__":
    final = reflection_loop("How MCP servers expose tools to LLM clients")
    print(final)
```

Diagram description

A cyclical flow — Draft → Critique → Decision diamond ("Approved?"). If yes, exit with final output. If no, route to Revise, then back into Critique, capped at `max_iterations` to prevent infinite loops.

⚠ **Common Pitfall:** Omitting the iteration cap. Without `max_iterations`, a reflection loop can cycle indefinitely if the critique agent never returns "APPROVED."

★ Ready to Get Certified?

You've built multi-agent systems, MCP servers, Claude Code workflows, and production-grade agents. The only thing left is the exam.

Visit PassITExams.com

and start your exam prep today.