

baswe.Ai Engineer Accelerator™

Integration, Agents, & Orchestration Checkpoint

Format:

Build + Technical Exam + Loom Walkthrough

Time investment:

Weekend sprint (10–14 hours)

Stack (locked for build):

Python · LangGraph · LangChain · FastAPI · MCP · Claude Code · LangSmith



baswe's

AiEngineerAccelerator

Pre-checkpoint checklist

Before you start, confirm all of the following.

Accounts & API access

- Anthropic API key
- LangSmith account, free tier is fine (you will trace at least one full agent run).
- GitHub account
- Loom account (free tier is fine for a single recording).
- Claude Code installed and configured locally.

Local environment

- Python 3.11 or higher.
- uv or poetry for dependency management (whichever you prefer; document which in your README).
- Docker installed (you will containerize the FastAPI service).
- LangGraph, LangChain, FastAPI, langsmith, and an MCP client library (e.g., mcp or langchain-mcp-adapters) installed in a fresh project venv.

MCP server identified

You must integrate at least one MCP server. You may choose any of:

- An off-the-shelf MCP server (filesystem, GitHub, Slack, Postgres, web fetch, etc.).
- A custom MCP server you author yourself, exposing 1–3 tools tied to your scenario.

At least one tool used by your agents must be reached through the MCP protocol, not as a Python function bound directly to LangGraph.

Part 1: The Build

Scenario

You have been hired as the first AI engineer at a 500-person professional services firm. The firm's analyst team spends roughly 4 hours a day on early-stage research — pulling background on companies, regulatory filings, and market context — before any deliverable work begins. Senior partners are tired of paying analyst rates for what amounts to internet research that an agent should be doing in minutes.

Your task: ship an internal multi-agent research workflow that takes a research question, you may pick the questions domain (technical research, market research, competitive analysis, codebase analysis, regulatory research), decomposes it into sub-tasks, executes those sub-tasks using tools (including at least one tool reached through MCP), and returns a grounded, citation-backed report. The system runs behind a FastAPI service, is traceable in LangSmith, and is evaluated against a golden task set by the author. This is a v1 and does **not** need to be deployed to production, but must be evaluated like it will.

Required architecture

Your system must include all of the following. The shape is non-negotiable; the implementation choices inside each piece are yours.

1. A LangGraph state graph with at least three agents

- A Planner agent that decomposes the user's research question into structured sub-tasks.
- At least one Researcher agent that executes sub-tasks using tools.
- A Synthesizer agent that produces the final structured report with citations.
- An explicit StateGraph (or equivalent). State must be inspectable, edges must be defined, and at least one conditional edge must drive routing.

2. At least one MCP-backed tool

- One or more agents must call a tool that is served by an MCP server, not bound directly as a Python function.
- If you author a custom MCP server, document its tool schema in your README.
- If you use an off-the-shelf MCP server, document which one and why you chose it.

3. A FastAPI service exposing the system

- POST /research: accepts a research question payload (validated with Pydantic) and returns either a job_id or the final report.
- GET /jobs/{job_id}: returns the status of an in-flight research job (queued / running / done / failed) and the final result if complete.
- GET /healthz: basic liveness probe.
- Pydantic schemas for every request and response, with appropriate HTTP status codes (200, 202, 400, 404, 500).
- At least one endpoint must be implemented async, not sync, and the choice must be defensible.

4. LangSmith tracing

- Every agent run must be traceable in LangSmith.

5. A small evaluation harness

- A golden task set of at least 5 research questions you wrote yourself, with expected output structure (example structure: "answer must include citations to ≥ 3 sources, must cover X, Y, Z subtopics").
- A simple Python script that runs the system over the golden set and reports a success rate.
- This is a lightweight harness, not a full eval framework. Full eval CI is the next module's checkpoint. Here you're proving you can think in evals at all.

6. Containerization

- A working Dockerfile that builds and runs the FastAPI service locally with one command.
- A docker-compose.yml is acceptable but not required.

7. Built with Claude Code

- Claude Code must be a primary tool in your development workflow. Not the only tool — but a primary one.
- You must include a CLAUDE_CODE_NOTES.md file in your repo documenting: which parts you built with Claude Code, what your prompting workflow was, what it got wrong that you had to correct, and one concrete example of a prompt that produced a meaningful piece of code.

Suggested phasing

This is a suggested 7–10 hour sequence.

Phase 1: Scope and scaffold (1–1.5 hrs)

1. **Choose your research domain.** Write a one-paragraph problem statement at the top of your README. State who would use this system and what question shape it answers.
2. **Sketch the state graph.** Before writing code, draw the graph: nodes, edges, conditional edges, and the State TypedDict. Include this diagram in your README (a simple Mermaid diagram is fine).
3. **Initialize the project.** Set up the repo, dependencies, .env.example, and a working FastAPI hello world. Verify Docker builds.
4. **Implement the State.** Define your TypedDict or Pydantic state explicitly.
5. **Implement the Planner.** An agent that takes the user's question and produces a structured list of sub-tasks. Constrain its output with a Pydantic schema.
6. **Implement the Researcher(s).** One or more agents that execute sub-tasks using tools. At least one tool must be MCP-backed (continue in Phase 3 if needed). A tool error must not kill the whole graph, researchers must be able to fail gracefully.
7. **Implement the Synthesizer.** An agent that combines researcher outputs into a final structured report with citations. Citations must point to specific source identifiers, not free text.
8. **Wire the conditional edge.** Add at least one routing decision — e.g., the Planner returns 0 sub-tasks → skip Researchers and respond with "insufficient input"; or Synthesizer detects missing information → loop back to Planner once. Document the routing rule in code comments.

Phase 3: MCP integration (1–1.5 hrs)

9. **Stand up your MCP server.** Either run an existing one (filesystem, GitHub, web fetch, etc.) or author a small one with 1–3 tools. Verify the server responds to a tools/list call before wiring it into the agent.
10. **Wire MCP into the Researcher.** Use a LangChain or LangGraph MCP adapter (or an MCP client directly) so at least one tool call goes over the MCP protocol.
11. **Add minimal MCP error handling.** If the MCP server is down or returns malformed output, the Researcher must catch it, log it, and continue.

Phase 4: FastAPI service layer (1.5–2 hrs)

12. **Build the routes.** POST /research, GET /jobs/{job_id}, GET /healthz. Pydantic models for request and response. Proper status codes.
13. **Pick a job model and defend it.** You can run research synchronously (simpler, but blocks for the duration of the research call) or asynchronously with an in-memory job store (closer to production, but more code). Either is acceptable. Document the choice in your README and be ready to defend it on the Loom.
14. **Add request validation and error handling.** Reject empty questions with 400. Return 404 for unknown job_ids. Return structured error responses, not raw stack traces.
15. **Add basic logging.** Every request should log: route, status, duration, job_id, and any tool error encountered. This is the minimum observability bar.

Phase 5: Tracing and evaluation (1–1.5 hrs)

16. **Wire LangSmith.** Set the env vars, give your project a clear name, and run one full research request end to end. Verify the trace shows planner → researcher(s) → synthesizer with all tool calls.
17. **Build the golden task set.** Write at least 5 research questions and define what "good" looks like for each. This is a JSON or YAML file in your repo.
18. **Build the eval runner.** A Python script that loads the golden set, calls your /research endpoint for each question, scores the output against the expected structure, and prints a pass/fail rate. LLM-as-judge is acceptable for the structural check.

Phase 6: Container and ship (0.5–1 hr)

19. **Containerize.** Write the Dockerfile. The container should start the FastAPI app and read API keys from environment variables, never from a baked-in file.
20. **Verify cold-start.** Build the image clean from a fresh clone of your repo. Run it. Hit /research with one of your golden questions. Confirm it returns a real answer end to end.
21. **Write the README.** Explain what the system does, how to run it, the architecture (with the Mermaid diagram from Phase 1), the MCP integration, the eval results, and any tradeoffs you made. Treat this README as onboarding documentation for a teammate joining your team.

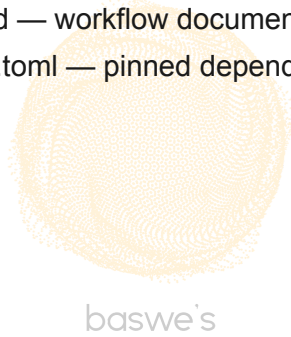
Required technical checks

#	Required check	How it's verified
1	Repo is cloneable and contains a working README	clone from URL
2	Project runs locally via documented setup steps	follow README
3	Docker image builds and runs	`docker build` + `docker run` succeed
4	POST /research returns a valid response on a real question	Curl or HTTP client test
5	GET /healthz returns 200	Curl test
6	LangGraph state graph is implemented as a real graph (not a chain)	Code review of graph definition
7	At least one tool call goes through MCP	Code review + trace inspection
8	LangSmith trace screenshot is included for one full run	Visual inspection of traces/ folder
9	Golden task set exists with ≥ 5 questions	File present, contents non-trivial
10	Eval runner script exists and runs	execute script
11	CLAUDE_CODE_NOTES.md exists with required sections	Visual inspection

Build contents

The repo must contain:

- README.md — problem statement, architecture (with Mermaid diagram), setup, usage, tradeoffs, eval results.
- /app or /src — your FastAPI app and LangGraph code, organized cleanly.
- /agents — agent definitions (planner, researcher(s), synthesizer).
- /tools — tool definitions, including the MCP wiring.
- /eval — golden task set + eval runner script.
- /traces — at least one screenshot of a LangSmith trace for a full run.
- Dockerfile — working, builds clean from a fresh clone.
- .env.example — every required env var documented.
- CLAUDE_CODE_NOTES.md — workflow documentation as specified.
- requirements.txt or pyproject.toml — pinned dependencies.



AiEngineerAccelerator

Part 2: Loom Walkthrough

Record one Loom (or equivalent screen recording) walking a reviewer through the system. This is the part that proves you actually built it and that you understand what you built. Do not script-read. Talk like an engineer defending an architecture review.

Format

- Single Loom recording, 10–30 minutes, screen share with face cam on.
- Public or unlisted Loom link, included in your README.

Required sections

1. Problem framing (2–3 minutes)

- State the problem in one sentence.
- State who the user is and what they get out of using this system.
- Name one decision you made about scope and what you cut to stay in the weekend window.

2. Architecture defense (8–12 minutes)

- Walk the state graph on the Mermaid diagram. Explain every node and every edge.
- Defend the orchestration pattern you chose (supervisor, sequential, hierarchical, swarm). Name the alternatives and why you didn't pick them.
- Defend the sync vs. async job model in your FastAPI layer. Name the tradeoff.
- Defend the MCP integration choice. What you used, why MCP and not a direct tool binding.

3. Live demo (8–12 minutes)

- Run the system end to end on a fresh question that is not in your golden set.
- Open the LangSmith trace and walk through one complete run. Point out any latency hotspot or any tool call that could fail.
- Show one of your golden task evals running and explain what it's actually measuring.

4. Tradeoffs and what you'd ship next (3–5 minutes)

- Name one thing in this system that is not production-ready and what it would take to harden it.
- Name one place where a different framework (CrewAI, LlamaIndex, smolagents, Haystack) would have been a better fit, and explain why you still picked LangGraph here.

- Name one thing Claude Code got wrong during the build that you had to correct manually.

5. Failure mode walk-through (2–3 minutes)

- Deliberately break one thing, kill the MCP server, send a malformed request, or send a question your Planner can't decompose. Show how the system fails. Show what the user sees. Explain what would log.

Part 3: Technical Exam

Format

- Open-book. Open-internet.
- Do **NOT** use an LLM to generate your answers. Do **NOT** Google the entire question. Cheating on these only cheats yourself. Use of LLMs to draft answers is grounds for **failing the entire checkpoint**. We are evaluating your judgment, not your prompting
- Time-boxed: 1-2 hours. We do not enforce the timer, but if you spend 6+ hours on this section, that is a signal you might not have understood the concepts deep enough.
- Written answers in a single Markdown file (exam-answers.md).

AiEngineerAccelerator

Section A: Agent foundations

Q1 (2 pts)

In one paragraph, define the difference between an LLM chain and an agent. Use the system you just built as an example. What in your system makes it an agent rather than a chain?

Q2 (3 pts)

Name three concrete failure modes that get worse as you scale an agentic system from a single agent to multiple agents. For each, name the mitigation pattern.

Section B: LangChain & LangGraph

Q3 (4 pts)

In LangGraph, what is the difference between a regular edge and a conditional edge, and what is the State object actually doing under the hood? Answer with reference to the StateGraph you wrote in your build.

Q4 (4 pts)

Compare the supervisor pattern vs. the hierarchical pattern vs. the swarm pattern for multi-agent orchestration in LangGraph. Give one scenario where each is the right answer.

Q5 (3 pts)

You have a text-to-SQL agent backed by LangGraph and MongoDB (as in Unit 2.2's text-to-query course). A user asks a question that requires data from two collections. Walk through the failure modes you'd expect, and how you would harden the system against each.

Section C: Multi-framework agent literacy

Q6 (5 pts)

You have four agentic projects on your plate: (1) a tightly orchestrated sales-research crew with clearly defined roles, (2) a heavy-document RAG-plus-agent system, (3) a Hugging Face open-source-only deployment for a privacy-sensitive client, (4) a healthcare research agent that must compose Haystack pipelines. For each, pick one of CrewAI, LlamaIndex, smolagents, or Haystack and defend the choice in 2–3 sentences. You may reuse a framework across projects only if you can defend why.

Q7 (4 pts)

Explain MCP in one paragraph as if you were explaining it to a senior backend engineer who has never heard of it. Then, in a second paragraph, explain why an AI engineer should care about MCP specifically, what does it solve that bespoke tool plumbing does not?

Section D: FastAPI for AI workloads

Q8 (3 pts)

Why is async FastAPI especially relevant for LLM-backed services compared to sync FastAPI? Give a concrete example from your own build where async handling matters or would matter at scale.

Q9 (3 pts)

Your /research endpoint hits an LLM provider that randomly returns 500s ~2% of the time. Sketch (in code or pseudocode) the retry, timeout, and circuit-breaker pattern you would apply, and explain what you'd log so a teammate could debug a regression next week.

Section E: AI-assisted development

Q10 (3 pts)

Compare Cursor, Claude Code, and GitHub Copilot in terms of where each is the right tool. Give one task per tool where it is clearly the strongest choice, and one task where it is clearly not.

Q11 (3 pts)

Open your CLAUDE_CODE_NOTES.md. Pick one prompt or workflow you used during this build that worked well, and one that failed. For each, explain in 3–4 sentences why it worked or failed, and what you'd change. Paste the prompts verbatim.

Section F: Synthesis

Q12 (8 pts)

Imagine you are 90 days into a new AI engineer role. The team owns a customer-support agent that is in production and unreliable: ~12% of conversations end in a state the team cannot explain, latency p95 is 18 seconds, and there is no eval harness. You have one quarter to fix it. Write a one-page plan that covers: (a) what you would instrument first and why, (b) the eval harness you would stand up and how you would seed it, (c) which architectural change you would prioritize first and what you would defer, and (d) one bet you would resist making no matter how loud the pressure to make it. baswe's

AiEngineerAccelerator