AGENTIC AI ROADMAP

# 120 concepts to become an agentic AI engineer

8 areas, 15 concepts each. Every concept explained in one line, the order to study each area, and two practice questions to test yourself.

How to use it: one area at a time, tick each concept only when you can explain it out loud in under a minute, then answer the two practice questions without notes.

@chalkesahil · chalkesahil.ai

AREA 01 / 08

# Python and APIs

**Study order:** Get comfortable calling APIs from Python before touching any agent framework.

| | | |
|---|---|---|
| 01 | **Functions and modules** | Small, testable pieces you can hand to an agent as tools. |
| 02 | **Type hints** | Agents and frameworks read your types to build tool schemas. |
| 03 | **Pydantic** | Validate and parse structured data, including model output. |
| 04 | **JSON** | The format models, tools and APIs all speak. |
| 05 | **HTTP requests** | requests or httpx to call any REST API. |
| 06 | **REST basics** | Methods, status codes, auth headers, pagination. |
| 07 | **async and await** | Run many model and tool calls at once. |
| 08 | **Environment variables** | Keep API keys out of code. |
| 09 | **Error handling** | Catch timeouts and bad responses without crashing the agent. |
| 10 | **Retries with backoff** | Rate limits are normal; retry politely. |
| 11 | **FastAPI** | Wrap your agent in an API others can call. |
| 12 | **Streaming responses** | Send tokens to the user as they arrive. |
| 13 | **Logging** | Record every step so you can see what the agent did. |
| 14 | **Virtual environments** | venv or uv so dependencies don't collide. |
| 15 | **Git** | Version your prompts and code together. |

**Practice**

1. Write a function that calls a weather API and returns clean JSON.
2. Make 10 API calls in parallel and handle the ones that fail.

AREA 02 / 08

# LLM basics

**Study order:** Learn how models behave before you trust one with actions.

| | | |
|---|---|---|
| 01 | **Tokens** | Models read and bill in tokens, not words. |
| 02 | **Context window** | Everything the model can see in one call. |
| 03 | **Temperature** | Higher means more varied answers; lower means more consistent. |
| 04 | **System prompt** | Standing instructions that shape every reply. |
| 05 | **Chat messages** | System, user and assistant turns make up the conversation. |
| 06 | **Next-token prediction** | The model predicts one token at a time. |
| 07 | **Hallucinations** | Confident wrong answers; plan for them. |
| 08 | **Embeddings** | Text turned into vectors so similar meaning sits close. |
| 09 | **Model choice** | Bigger is smarter and slower; smaller is cheaper and faster. |
| 10 | **Pricing** | Input and output tokens are priced differently. |
| 11 | **Latency** | Long prompts and long answers both add wait time. |
| 12 | **Rate limits** | Requests and tokens per minute per account. |
| 13 | **Structured output** | Ask for JSON that matches a schema. |
| 14 | **Multimodal input** | Many models read images and files too. |
| 15 | **Open vs closed models** | Run Llama or Qwen yourself, or call Claude or GPT through an API. |

**Practice**

1. Why does the same prompt give different answers on two runs?
2. Estimate the monthly cost of an assistant handling 10,000 chats a day.

AREA 03 / 08

# Prompt engineering

**Study order:** Write prompts like specs: role, task, context, format, examples.

| | | |
|---|---|---|
| 01 | **Clear instructions** | Say exactly what you want, including what to avoid. |
| 02 | **Role and context** | Tell the model who it is helping and why. |
| 03 | **Few-shot examples** | Show two or three good answers to copy. |
| 04 | **Output format** | Specify JSON keys, length and tone. |
| 05 | **Step-by-step reasoning** | Ask the model to think before answering on hard tasks. |
| 06 | **Delimiters** | Separate instructions from data with tags or fences. |
| 07 | **Prompt templates** | Reusable prompts with variables. |
| 08 | **Prompt chaining** | Split a big task into smaller prompts in sequence. |
| 09 | **Self-check** | Ask the model to verify its own answer. |
| 10 | **Negative examples** | Show what a bad answer looks like. |
| 11 | **Prompt injection** | Text in data that tries to override your instructions. |
| 12 | **Prompt versioning** | Track prompt changes like code. |
| 13 | **Long context tips** | Put the question after long documents. |
| 14 | **Refusals** | Know why a model declines and how to rephrase a fair request. |
| 15 | **Testing prompts** | Run a prompt on many inputs before calling it done. |

**Practice**

1. Write a prompt that extracts invoice fields as JSON every time.
2. How would you stop a support bot from following instructions hidden in a user's email?

AREA 04 / 08

# Tool calling

**Study order:** This is the core of agents: the model decides, your code acts.

| | | |
|---|---|---|
| 01 | **What a tool is** | A function the model can ask your code to run. |
| 02 | **Tool schemas** | Name, description and parameters as JSON Schema. |
| 03 | **Good tool descriptions** | The model picks tools by reading these. |
| 04 | **The tool loop** | Model asks, code runs the tool, result goes back, repeat. |
| 05 | **Parallel tool calls** | Several tools requested in one turn. |
| 06 | **Tool results** | Return short, clear results the model can use. |
| 07 | **Tool errors** | Return the error as text so the model can recover. |
| 08 | **Stopping conditions** | Max steps, budget, or a final-answer signal. |
| 09 | **Read vs write tools** | Reading is safe; writing needs more guardrails. |
| 10 | **Human approval** | Pause before risky actions like sending or paying. |
| 11 | **MCP** | Model Context Protocol: a standard way to plug tools into any agent. |
| 12 | **Function calling across providers** | Same idea, slightly different formats in each API. |
| 13 | **Too many tools** | More tools means more wrong picks; keep the set small. |
| 14 | **Idempotent tools** | Safe to run twice if the agent retries. |
| 15 | **Tool permissions** | Each tool gets only the access it needs. |

**Practice**

1. Build an agent with a calculator and a web search tool.
2. Your agent keeps calling the same tool in a loop. How do you stop it?

AREA 05 / 08

# RAG

**Study order:** Build a simple RAG over your own notes before learning advanced tricks.

| | | |
|---|---|---|
| 01 | **Why RAG** | Give the model facts it was never trained on. |
| 02 | **Chunking** | Split documents into pieces that keep meaning together. |
| 03 | **Embedding models** | Turn chunks and questions into vectors. |
| 04 | **Vector databases** | Store vectors and find the nearest ones fast. |
| 05 | **Similarity search** | Cosine similarity finds chunks close in meaning. |
| 06 | **Top-k retrieval** | How many chunks to pass to the model. |
| 07 | **Metadata filters** | Search only the right source, date or user. |
| 08 | **Hybrid search** | Combine keyword and vector search. |
| 09 | **Re-ranking** | A second model reorders results by relevance. |
| 10 | **Query rewriting** | Turn a vague question into a better search. |
| 11 | **Citations** | Make the model point to the chunk it used. |
| 12 | **Grounding** | Tell the model to answer only from retrieved text. |
| 13 | **Updating the index** | Re-embed when documents change. |
| 14 | **Agentic RAG** | The agent decides when and what to search. |
| 15 | **RAG evaluation** | Check retrieval quality and answer quality separately. |

**Practice**

1. Build a bot that answers questions from a 100-page PDF with citations.
2. Your RAG bot gives wrong answers. How do you tell if retrieval or generation is at fault?

AREA 06 / 08

# Agent frameworks

**Study order:** Build one agent without a framework first so you know what they do for you.

| | | |
|---|---|---|
| 01 | **When you need an agent** | If a fixed workflow works, skip the agent. |
| 02 | **Workflows vs agents** | Code decides the steps vs the model decides the steps. |
| 03 | **ReAct pattern** | Reason, act with a tool, observe, repeat. |
| 04 | **Planning** | Make a plan first, then execute step by step. |
| 05 | **Routing** | Send each request to the right prompt or sub-agent. |
| 06 | **Multi-agent systems** | Several specialised agents passing work around. |
| 07 | **Orchestrator and workers** | One agent splits the task, others do the parts. |
| 08 | **LangChain** | Popular building blocks for chains and tools. |
| 09 | **LangGraph** | Agents as graphs with state and loops. |
| 10 | **CrewAI** | Role-based multi-agent setups. |
| 11 | **Agent SDKs** | OpenAI Agents SDK and Claude Agent SDK give you the loop ready-made. |
| 12 | **n8n and no-code agents** | Visual workflows; common for business automation. |
| 13 | **State management** | Keep track of what the agent has done and learned. |
| 14 | **Handoffs** | Pass the conversation from one agent to another. |
| 15 | **Choosing a framework** | Pick for your use case, not for the hype. |

**Practice**

1. Build a research agent that searches, reads and writes a summary.
2. When would you use one agent with many tools instead of many agents?

AREA 07 / 08

# Memory and evals

**Study order:** Write evals early; they tell you if a change helped or hurt.

| | | |
|---|---|---|
| 01 | **Short-term memory** | The conversation so far, inside the context window. |
| 02 | **Long-term memory** | Facts saved outside the model and fetched later. |
| 03 | **Summarising history** | Compress old turns to fit the context window. |
| 04 | **Context engineering** | Choosing exactly what goes into each call. |
| 05 | **User profiles** | Remember preferences across sessions. |
| 06 | **Why evals** | Prompts break silently; tests catch it. |
| 07 | **Golden datasets** | Real inputs with known good answers. |
| 08 | **LLM as a judge** | A model grades outputs against a rubric. |
| 09 | **Task success rate** | Did the agent finish the job correctly? |
| 10 | **Trajectory evals** | Check the steps, not just the final answer. |
| 11 | **Regression tests** | Re-run evals on every prompt or model change. |
| 12 | **Tracing** | LangSmith, Langfuse or similar to see each step. |
| 13 | **Cost and latency tracking** | Evals should measure money and time too. |
| 14 | **Human review** | Spot-check real conversations every week. |
| 15 | **A/B testing prompts** | Compare two versions on live traffic. |

**Practice**

1. Design an eval set for a customer support agent.
2. How would you give an agent memory of a user's past orders?

AREA 08 / 08

# Deployment and safety

**Study order:** Ship a small agent to real users, then harden it.

| | | |
|---|---|---|
| 01 | **Serving an agent** | Behind an API with auth and rate limits. |
| 02 | **Background jobs** | Long tasks run in a queue, not in the request. |
| 03 | **Streaming to the UI** | Show progress so users don't stare at a spinner. |
| 04 | **Caching** | Reuse answers and prompt prefixes to save cost. |
| 05 | **Cost controls** | Budgets per user and per task. |
| 06 | **Timeouts and fallbacks** | Switch models or degrade gracefully when one fails. |
| 07 | **Guardrails** | Check inputs and outputs for unsafe content. |
| 08 | **Prompt injection defence** | Treat fetched content as data, never as instructions. |
| 09 | **Least privilege** | Agents get only the permissions their tools need. |
| 10 | **Sandboxing** | Run agent-written code in isolation. |
| 11 | **PII handling** | Mask personal data before it reaches logs or models. |
| 12 | **Audit logs** | Record who asked, what the agent did, and why. |
| 13 | **Monitoring** | Alert on error rates, cost spikes and stuck runs. |
| 14 | **Human in the loop** | Escalate to a person when confidence is low. |
| 15 | **Portfolio project** | One deployed agent you can demo beats ten notebooks. |

**Practice**

1. Your agent's monthly bill tripled overnight. Where do you look?
2. How do you stop an email agent from forwarding private data when a message tells it to?