datacamp

AI Agents Cheat Sheet

Learn about AI agents online at www.DataCamp.com

0 How to Use This Cheat Sheet

This cheat sheet is a companion to our ***Introduction to AI Agents*** course, but it also works on its own. It's organized into numbered sections, so you can go through them in order or jump straight to the part you're most interested in. While the concept of AI agents is broad, this cheat sheet focuses on systems built around language models.

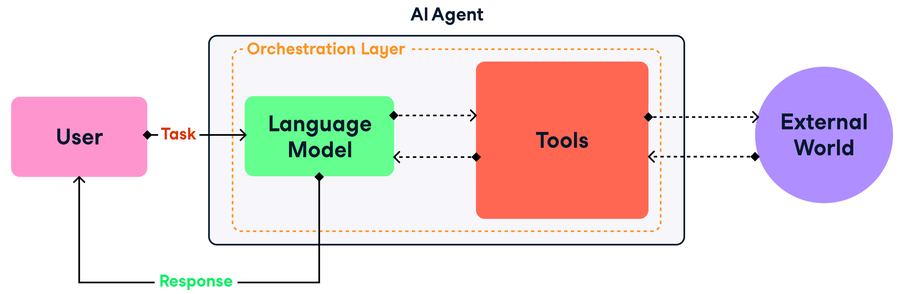
1 What Is an AI Agent?

An **AI Agent** is a system that uses a language model to achieve a user-defined goal. It interacts with its environment by reasoning, planning, and executing actions, often with the help of external tools.

For example, an AI agent could help you book a flight by searching for options and completing the reservation for you.

At its core, an AI agent consists of **three main components**:

- The **language model** powering its reasoning and decision-making;
- The **tools** it uses to interact with the external world and gather information;
- The **orchestration layer** that governs how the agent processes information, plans, and executes actions to achieve its goals.



2 Language Model

A **language model (LM)** is a type of artificial intelligence program designed to understand, generate, and process human language. The LM is the central decision-maker and reasoning engine in an AI agent.

A common misconception is that the underlying LM is the agent. LMs, such as GPT-4o, lack real-world access and can't "think" beyond their training data. They need to be connected to tools (e.g., a weather API) to act as AI agents.

Different types of language models can be used in AI agents—see the table on the right.

Type	Description	Examples	Suitable for
Large Language Models (LLMs)	General-purpose models	GPT-4o, Gemini 2.5 Flash, DeepSeek-V3	Tasks of medium complexity
Small Language Models (SLMs)	Efficient and cost-effective models	Gemma 3n, DeepSeek-R1-Distill-Qwen-1.5B	Simpler tasks
Reasoning Models	Powerful models that generate long chains of thought before generating an answer	OpenAI O3, DeepSeek-R1, Claude Opus 4	Complex problems in coding, math and science

3 Tools

Tools extend an AI agent's capabilities by allowing it to interact with external systems and data. They bridge the gap between a language model's internal capabilities and the outside world.

Tools can take various forms, often aligning with common web API methods like GET, POST, PATCH, and DELETE. Broadly, agents need three types of tools:

Extensions

Extensions bridge an agent and an external API (weather API, flights API, maps API, etc.). Each extension comes with example calls that teach the model which endpoint to invoke and which parameters are required, letting the agent decide at run time when (and how) to execute the API call.

Functions

Functions are custom chunks of code that live outside the model but can be called by it. The model decides which function to invoke and fills in the arguments, yet the code actually runs in your own app (client-side), not inside the agent. This lets developers keep tight control over execution, security, and post-processing.

Data stores

Data stores function as a "knowledge vault" from which agents can retrieve information embedded in existing files and databases. At runtime, the agent taps into these sources (spreadsheets, PDFs, databases, websites, etc.) pulling back just the passages it needs to stay accurate and current.

4 Orchestration Layer

The **orchestration layer** is the cyclical process governing how an AI agent processes information, reasons, plans, and executes actions to achieve its goals. It maintains the agent's memory, state, reasoning, and planning. This layer uses various cognitive architectures and prompt engineering frameworks to guide reasoning and planning:

- **Chain-of-Thought (CoT):** Enables reasoning through intermediate steps, breaking down complex problems into a series of simpler, sequential thoughts.
- **Tree-of-Thoughts (ToT):** Generalizes over CoT prompting by allowing the model to explore various "thought chains."
- **ReAct:** A framework that allows language models to Reason and take Action.

Orchestration patterns generally fall into two categories:

- **Single-agent systems:** A single LM, equipped with tools and instructions, executes workflows in a loop.
- **Multi-agent systems:** Workflow execution is distributed across multiple coordinated agents, often used when complex instructions or tool overload become an issue for a single agent. These can follow:
  - A **Manager** pattern (a central agent delegates tasks via tool calls); or a
  - A **Decentralized** pattern (agents hand off tasks to one another as peers).

5 Agentic Protocols

Agentic protocols are standardized frameworks that facilitate communication and interaction within and between AI agents.

Model Context Protocol (MCP)

Anthropic's model context protocol (MCP) is an open standard designed to standardize how applications provide context to LLMs and connect them to external tools and services.

Example: An AI assistant in Slack could use MCP to pull the latest project updates directly from a project management tool like Asana and display them in your channel.

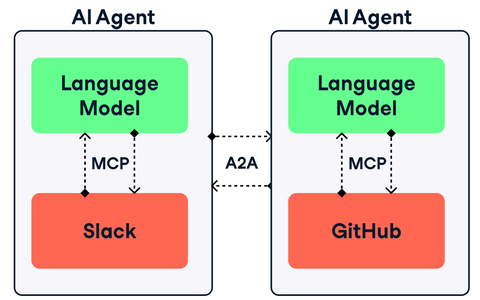
Agent2Agent (A2A) Protocol

Google's Agent2Agent (A2A) protocol enables communication and collaboration between AI agents.

Example: The same AI assistant in Slack, after fetching Asana updates, identifies a critical task that needs a specific report generated. Instead of doing it itself, it uses A2A to securely "talk" to a specialized "Reporting Agent" (which might live on a separate server).

Complementary Protocols

Our example shows that MCP and A2A are complementary protocols that aim to create a more interconnected and powerful AI agent ecosystem.



6 Building AI Agents (What to Choose Depending on Your Goals and Skills)

One-Prompt Agents

One-prompt agents primarily rely on a single, well-crafted prompt to guide the language model's behavior and output for straightforward tasks, minimizing complex multi-step reasoning or tool use.

Use cases: Generating reports, answering complex questions, or performing actions like booking tickets or buying groceries from an online store.

Ease of use: Easy, primarily requiring basic prompt engineering skills.

Manus · Project Mariner · Operator · Perplexity Labs

Coding Agents

Coding agents are AI agents specifically designed to assist with or perform coding tasks, such as generating code snippets, debugging, or refactoring.

Use cases: Automating development tasks, assisting software engineers with code generation or analysis, or enabling natural language programming.

Ease of use: Easy to hard, depending on the required programming knowledge and complexity of the task.

Codex · Google Jules · Devin · Replit · Lovable · Firebase Studio

Workflow-Based Agents

Workflow-based agents offer pre-built functionalities or visual interfaces for constructing agent workflows, requiring little or no coding. These are often geared towards automating specific business processes.

Use cases: Automating repetitive business processes, customer service triage, data entry, or other operational tasks that can be defined through a structured workflow.

Ease of use: Medium, as it requires understanding the workflow logic and configuration.

n8n · Make · Dify · FloWise · Langflow · AirOps

Agentic Frameworks

Agentic frameworks are software libraries and platforms that provide structures, tools, and best practices to help developers build, deploy, and manage AI agents, abstracting away some complexities.

Use cases: Developing custom, complex AI agent applications that require significant integration, specialized logic, or novel architectures.

Ease of use: Generally medium to hard, depending on framework complexity and desired customization.

OpenAI Agents SDK · Google's ADK · LangGraph · CrewAI · Smolagents · LlamaIndex

datacamp

Building AI Agents with Haystack

Getting started with Agents Cheat Sheet

Learn Python online at www.DataCamp.com

> What you learn in the course

Goal: Build a healthcare agent that answers user questions by combining real-time web info with private, structured patient data.

Stack: Haystack (open-source AI orchestration), OpenAI LLMs, Serper (web search), SQLite+pandas (patient DB).

Modularity: Model-agnostic components; tools can be APIs, Python functions, Haystack components, pipelines, other agents, or MCP servers.

> Key definitions

- **LLM:** Large Language Model that generates and reasons over text.
- **Agent:** an autonomous system powered by an LLM that can interpret user requests, decide on actions, and carry them out using tools, going beyond just responding with text.
- **Tool:** A callable capability the agent can invoke (API, component, pipeline, function).
- **Function calling/tool use:** LLM emits structured calls that the runtime executes, feeding results back to the model.
- **Pipeline:** Directed graph of components for deterministic, repeatable workflows.
- **Component:** The unit of a pipeline, with a run() method (e.g., OpenAIChatGenerator, SerperDevWebSearch).
- **Document object:** Standard result unit (id, content, title, url, metadata) returned by retrievers/search.
- **State schema:** Structured storage of agent state across steps.
- **Streaming callback:** Hook to stream intermediate reasoning, tool calls, and outputs.
- **Groundedness:** How well outputs are supported by retrieved or tool-returned evidence.

> When to use agents vs pipelines

- Use an agent when: open-ended queries; uncertain paths; multi-step reasoning; tool selection needed.
- Use a pipeline when: steps are fixed, deterministic, and predictable (e.g., text-to-SQL, batch data processing).
- Best of both: Wrap pipelines as tools for agents to call.

> Setup and environment

- Install Haystack and dependencies (done per exercise/notebook).
- Environment variables:
  - `SERPERDEV_API_KEY` for Serper web search.
  - `OPENAI_API_KEY` in real projects (course environment preconfigured).
- Data: CSV with patient records loaded to an in-memory SQLite table (patients).

> LLM Basics with Haystack

- `Component: OpenAIChatGenerator`
- `Messages: Use ChatMessage.from_user() and ChatMessage.from_system() to structure roles.`
- `System messages steer tone, style, and behavior.`
- `run() returns a list of ChatMessage; read replies[0].text for content.`

Code sketch: chat generator

- `Initialize chat generator with your chosen chat model.`
- `Create messages: system then user.`
- `Call run(messages=[...]) and read replies.`

Building your first agent (no tools)

- `Agent mirrors chat generator behavior but can later call tools.`
- `Pass system_prompt to define the agent's identity/behavior.`
- `last_message holds the agent's latest reply for easy access.`

`Note: Without tools, prefer using ChatGenerator directly for efficiency.`

> LLM Basics with Haystack

Code sketch: minimal agent

- `agent = Agent(chat_generator=OpenAIChatGenerator(...), system_prompt="...")`
- `agent.run(messages=[ChatMessage.from_user("...")])`
- `agent.last_message.text`

Adding web search (Serper) as a tool

- `Component: SerperDevWebSearch()`
- `Test the component directly via run(query="...").`
- `Wrap it with ComponentTool() to auto-generate name, description, parameters.`
- `Pass tool to the agent via tools=[...]; update system prompt to allow internet use.`
- `The agent decides when to call the tool for up-to-date info and cites sources.`

Code sketch: web search tool and agent

- `search = SerperDevWebSearch()`
- `search_tool = ComponentTool(component=search) // auto name/desc/params from docstring`
- `agent = Agent(chat_generator=..., system_prompt="You can search the web for current info.", tools=[search_tool])`
- `agent.run(messages=[ChatMessage.from_user("...")])`

Making tools and agents transparent and efficient

- `Customize tool metadata:`
  - `name, description, parameters: improve LLM understanding and correct usage.`
- `outputs_to_string: pass only essential info to the LLM to save context (e.g., snippet + URL).`
  - `Define a doc_to_string(docs) to format documents to concise strings.`
- `outputs_to_state: persist tool outputs (e.g., documents) into agent state for inspection/evaluation.`
- `Agent state_schema: declare types for stored state (e.g., list[Document]).`
- `streaming_callback: use print_streaming_chunk() to stream tool calls, inputs, outputs, and final answer.`

Code sketch: tool I/O shaping + tracing

- `def doc_to_string(docs): return "\n".join(f"- {d.content}\nSource: {d.meta.get('url')}" for d in docs)`
- `search_tool = ComponentTool(component=search, outputs_to_string=doc_to_string, outputs_to_state={"documents": "documents"})`
- `agent = Agent(generator=..., system_prompt="...", tools=[search_tool], state_schema={"documents": list}, streaming_callback=print_streaming_chunk)`
- `agent.run(messages=[ChatMessage.from_user("...")])`

Pipelines for deterministic tasks (text-to-SQL)

- `Build a small, in-memory SQLite DB from CSV using pandas and sqlite3.`
- `Components:`
  - `OpenAIChatGenerator: generates SQL from the prompt.`
  - `Custom SQLConnector: executes SQL and returns results.`
- `Wire components in a Pipeline: prompt_builder -> chat_generator -> sql_connector.`
- `Run with include_outputs_from to also return the generated SQL.`

Code sketch: DB and pipeline

- `Load CSV to pandas DataFrame; write to sqlite3 connection as table "patients".`
- `@component class SQLConnector:`
  - `init(conn): keep DB connection.`
  - `run(self, llm_replies): extract SQL from llm_replies, strip backticks, execute via pandas.read_sql, return {"results": rows_as_strings}`
- `prompt = ChatPromptBuilder(template="""`
  `You are a text-to-SQL assistant...`
  `Table: patients(columns: name, age, condition, medication, ...)`
  `Use SQLite dialect. Return only SQL between triple backticks.`
  `Question: {{query}}`
  `""")`
- `pipe = Pipeline()`
  - `add_component("prompt_builder", prompt)`
  - `add_component("chat_generator", OpenAIChatGenerator(...))`
  - `add_component("sql_connector", SQLConnector(your_database.db))`
  - `connect("prompt_builder", "chat_generator")`
  - `connect("chat_generator.replies", "sql_connector.llm_replies")`
- `pipe.show() // visualize`
- `out = pipe.run(inputs={"prompt_builder": {"query": "Most common diagnosis > 60?"}}, include_outputs_from=["chat_generator"])`

> LLM Basics with Haystack

Wrapping pipelines as tools and integrating into the agent

- `Use @tool on a Python function to expose the pipeline as a tool class.`
- `Function signature: get_patient_information(query: str) -> dict; runs the pipeline and returns results.`
- `Provide a clear docstring and parameter descriptions; the agent uses them to decide when/how to call the tool.`
- `Update agent's system prompt with tool usage guidelines and examples:`
  - `Use SQL tool for patient- or database-related queries.`
  - `Use web search for general medical knowledge or drug interactions.`
- `Initialize the final agent with both tools and streaming enabled.`

Code sketch: pipeline tool + final agent

- `@tool`
  - `def get_patient_information(query: str) -> dict:`
    - `"Fetch structured patient info from the patients SQLite DB based on a natural language query."`
    - `return pipe.run({"query": query})`
- `final_agent = Agent(chat_generator=..., system_prompt="""`
  `You are a healthcare agent.`
- `For patient/database questions: call get_patient_information.`
- `For medical knowledge/drug interactions: call web_search.`
  `Cite sources when using web data.`
  `""", tools=[search_tool, get_patient_information], streaming_callback=print_streaming_chunk)`
- `final_agent.run(messages=[ChatMessage.from_user("...")])`

Prompting patterns and guidance

- `System prompt content`
  - `Role: "You are a helpful, concise healthcare agent."`
  - `Tool policy: Clear criteria for when to use each tool.`
  - `Output policy: Cite URLs for web data; summarize SQL results clearly.`
  - `Safety: Avoid making medical decisions; present information with sources.`
- `Message sequencing`
  - `System message first; user message follows; agent/tool calls streamed.`
- `Examples in prompt help the LLM learn tool selection behavior.`

Observability and evaluation

- `Use streaming_callback to trace:`
  - `Tool selected, parameters used, returned outputs, final synthesis.`
- `Save key evidence in state (e.g., documents) for post-hoc checks.`
- `Failure modes to watch:`
  - `Wrong tool selection (e.g., using web search for DB questions).`
  - `Incorrect tool parameters (bad queries).`
  - `Ignoring relevant tool outputs.`
  - `Overlong tool outputs causing context overflow.`

Tips and pitfalls

- Keep tool outputs concise (use outputs_to_string) to preserve context.
- Give tools clear names, descriptions, and parameter schemas.
- Provide the DB schema, column types, and example values in the text-to-SQL prompt.
- Strip formatting (e.g., ```sql) before executing generated SQL.
- Prefer read-only DB queries; validate or sandbox SQL execution.
- Add few-shot examples in prompts to improve SQL accuracy and tool choice.
- Use an agent only when needed; default to pipelines for deterministic flows.

Extensions and next steps

- Add more tools: retrieval over private documents, calculators, specialty APIs.
- Multi-agent systems: specialized agents (e.g., triage, literature review).
- Multimodal: images, PDFs, tables as inputs/outputs.
- Productionization: monitoring, evals, and deployment with Hayhooks.
- Connect to MCP servers for distributed or externalized capabilities.

Minimal end-to-end flow recap

- Start: User asks a healthcare question.
- Agent reasons: Do I need web, SQL, both?
- Tool use:
- Web search tool via Serper returns Documents (formatted snippets + URLs).
- SQL pipeline tool returns structured results from patients DB.
- Synthesis: Agent cites sources and summarizes patient data.
- Transparency: Streaming trace + saved state for groundedness and debugging.

datacamp

Building Agentic Workflows with LlamaIndex

Getting started with Agents Cheat Sheet

Learn Python online at www.DataCamp.com

> Overview

- **Goal:** Build agentic workflows (agents that plan, use tools, remember, and collaborate) in LlamaIndex.
- **When to use agents vs direct LLM:**
  - Direct LLM: single-shot, self-contained tasks (summarize, translate).
  - Agent: multi-step, conditional, tool-using, or information-fetching tasks (research, reports).
- **Core building blocks:**
  - AgentWorkflow: High-level orchestration for single or multi-agent systems.
  - FunctionAgent: Specialized, role-focused agent with tools and prompts.
  - Context: Shared state and event stream across runs and agents.
  - Tools: Async functions with types and docstrings that agents can call.

> Setup and Keys

- Install LlamaIndex + Tavily client (and any provider SDKs).
- Use environment variables for keys (never hard-code):
- OPENAI_API_KEY
- TAVILY_API_KEY
- Model providers: LlamaIndex supports 80+ providers and 400+ models (OpenAI as default in course; local models supported).

> Key Concepts

- Agent: LLM-driven system that decides, plans, and executes actions (including actions that need external tools) to achieve an objective defined by the user.
- Tools:. A function or external capability that an agent can invoke to perform a specific task or retrieve information during its reasoning process.
- Context: Persistent memory/state across runs (messages, artifacts, preferences).
- Events and streaming:
  - AgentStream: incremental text chunks from the LLM.
  - AgentInput: an agent (or sub-agent) receives work.
  - AgentOutput: an agent produces an answer (intermediate or final).
  - ToolCall and ToolCallResult: a tool is invoked and returns.

> Building Agentic Workflows

Build a Single-Agent with a Web Search Tool

- Define a tool (Tavily-based):
  - Async function search_web(query: str) -> str
  - Docstring: “Useful for using the web to answer questions.”
  - Inside: create Tavily client with key; call search; return string result.
- Create the agent:
  - AgentWorkflow.from_tools_or_functions([search_web], llm=..., system_prompt="You answer questions; use web search if needed.")
- Run:
  - response = await workflow.run("What's the weather in San Francisco?")
  - The agent decides to call search_web and returns a live result.

Add Memory with Context

- Default agents are stateless between runs.
- Use Context to persist state/conversation:
  - ctx = Context(workflow)
  - First run: await workflow.run("My name is Laurie, nice to meet you!", ctx=ctx)
  - Store structured state (e.g., ctx.state["name"] = "Laurie")
  - Second run: await workflow.run("What is my name?", ctx=ctx) -> “Your name is Laurie.”
- Access state:
  - state = await ctx.get("state") # returns a dict-like state store

> Building Agentic Workflows

Streaming and Event Tracing

- Purpose: improved UX (token streaming), observability, and debugging.
- Usage:
  - handler = await workflow.run("Your prompt", ctx=ctx)
  - for event in handler.stream_events():
    - if AgentStream: print(event.delta, end="")
    - if ToolCall/ToolCallResult: log decisions and results
    - if AgentInput/AgentOutput: trace multi-agent handoffs
- Tip: selective logging makes internal reasoning and tool calls transparent.

Multi-Agent Systems: When and Why

- Use multiple specialized agents when:
  - Tasks naturally decompose (research → write → review).
  - One agent struggles with many tools or long prompts.
  - Parallelization is beneficial.
- Common pattern:
  - ResearchAgent: gathers facts (search_web, record_notes).
  - WriteAgent: drafts report (write_report).
  - ReviewAgent: critiques/approves (review_report).

Build a Multi-Agent Workflow (FunctionAgent + AgentWorkflow)

- Define agents (FunctionAgent):
  - name: "ResearchAgent" | "WriteAgent" | "ReviewAgent"
  - description: routing hint for the system
  - system_prompt: role-specific instructions
  - tools: assign minimal, relevant tools per role
- Handoff hints:
  - ResearchAgent.can_handoff_to = ["WriteAgent"]
  - WriteAgent.can_handoff_to = ["ReviewAgent"]
- Orchestrate:
  - workflow = AgentWorkflow(agents=[research, write, review], root_agent="ResearchAgent")
  - Initialize shared state in Context:
    - state = { "research_notes": [], "report_content": "", "review_feedback": "" }
- Example flow:
  - ResearchAgent → search_web multiple times → record_notes to context
  - WriteAgent → reads notes → composes and saves draft to report_content
  - ReviewAgent → critiques/report approval → writes feedback

Custom Workflows (Fine-Grained Control)

- Use when you need custom control beyond AgentWorkflow.
- Structure:
- Subclass Workflow
  - Define typed async steps with @step
  - Use typed Events to carry data between steps
  - StartEvent: auto-emitted at run start
  - StopEvent: final result
- Example (single-step):
  - step: StartEvent → StopEvent with "Hello, world!"
- Example (multi-step pipeline):
  - FirstEvent(first_output: str)
- SecondEvent(second_output: str)
  - step_one: StartEvent → FirstEvent("First step complete.")
  - step_two: FirstEvent → SecondEvent("Second step complete.")
  - step_three: SecondEvent → StopEvent("Workflow complete.")

Loops and Branching

- Looping:
  - Emit an event type handled by an earlier step (e.g., LoopEvent) to re-run prior logic (reflection, retries).
- Branching:
  - Step emits different event types based on conditions (like if/else), routing to different downstream steps.

Parallelism and collect_events

- Concurrent execution: launch multiple steps simultaneously.
- Default behavior: workflow halts on first StopEvent.
- Wait for many:
  - Use Context.collect_events(...) to wait for:
    - a specific count of the same event type, or
    - a sequence of different event types in a specified order
  - If not enough events yet, collect_events returns None; when ready, returns an array of events ordered by arrival (or by the specified sequence for heterogeneous types).
- Stream your own events for observability:
  - Context.write_event_to_stream(...) to emit custom logs/tokens.

> Building Agentic Workflows

Deep Research System (End-to-End)

- Agents:
  - QuestionAgent: generate list of research questions for a topic.
  - AnswerAgent: answer each question via search_web; store in state["answers"].
  - ReportAgent: compile Q&A pairs into a final report.
- Orchestration:
  - Initialize context with state["research_topic"], state["questions"] = [], state["answers"] = []
  - Run QuestionAgent once → populate questions
  - Loop: for each question → run AnswerAgent → append to answers
  - Run ReportAgent → assemble final report from answers
- Behavior:
  - Supports streaming for transparency and better UX.

Add Self-Reflection (Review and Rewrite Loop)

- Motivation: first drafts can be incomplete or inaccurate.
- Mechanism:
  - WriteAgent handles WriteEvent and RewriteEvent (rewrite applies review feedback to the draft).
  - ReviewAgent analyzes report; if insufficient, emits RewriteEvent and feedback.
  - Gate decision: LLM (or rule) classifies review as “good” (stop) vs “bad” (rewrite).
  - Loop: write → review → optional rewrite until acceptable; then StopEvent.
- Visualize: use the workflow visualizer to see event flow across write/review/rewrite.

Tool Design Best Practices

- Use async functions for tools.
- Provide precise type hints and descriptive docstrings.
- Keep tool names intuitive (search_web, record_notes, write_report).
- Limit tool count per agent to reduce confusion and improve routing.
- Validate and sanitize tool inputs; handle errors gracefully.

Prompts and Roles

- System prompts act as job descriptions:
  - ResearchAgent: “Search the web for the topic; record concise, sourced notes.”
  - WriteAgent: “Draft a clear, structured report from research notes; store in report_content.”
  - ReviewAgent: “Critique accuracy, coverage, clarity; propose revisions or approve.”
- Include guardrails: cite sources, avoid hallucinations, be concise.

Memory and State Tips

- Explicitly persist key fields in ctx.state (notes, drafts, feedback, user preferences).
- Read/modify only the fields relevant to each role.
- Treat ctx.state as the canonical shared memory across agents.

Streaming and Observability Tips

- Stream AgentStream text to provide immediate feedback to users.
- Log ToolCall/ToolCallResult for traceability.
- Use AgentInput/AgentOutput to understand handoffs in multi-agent systems.
- Write custom events to the stream for critical milestones.

Tips & Pitfalls

- Don’t overload a single agent with too many tools or responsibilities.
- Always pass ctx=ctx when you expect stateful behavior; otherwise runs are stateless.
- Keep prompts short and role-specific; avoid one mega-prompt.
- Use environment variables for keys; never commit secrets.
- In notebooks, avoid direct asyncio.run conflicts; use await or other framework-provided run methods.
- For parallel steps, use collect_events to wait for all necessary outputs.
- Visualize early; event graphs reveal bottlenecks and logic errors.

Troubleshooting

Agent won’t use a tool:
- Check tool name, docstring relevance, and type hints.
- Ensure the system prompt mentions when to use the tool.

Memory not persisting:
- Verify ctx is reused and passed into every run.
- Confirm writes to ctx.state and reads are correct.

Streaming doesn’t show:
- Use run and iterate events; print AgentStream.delta.
- Multi-agent dead ends:
- Check can_handoff_to hints and root_agent.
- Ensure required state fields exist before downstream agents read them.

Building Multi-Agent Systems with LangGraph

Getting started with Agents Cheat Sheet

Learn Python online at www.DataCamp.com

> Overview

Goal: Build an agentic assistant that researches Fortune 500 companies, loads local stock CSVs, and generates plots via Python.
Tech stack:
- LangGraph for orchestration (nodes, edges, conditional routing, memory).

Path to production:
1. Single LLM node (no tools) → 2) Linear LLM→Tool flow → 3) Conditional tool calling → 4) High-level single-agent shortcut → 5) Multi-agent: Swarm → 6) Multi-agent: Supervisor.

> Core Concepts and Definitions

- Agent: LLM + ability to take actions via tools; orchestrated by a workflow.
- Tool: A callable function the LLM can invoke to act in the real world (APIs, DBs, code).
- Orchestration (LangGraph): Graph of nodes and edges describing control flow.
- Node: Executable unit (e.g., LLM node, Tool node).
- Edge: Directed path between nodes.
- Conditional edge: Branches based on a predicate (e.g., whether the LLM requested a tool call).
- State: Shared memory that persists messages across node executions.
- Tool-aware LLM: LLM bound to a set of tools (via .bind_tools) with access to tool names and descriptions.
- Swarm multi-agent: Multiple specialized agents with handoff tools; may loop between agents.
- Supervisor multi-agent: A supervisor agent delegates to workers and owns the final response.

> Tools: Design and Implementation

Best practice: Use clear, specific docstrings. The `@tool` decorator turns function name and docstring into the tool's name and description exposed to the LLM.
Wikipedia summary tool:
- Purpose: Retrieve an 8-sentence summary for a query. Notes: Uses `wikipedia.search()` then `wikipedia.summary(top_result, sentences=8)`.

Example Snippet
```
from langchain.tools import tool
import wikipedia

@tool
def wikipedia_tool(query: str) -> str:
    """Search Wikipedia for a query and return an 8-sentence summary of the top result."""
    title = wikipedia.search(query)[0]
    return wikipedia.summary(title, sentences=8)
```

Stock data tool:
- Purpose: Load last N trading days from local .csv and return as markdown table.
- Notes: CSV exists only for selected tickers. Trading days only (weekends omitted), so `num_days` may return fewer rows than calendar days. Use pandas and `DataFrame.to_markdown()`.

Example Snippet
```
from langchain.tools import tool
import os, pandas as pd

@tool
def stock_data_tool(ticker: str, num_days: int) -> str:
    """Return last N trading days of OHLCV data for ticker from local CSV as a markdown table."""
    path = f"{ticker}.csv"
    if not os.path.exists(path):
        return f"No CSV found for ticker {ticker}."
    df = pd.read_csv(path, parse_dates=["Date"]).sort_values("Date")
    num_days = min(num_days, len(df))
    out = df.tail(num_days)
    return out.to_markdown(index=False)
```

Python execution tool:
- Purpose: Execute arbitrary Python to analyze/plot stock data.
- Notes: Use `langchain_experimental.tools.PythonREPL`. Dangerous: can run arbitrary code. Add guardrails in real systems.

Example Snippet
```
from langchain.tools import tool
from langchain_experimental.tools import PythonREPL
_py = PythonREPL()

@tool
def python_tool(code: str) -> str:
    """Execute Python code in an isolated REPL. Use for plotting/analysis with prior data."""
    return _py.run(code)
```

> State and Memory

- Use a TypedDict to define graph-wide state. Include messages and configure message appending.
- add_messages() ensures new messages are appended, not replaced.

Example:
```
from typing import List
from typing_extensions import TypedDict
from langgraph.graph.message import add_messages
from langchain_core.messages import BaseMessage
```

> Single-Agent: Minimal LLM-only Graph

- Build a graph with one LLM node; no tools yet.

Steps:
- Define LLM (e.g., ChatOpenAI).
- Node function llm_node(state) → invoke LLM with state["messages"]; return {"messages": [llm_output]}.
- Add node, connect START→LLM→END, compile.

Notes:
- Re-running .add_node with same name raises "node already exists"; re-create graph before re-adding.

Sketch:
```
from langgraph.graph import StateGraph, START, END
from langchain_openai import ChatOpenAI

llm = ChatOpenAI()  # env preconfigured in course

def llm_node(state: State):
    msg = llm.invoke(state["messages"])
    return {"messages": [msg]}

graph = StateGraph(State)
graph.add_node("llm", llm_node)
graph.add_edge(START, "llm")
graph.add_edge("llm", END)
app = graph.compile()
```

> Single-Agent: Conditional Tool-Calling

- Keep LLM tool-aware (bind_tools).
- Add ToolNode as before.
- Use add_conditional_edges("llm", tools_condition, ["tools", END]) to branch:
  - If the last LLM message requests a tool call → "tools".
  - Otherwise → END.
- Also add edge "tools"→"llm" to allow iterative cycles until task completion.

Benefits:
- Tools called only when needed.
- Supports iterative tool use (e.g., fetch data, then plot).

Sketch:
```
from langgraph.prebuilt import ToolNode, tools_condition

tools = [wikipedia_tool, stock_data_tool, python_tool]
llm = ChatOpenAI().bind_tools(tools)

def llm_node(state: State):
    return {"messages": [llm.invoke(state["messages"])]}

graph = StateGraph(State)
graph.add_node("llm", llm_node)
graph.add_node("tools", ToolNode(tools))
graph.add_edge(START, "llm")
graph.add_conditional_edges("llm", tools_condition, ["tools", END])
graph.add_edge("tools", "llm")
app = graph.compile()
```

> High-level Shortcut: create_react_agent

- Use for a standard single agent without verbose graph code.
- Provide LLM, tools, optional name and prompt (system message).
- Trade-off: Faster to build, less customizable (e.g., cannot change routing predicate easily).

Sketch:
```
from langgraph.prebuilt import create_react_agent

agent = create_react_agent(
    llm=ChatOpenAI(),
    tools=[wikipedia_tool, stock_data_tool, python_tool],
    name="company-agent",
    prompt="You are a research assistant. Use tools when needed; otherwise answer concisely."
)
# agent is an app you can visualize/invoke like the previous graph
```

> Multi-Agent Patterns

Swarm (Network) Multi-Agent

Two specialized agents:
- Researcher: summaries + stock data.
- Analyst: plotting via Python.

- Add "handoff tools" so agents can pass control to each other when appropriate.

> Multi-Agent Patterns

Swarm (Network) Multi-Agent

Key steps:
- Create handoff tools with create_handoff_tool(agent_name, description).
- Build each agent with create_react_agent, including handoff tool + its functional tools.
- Use create_swarm([agents], entrypoint="researcher") to orchestrate.
- Use InMemorySaver as checkpointer; supply config on invoke/stream.

Pros:
- Natural extension of single-agent with clear specializations.

Cons:
- Risk of infinite handoff loops.
- Requires choosing an entrypoint agent.

Sketch:
```
from langgraph_swarm import create_handoff_tool, create_swarm
from langgraph.checkpoint.memory import InMemorySaver

to_analyst = create_handoff_tool(agent_name="analyst",
    description="Handoff when data is ready or visualization requested.")
to_researcher = create_handoff_tool(agent_name="researcher",
    description="Handoff when data gathering is needed or missing.")

researcher = create_react_agent(
    llm=ChatOpenAI(),
    tools=[wikipedia_tool, stock_data_tool, to_analyst],
    name="researcher",
    prompt="Retrieve company info and stock data. Handoff when plotting is needed."
)

analyst = create_react_agent(
    llm=ChatOpenAI(),
    tools=[python_tool, to_researcher],
    name="analyst",
    prompt="Generate plots and analysis using available data. Handoff if data missing."
)

memory = InMemorySaver()
swarm = create_swarm(agents=[researcher, analyst], entrypoint="researcher")
app = swarm.compile(checkpointer=memory)

config = {"configurable": {"thread_id": "session-1"}}
# app.invoke(input, config=config)
```

Supervisor Multi-Agent

- Add a supervisor that delegates tasks and produces the final user response.
- Workers respond back to supervisor; no direct worker→worker edges.

Key steps:
- Adjust worker prompts to "report results to supervisor" (no direct user responses).
- Define supervisor with create_supervisor(llm, workers=[...], prompt=..., add_handoff_back_messages=True, output_mode="all").
- Use same memory/checkpointer pattern; invoke with config.

Pros:
- Reduces looping; centralized decision making.
- Scales to more workers.

Cons:
- More centralized complexity in supervisor prompt/design.

Sketch:
```
from langgraph_supervisor import create_supervisor

researcher = create_react_agent(
    llm=ChatOpenAI(),
    tools=[wikipedia_tool, stock_data_tool],
    name="researcher",
    prompt="Retrieve data and return results to the supervisor only."
)

analyst = create_react_agent(
    llm=ChatOpenAI(),
    tools=[python_tool],
    name="analyst",
    prompt="Generate plots using available data and return results to the supervisor only."
)

supervisor = create_supervisor(
    llm=ChatOpenAI(),
    workers=[researcher, analyst],
    prompt="Delegate tasks to workers; coordinate data→plot flow; produce final answer.",
    add_handoff_back_messages=True,
    output_mode="all"
)

memory = InMemorySaver()
app = supervisor.compile(checkpointer=memory)

config = {"configurable": {"thread_id": "session-2"}}
# app.invoke(input, config=config)
```

Tips, Pitfalls, and best Practices

- Tool docstrings matter:
  - Be explicit about inputs, outputs, and when to use the tool; improves tool selection reliability.
- PythonREPL safety:
  - Capabilities are broad; add guardrails (whitelists, timeouts, sandboxes) in production.
- Data dependencies:
  - Use conditional edges to support iterative multi-tool workflows (data→analysis).
- Linear graphs are brittle:
  - Avoid unconditional tool execution; prefer conditional routing.
- Infinite loops (swarm):
  - Add loop limits or termination criteria; detect no-progress cycles.
- Choosing entrypoint (swarm):
  - Start with the agent that can make first progress (e.g., research before analysis).
- Memory and config:
  - With InMemorySaver, always pass config (e.g., thread_id) on invoke/stream to keep histories separated.
- Re-running graph definitions:
  - If "node already exists," re-create the graph object before re-adding nodes/edges.

datacamp

Text-to-Query Agents with MongoDB, LangChain, and LangGraph

Getting started with Agents Cheat Sheet

Learn Python online at www.DataCamp.com

> Definitions

- Text-to-query agent: Converts natural language into database queries and answers from results.
- RAG (Retrieval Augmented Generation): Retrieve external data, augment the prompt, generate an answer with an LLM.
- AI agent: An LLM that can decide to call tools (functions) to act (e.g., read schema, run queries).
- MongoDB Atlas: Fully managed MongoDB in the cloud (cluster = managed database deployment).
- MongoDB Query API: Query via JSON-like objects in code (find, insert, update, delete; aggregate pipelines).
- LangChain: Library for LLMs, prompts, and tool integration.
- LangGraph: Agent workflow orchestration using graphs (nodes, edges, shared state).
- Short-term memory: Per-conversation/task state. Disappears unless checkpointed.
- Checkpoints/threads: Save/restore agent state across runs (thread = storyline; checkpoints = savepoints).

> Setup and Connection

- Create free MongoDB Atlas cluster; preload sample dataset; create DB user; copy credentials.
- Open Network Access to Datacamp/remote (Allow Access From Anywhere for test only).
- Store connection string as MONGODB_URI environment variable.

```
# Install
# pip install pymongo langchain-openai langchain-mongodb langgraph

import os
from pymongo import MongoClient

uri = os.environ["MONGODB_URI"]
client = MongoClient(uri)
client.admin.command("ping")  # Verify connection

db = client["sample_mflix"]
movies = db["movies"]
```

> MongoDB Query API Essentials

- Collections ~ tables; documents ~ JSON-like rows; flexible schema.
- Reading documents:

```
# Preview one document
doc = movies.find_one({})

# Simple filter: all Romance movies
cursor = movies.find({"genres": "Romance"})

# Multi-condition: Romance movies in the 1990s
query = {
  "$and": [
    {"genres": "Romance"},
    {"year": {"$gte": 1990, "$lt": 2000}}
  ]
}
cursor = movies.find(query)
for d in cursor:
    print(d)
```

- Operators: `$and, $or, $gte, $gt, $lte, $lt, $in, $nin, $exists`, etc.

> Aggregation Pipelines

- Pipelines are ordered lists of stages transforming data.

```
# Latest 5 releases (title only)
pipeline = [
  {"$sort": {"released": -1}},
  {"$limit": 5},
  {"$project": {"title": 1, "_id": 0}}
]
for d in movies.aggregate(pipeline):
    print(d)

# Top 5 by IMDB rating (title only)
pipeline = [
  {"$sort": {"imdb.rating": -1}},
  {"$limit": 5},
  {"$project": {"title": 1, "_id": 0}}
]

# Count movies by genre (>= 50)
pipeline = [
  {"$unwind": "$genres"},
  {"$group": {"_id": "$genres", "numMovies": {"$sum": 1}}},
  {"$match": {"numMovies": {"$gte": 50}}},
  {"$sort": {"numMovies": -1}}
]

# Top 5 directors by avg IMDB rating, min 21 films
pipeline = [
  {"$unwind": "$directors"},
  {"$group": {
    "_id": "$directors",
    "filmCount": {"$sum": 1},
    "avgRating": {"$avg": "$imdb.rating"}
  }},
  {"$match": {"filmCount": {"$gt": 20}}},
  {"$sort": {"avgRating": -1}},
  {"$limit": 5}
]
```

> MongoDB Database Toolkit (LangChain Tools)

Tools (from langchain-mongodb):

- mongodb_list_collections: lists DB collections.
- mongodb_schema: schema + sample docs for given collection(s).
- mongodb_query_checker: validates/fixes MongoDB query text.
- mongodb_query: executes a Query API call and returns results.

```
# pip install langchain-mongodb
from langchain_mongodb import MongoDBDatabase, MongoDBDatabaseToolkit

db_lc = MongoDBDatabase.from_connection_string(os.environ["MONGODB_URI"], db_name="sample_mflix")
toolkit = MongoDBDatabaseToolkit(db=db_lc, llm=llm)
tools = {t.name: t for t in toolkit.get_tools()}

# Explore tools
print(tools["mongodb_list_collections"].description)

# Calls
tools["mongodb_list_collections"].invoke("")  # -> "movies, theaters, users, ..."
tools["mongodb_schema"].invoke("movies")      # -> schema + samples
fixed = tools["mongodb_query_checker"].invoke('{"find":"movies","filter":{"genres":"Romnce"}}')
result = tools["mongodb_query"].invoke('{"aggregate":"movies","pipeline":[{"$limit":3}]}')
```

> Prompting the Tool-Augmented LLM

- Use the system prompt from langchain-mongodb (includes how to form queries and use tools).
- Bind tools to the LLM; set defaults like top_k.

```
# Example shape (names may vary by package version)
from langchain_mongodb.prompts import mongodb_system_prompt  # provided by the package
from langchain_core.tools import tool

prompt = ChatPromptTemplate.from_messages([
  ("system", mongodb_system_prompt),
  ("system", "Tools available: {tool_names}"),
  MessagesPlaceholder("messages")
]).partial(top_k="5", tool_names=", ".join(tools))

llm_with_tools = llm.bind_tools(list(tools.values()))
chain = prompt | llm_with_tools

resp = chain.invoke({"messages": [("human", "Give the top 5 latest movie releases.")]})
print(resp.tool_calls)  # Typically starts with mongodb_list_collections
```

> Example End-to-End Prompts

Latest releases:

- Human: “Give the top 5 latest movie releases.”
- Agent: list collections → load schema → write/validate query → execute → answer.

Complex aggregation:

- Human: “Top 5 directors by average IMDB rating with more than 20 movies.”
- Agent: schema → write pipeline with $unwind/$group/$match/$sort/$limit → execute → summarize.

Follow-up with memory:

- Human: “Which states have the most theaters?”
- Human: “How many theaters does California have?” (same thread_id; uses memory, no new tools if cached)

> Orchestrating the Agent with LangGraph

- Graph = nodes + edges + shared state (messages).
- Agent node: LLM decides if tools are required.
- Tools node: Executes tool_calls produced by LLM.
- Conditional routing: continue calling tools until final answer.

```
# pip install langgraph
from typing import TypedDict, Annotated, List
from langgraph.graph import StateGraph, START, END
from langgraph.graph.message import add_messages
from langgraph.prebuilt import tools_condition

class GraphState(TypedDict):
    messages: Annotated[List, add_messages]

def agent_node(state: GraphState):
    res = llm_with_tools.invoke({"messages": state["messages"]})
    return {"messages": [res]}

def tools_node(state: GraphState):
    last = state["messages"][-1]
    outputs = []
    for call in getattr(last, "tool_calls", []):
        name = call["name"]
        args = call.get("args", "")
        out = tools[name].invoke(args)
        outputs.append(("tool", f"{name} -> {out}"))
    return {"messages": outputs}

graph = StateGraph(GraphState)
graph.add_node("agent", agent_node)
graph.add_node("tools", tools_node)
graph.add_edge(START, "agent")
graph.add_edge("tools", "agent")
graph.add_conditional_edges("agent", tools_condition, {"tools": "tools", "end": END})

app = graph.compile()

# Stream a run
for step in app.stream({"messages": [("human", "List top 5 latest releases")]}, stream_mode="values"):
    pass  # pretty_print(step["messages"][-1]) if desired
```

> Short-Term Memory with Checkpoints and Threads

- Problem: short-term state vanishes between runs.
- Solution: persist checkpoints in MongoDB for durability and resume.

```
# Persist checkpoints in MongoDB
from langgraph.checkpoint.mongodb import MongoDBSaver

checkpointer = MongoDBSaver(client)  # reuse your MongoClient
app = graph.compile(checkpointer=checkpointer)

# Use thread_id to resume the same conversation
config = {"configurable": {"thread_id": "user-123"}}

# Turn 1
for _ in app.stream({"messages": [("human", "Which states have the most theaters?")]}, config,
stream_mode="values"):
    pass

# Turn 2 (follow-up; reuses memory, often no new tool calls)
for _ in app.stream({"messages": [("human", "How many theaters does California have?")]}, config,
stream_mode="values"):
    pass

# Inspect/restore checkpoints
# checkpointer.list(config, limit=5)
# cp = checkpointer.get(config)
# app = graph.compile(checkpointer=checkpointer, checkpoint=cp)
```

> Tips and Pitfalls

- **Security:**
  - Never hard-code credentials; use environment variables.
  - “Allow Access From Anywhere” only for test. Restrict IPs in production.
- **Query correctness:**
  - Prefer mongodb_query_checker before execution.
  - Use correct field paths (e.g., "imdb.rating"); prefix fields with $ in $group _id and expressions.
  - $project to exclude _id when not needed.
- **Tool usage:**
  - mongodb_list_collections input is empty string.
  - mongodb_schema expects comma-separated collection names.
  - mongodb_query expects a valid Query API call string (find/aggregate in JSON-like form), not plain English.
- **LLM settings:**
  - Set temperature=0 for determinism.
  - Keep prompts concise; pass only necessary query results to reduce context length.
- **Performance:**
  - Use top_k defaults and $limit early in pipelines.
  - Project only needed fields to reduce payload size.
- **Debugging:**
  - Stream graph state to observe tool decisions.
  - Use checkpoints to analyze, branch, or reset runs.