Mar 23, 2025

A Practical Deep Dive Into Knowledge for Agentic Systems

AI Agents Crash Course—Part 7 (with implementation).

 Avi Chawla, Akshay Pachaar

Introduction

So far in this AI Agent crash course series, we've built powerful agents that:

- Collaborate across tasks and roles.
- Execute asynchronously.
- Use callbacks, guardrails, and multimodal inputs.
- Work under human supervision and within hierarchical processes.
- and many many more things.

However, so far, our Agents have mostly worked with inputs and tools we provided at runtime, like a prompt, a blog URL, or a tool to take some real-world action.

What if you want your agents to:

- Access a company knowledge base?
- Query structured datasets like CSVs or JSON?
- Read and recall insights from PDFs, docs, or text files?
- Answer questions based on internal documentation or product specs.

Knowledge helps you do that and this is what we are discussing today.

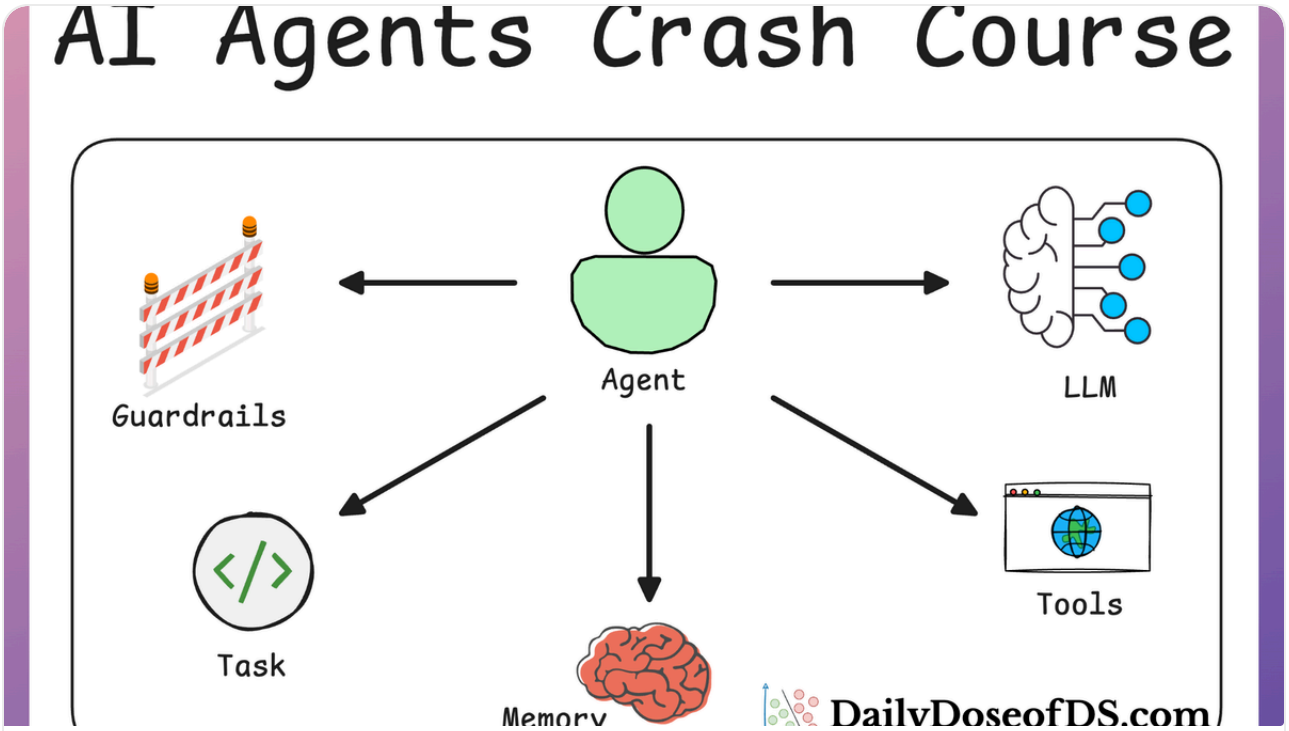
Let's dive in to learn more about it. Like always, everything will be supported with clear explanations to help you learn practically.

A quick note

In case you are new here...

Over the past six parts of this crash course, we have progressively built our understanding of Agentic Systems and multi-agent workflows with CrewAI.

- In Part 1, we covered the fundamentals of Agentic systems, understanding how AI agents can act autonomously to perform structured tasks.

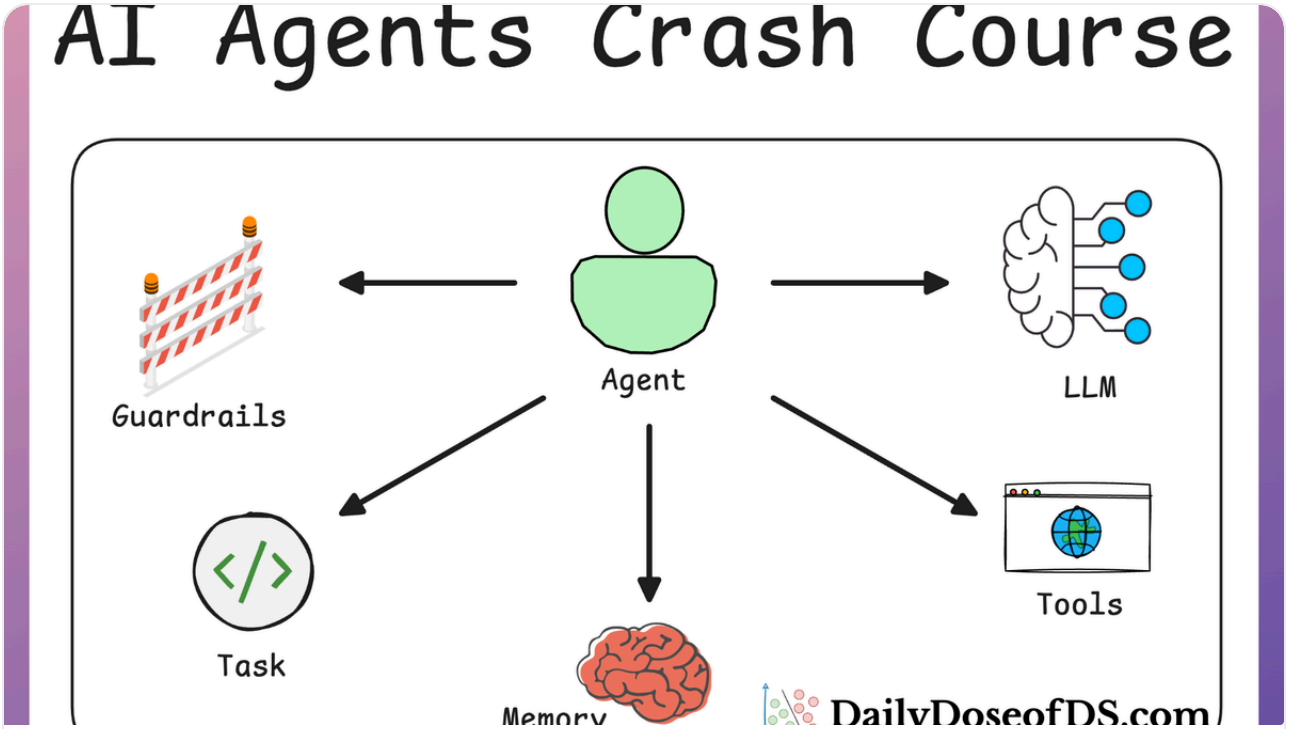


AI Agents Crash Course—Part 1 (With Implementation)

A deep dive into the fundamentals of Agentic systems, building blocks, and how to build them.

Daily Dose of Data Science • Avi Chawla

- In Part 2, we explored how to extend Agent capabilities by integrating custom tools, using structured tools, and building modular Crews to compartmentalize responsibilities.

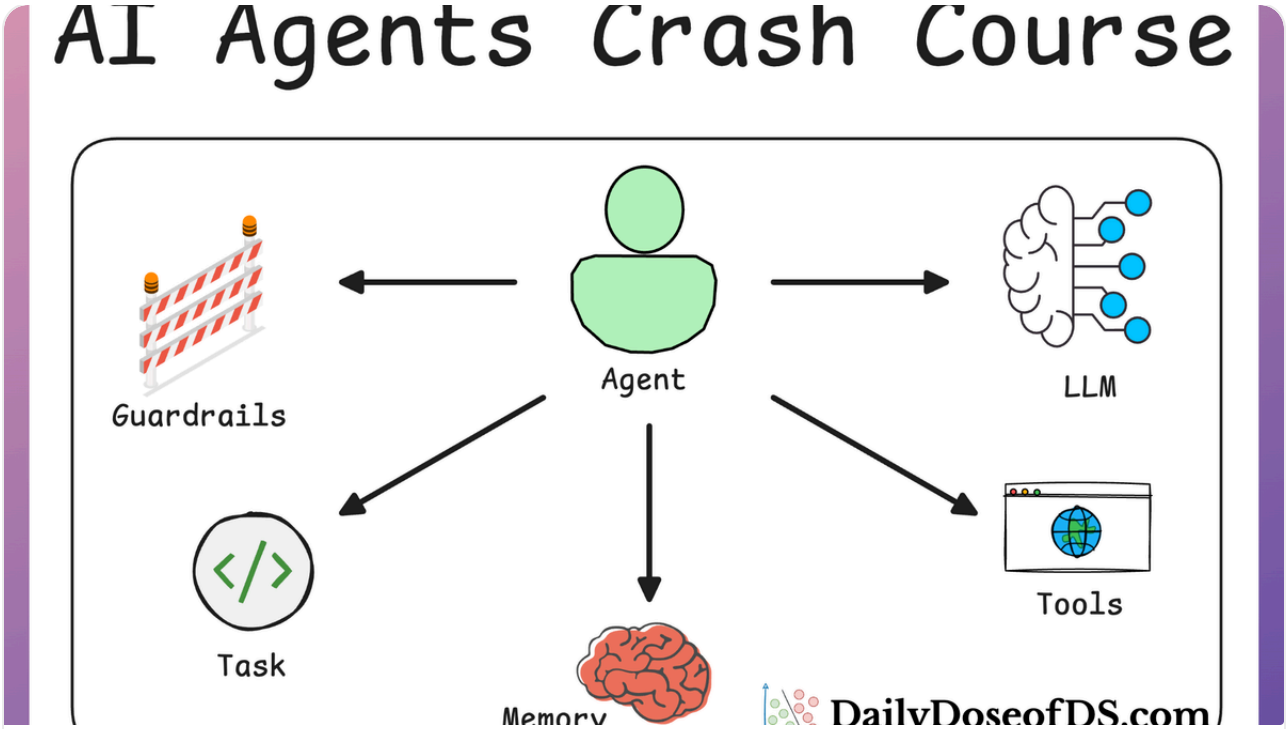


AI Agents Crash Course—Part 2 (With Implementation)

A deep dive into the fundamentals of Agentic systems, building blocks, and how to build them.

Daily Dose of Data Science • Avi Chawla

- In Part 3, we focused on Flows, learning about state management, flow control, and integrating a Crew into a Flow. As discussed last time, with Flows, you can create structured, event-driven workflows that seamlessly connect multiple tasks, manage state, and control the flow of execution in your AI applications.

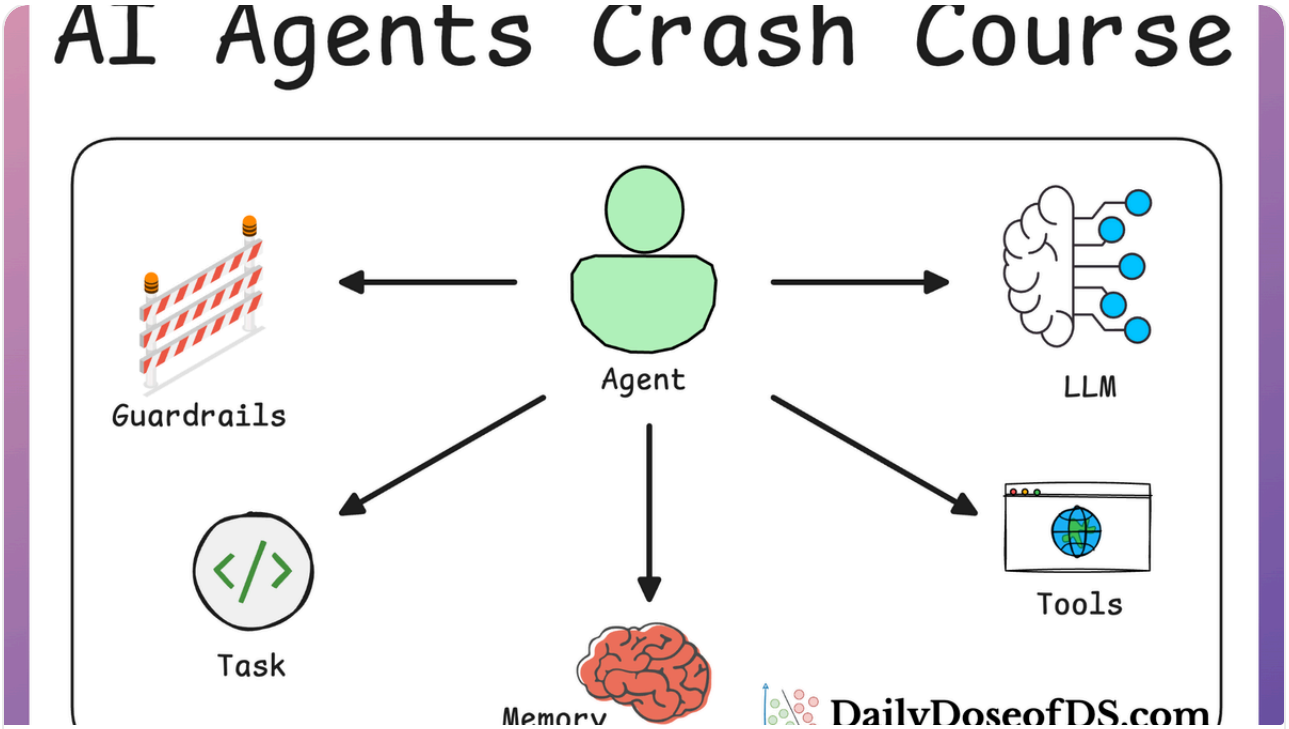


AI Agents Crash Course—Part 3 (With Implementation)

A deep dive into implementing Flows for building robust Agentic systems.

Daily Dose of Data Science • Avi Chawla

- In Part 4, we extended these concepts into real-world multi-agent, multi-crew Flow projects, demonstrating how to automate complex workflows such as content planning and book writing.



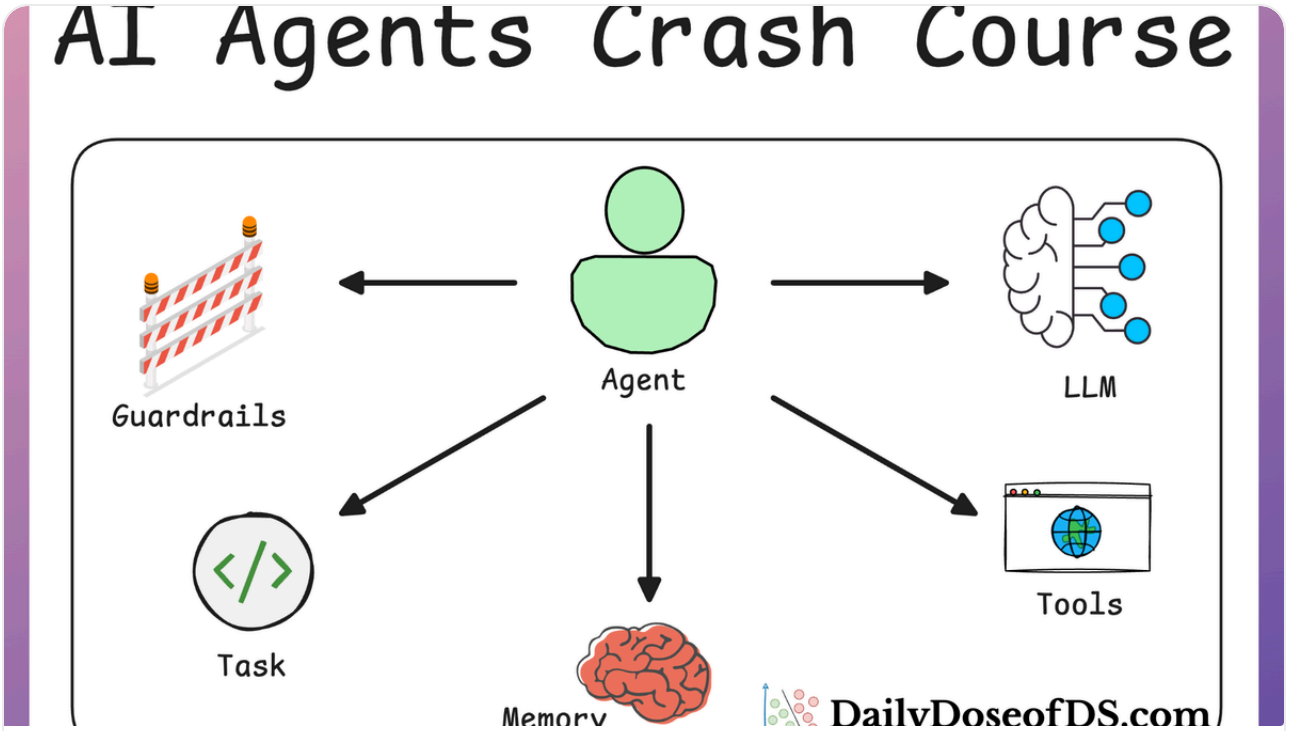
AI Agents Crash Course—Part 4 (With Implementation)

A deep dive into implementing Flows for building robust Agentic systems.

Daily Dose of Data Science • Avi Chawla

- In Part 5 and 6, we shall move into advanced techniques that make AI agents more robust, dynamic, and adaptable.
    - Guardrails → Enforcing constraints to ensure agents produce reliable and expected outputs.
    - Referencing other Tasks and their outputs → Allowing agents to dynamically use previous task results.
    - Executing tasks async → Running agent tasks concurrently to optimize performance.
    - Adding callbacks → Allowing post-processing or monitoring of task completions.
    - Introduce human-in-the-loop during execution → Introducing human-in-the-loop mechanisms for validation and control.
    - Hierarchical Agentic processes → Structuring agents into sub-agents and multi-level execution trees for more complex workflows.

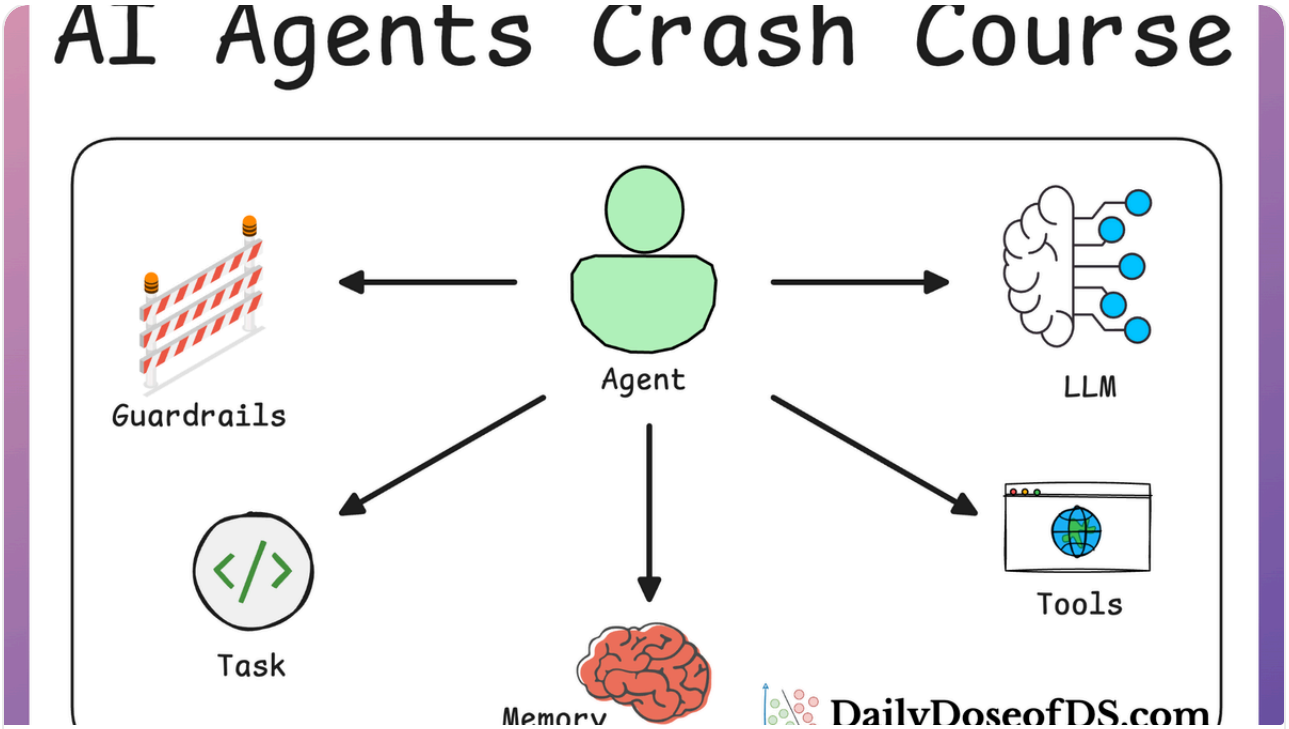
- Multimodal Agents → Extending CrewAI agents to handle text, images, audio, and beyond.
- and more.



AI Agents Crash Course—Part 5 (With Implementation)

A deep dive into advanced techniques to make robust Agentic systems.

Daily Dose of Data Science • Avi Chawla



AI Agents Crash Course—Part 6 (With Implementation)

A deep dive into advanced techniques to make robust Agentic systems.

Daily Dose of Data Science • Avi Chawla

Installation and setup

💡 Feel free to skip this part if you have followed these instructions before.

Throughout this crash course, we have been using CrewAI, an open-source framework that makes it seamless to orchestrate role-playing, set goals, integrate tools, bring any of the popular LLMs, etc., to build autonomous AI agents.

GitHub - crewAIInc/crewAI: Framework for orchestrating role-playing, autonomous AI agents. By fostering collaborative intelligence, CrewAI empowers agents to work together seamlessly, tackling complex tasks.

Framework for orchestrating role-playing, autonomous AI agents. By fostering collaborative intelligence, CrewAI empowers agents to work together seamlessly, tackling...

GitHub • crewAIInc

To highlight more, CrewAI is a standalone independent framework without any dependencies on Langchain or other agent frameworks.

Let's dive in!

To get started, install CrewAI as follows:

notebook.ipynb

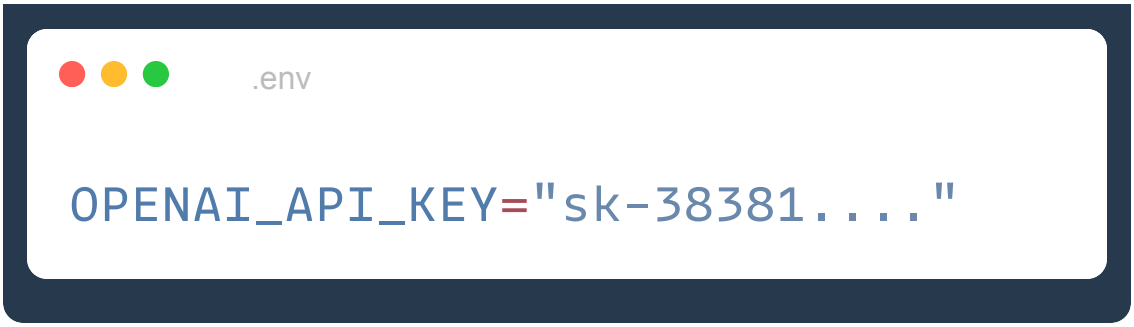
```
!pip install crewai
!pip install crewai-tools
```

Like the RAG crash course, we shall be using Ollama to serve LLMs locally. That said, CrewAI integrates with several LLM providers like:

- OpenAI
- Gemini
- Groq
- Azure
- Fireworks AI
- Cerebras
- SambaNova
- and many more.

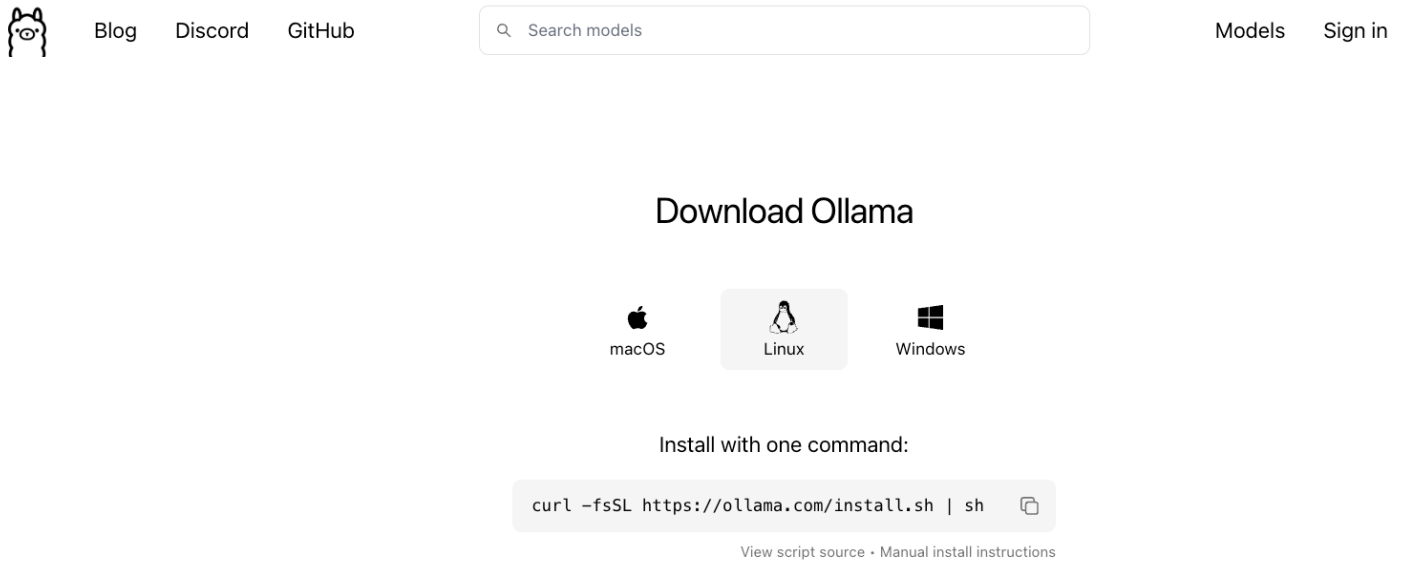
> 💡 If you have an OpenAI API key, we recommend using that since the outputs may not make sense at times with weak LLMs. If you don't have an API key, you can get some credits by creating a dummy account on OpenAI and use that instead. If not, you can continue reading and use Ollama instead but the outputs could be poor in that case.

To set up OpenAI, create a `.env` file in the current directory and specify your OpenAI API key as follows:

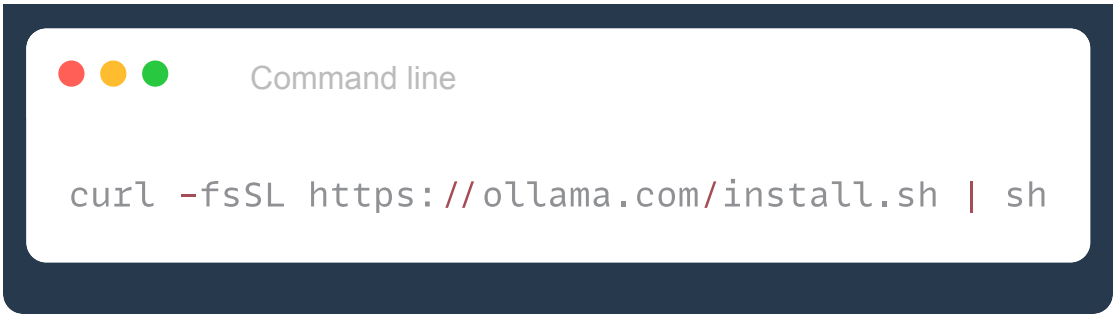


Also, here's a step-by-step guide on using Ollama:

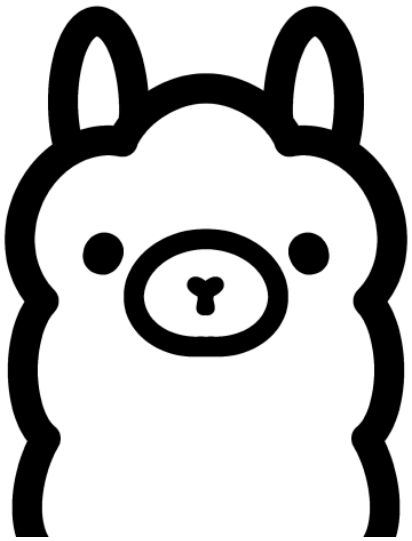
- Go to **Ollama.com**, select your operating system, and follow the instructions.



- If you are using Linux, you can run the following command:



- Ollama supports a bunch of models that are also listed in the model library:



library

Get up and running with large language models.

Blog Discord GitHub

Search models

Models Sign in Download

Models

Filter by name...

Most popular

llama3.2

Meta's Llama 3.2 goes small with 1B and 3B models.

tools 1b 3b

2.2M Pulls 63 Tags Updated 5 weeks ago

llama3.1

Llama 3.1 is a new state-of-the-art model from Meta available in 8B, 70B and 405B parameter sizes.

tools 8b 70b 405b

8M Pulls 93 Tags Updated 7 weeks ago

gemma2

Google Gemma 2 is a high-performing and efficient model available in

Once you've found the model you're looking for, run this command in your terminal:

Command line

```
# Replace 'llama3.2' with the desired model name

ollama run llama3.2
```

The above command will download the model locally, so give it some time to complete. But once it's done, you'll have Llama 3.2 3B running locally, as shown below, which depicts Microsoft's Phi-3 served locally through Ollama:

That said, for our demo, we would be running Llama 3.2 1B model instead since it's smaller and will not take much memory:

Done!

Everything is set up now and we can move on to building robust agentic workflows.

You can download the code for this article below:

AI Agent crash course Part 7

AI Agent crash course Part 7.zip • 37 KB

Knowledge for Agents

In CrewAI, `knowledge_sources` allow you to augment your agents with external reference material. Think of it as giving your agent a library of context they can search and use while performing a task.

Supported formats include:

- Plaintext (`.txt`)
- PDFs
- Markdown (`.md`)

- CSV / Excel spreadsheets
- JSON files
- Custom APIs
- Raw strings

You can attach knowledge either at the agent level (agent-specific memory) or crew level (shared knowledge across agents).

One thing we want to clarify here is that this is not the same as tools.

While tools allow agents to take real-time actions—like performing a web search, reading a file, or scraping a site—knowledge sources are passive, meaning they provide embedded context that agents can retrieve and reason over during execution.

Let’s break this down further.

- Primary role:
  - Tools → Perform an action.
  - Knowledge → Provide reference context for reasoning.
- Invocation:
  - Tools → Explicitly called by the agent
  - Knowledge → Implicitly accessed via semantic retrieval
- Samples:
  - Tools → Web search, file reader, API calls
  - Knowledge → PDFs, CSVs, docs, preloaded text
- Example:
  - Tools → “Go fetch this from the web”
  - Knowledge → “Use what you already know to answer this question”
- Execution Time:
  - Tools → Happens live during the agent's run.

    - Knowledge → Preprocessed and stored prior to task execution.

In short:

- If you're telling the agent to do something, you use a tool.
- If you're giving the agent something to know, you use knowledge.

To exemplify, imagine you're designing an agent to onboard a new employee.

- You might attach a PDF of the employee handbook as a knowledge source—so the agent can answer questions like "What's our vacation policy?"
- Separately, you might give it access to a custom HR API as a tool—so it can request the new hire's ID badge or send a Slack message.

Understanding this distinction will help you design better agentic workflows by deciding which elements of the task are static reference vs dynamic operations.

Now that we've made this distinction clear, let's look at how to load and use knowledge sources in practice. We'll start with something simple: passing in a plain string.

Creating a string knowledge source

Let's start with the simplest kind of memory—a plain string.

This is useful when you want your agent to have access to a small amount of structured or unstructured information without having to reference an external file or database.

Let's walk through a basic example. Suppose we're building a support assistant that can answer questions about a company's product return policy. We want the agent to reference the policy from memory and respond to questions in natural language.

Here's how we can do it, step by step:

We define the return policy as a raw string and wrap it in a `StringKnowledgeSource`:

```
from crewai.knowledge.source.string_knowledge_source
 import StringKnowledgeSource


policy_text = """
Our return policy allows customers to return
any product within 30 days of purchase. Refunds
will be issued only if the item is unused and in original packaging.
Customers must provide proof of purchase when requesting a return.
"""

# Create a StringKnowledgeSource object
return_policy_knowledge = StringKnowledgeSource(content=policy_text)
```

Define the knowledge

We use an LLM like `gpt-4o` (you can use a local LLM through Ollama as well here):

```
from crewai import LLM

llm = LLM(model="gpt-4o")
```

Define the LLM

We now create a support agent who knows everything about the return policy.

Define the Agent

```
from crewai import Agent

returns_agent = Agent(
    role="Product Returns Assistant",
    goal="Answer customer questions about return policy accurately.",
    backstory="""You work in customer service and specialize
                 in returns, refunds, and policies.""",
    allow_delegation=False,
    verbose=True,
    llm=llm
)
```

We don't attach the knowledge source directly to the agent here—we'll do it at the crew level in a moment. This keeps the agent reusable.

Now, we give the agent a task: answer a user's question about returns.

Define the Task

```
from crewai import Task

returns_task = Task(
    description="Answer the customer question: {question}",
    expected_output="A concise and accurate answer.",
    agent=returns_agent
)
```

Note that `{question}` is an input placeholder. We'll provide that dynamically when we run the workflow.

Now we assemble the crew, attach our knowledge source at the crew level, and define the process.

Specify knowledge

```
from crewai import Crew, Process

crew = Crew(
    agents=[returns_agent],
    tasks=[returns_task],
    process=Process.sequential,
    knowledge_sources=[return_policy_knowledge],
    verbose=True
)
```

You can also attach knowledge at the agent level, but attaching it to the crew makes it available to all agents in the workflow.

Now we run the crew with a specific input question.

Kickoff

```
query = "Can I get a refund if I used the item once?"

result = crew.kickoff(inputs={"question": query})
```

The agent will semantically search the return policy knowledge and respond accordingly.

```
result = crew.kickoff(inputs={
    "question": "Can I get a refund if I used the item once?"
})

from pprint import pprint
pprint(result.raw)
```

✓ 2.4s

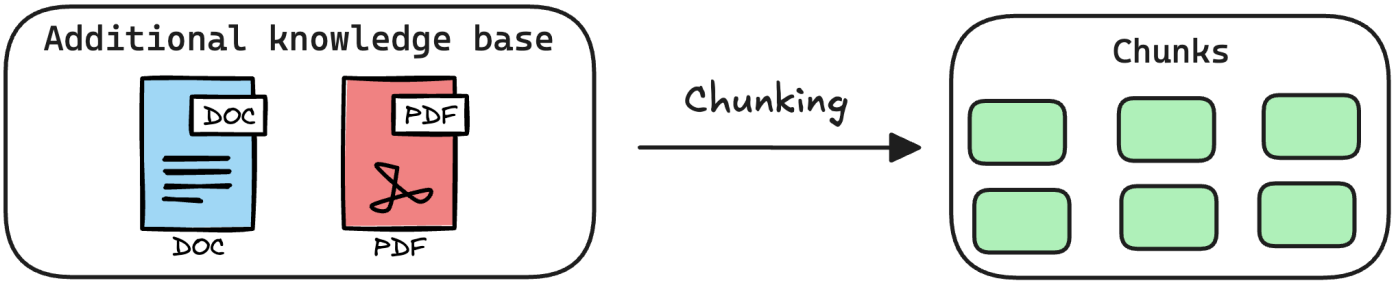
```
# Agent: Product Returns Assistant
## Task: Answer the following customer question about returns: Can I get a refund if I used the item onc


# Agent: Product Returns Assistant
## Final Answer:
According to our return policy, refunds are issued only for items that are unused and in original packag


('According to our return policy, refunds are issued only for items that are '
 'unused and in original packaging, accompanied by proof of purchase. Since '
 'you mentioned that the item was used, it would not qualify for a refund '
 'under our current policy.')
```

The reason this is useful is because now that we have a string knowledge source (which could be any string for that matter), we can include the typical RAG setup here, wherein, we retrieve the context from a knowledge source based on the user query, embed it, and query the vector database to find the most relevant context related to the query.

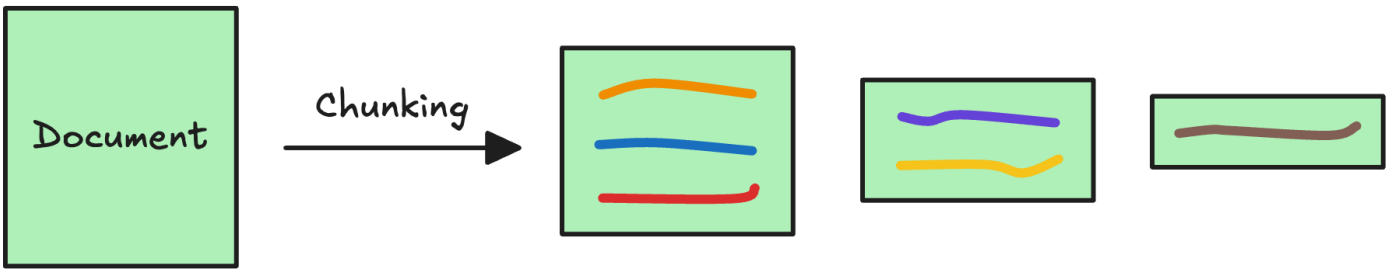
Once that's done, this retrieved context can become our external knowledge base which can be converted into a `StringKnowledgeSource` class, and passed to Agents for further processing.

Chunking

Typically, when we start with some external knowledge that we want to augment the LLM with, the first step is to break down this additional knowledge into chunks before embedding and storing it in the vector database.

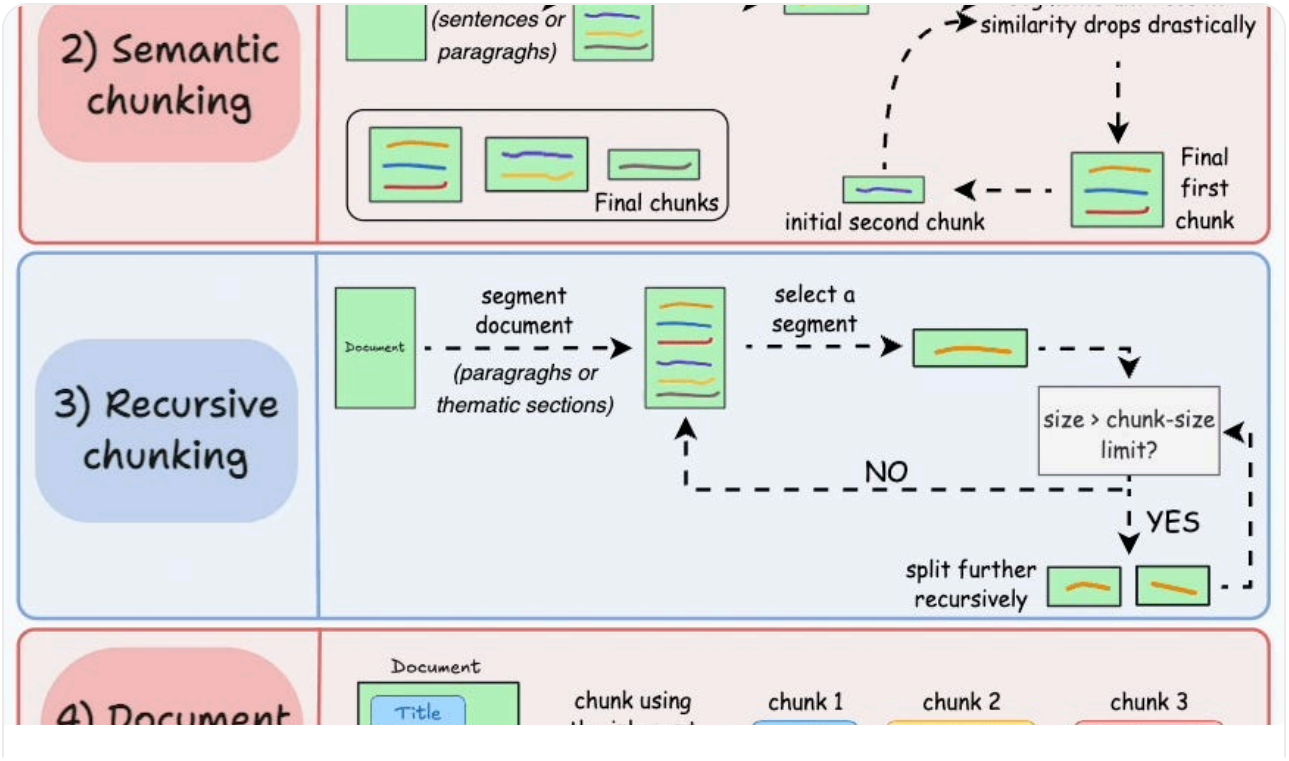


We do this because the additional document(s) can be pretty large. Thus, it is important to ensure that the text fits the input size of the embedding model.



Moreover, if we don't chunk, the entire document will have a single embedding, which won't be of any practical use to retrieve relevant context.

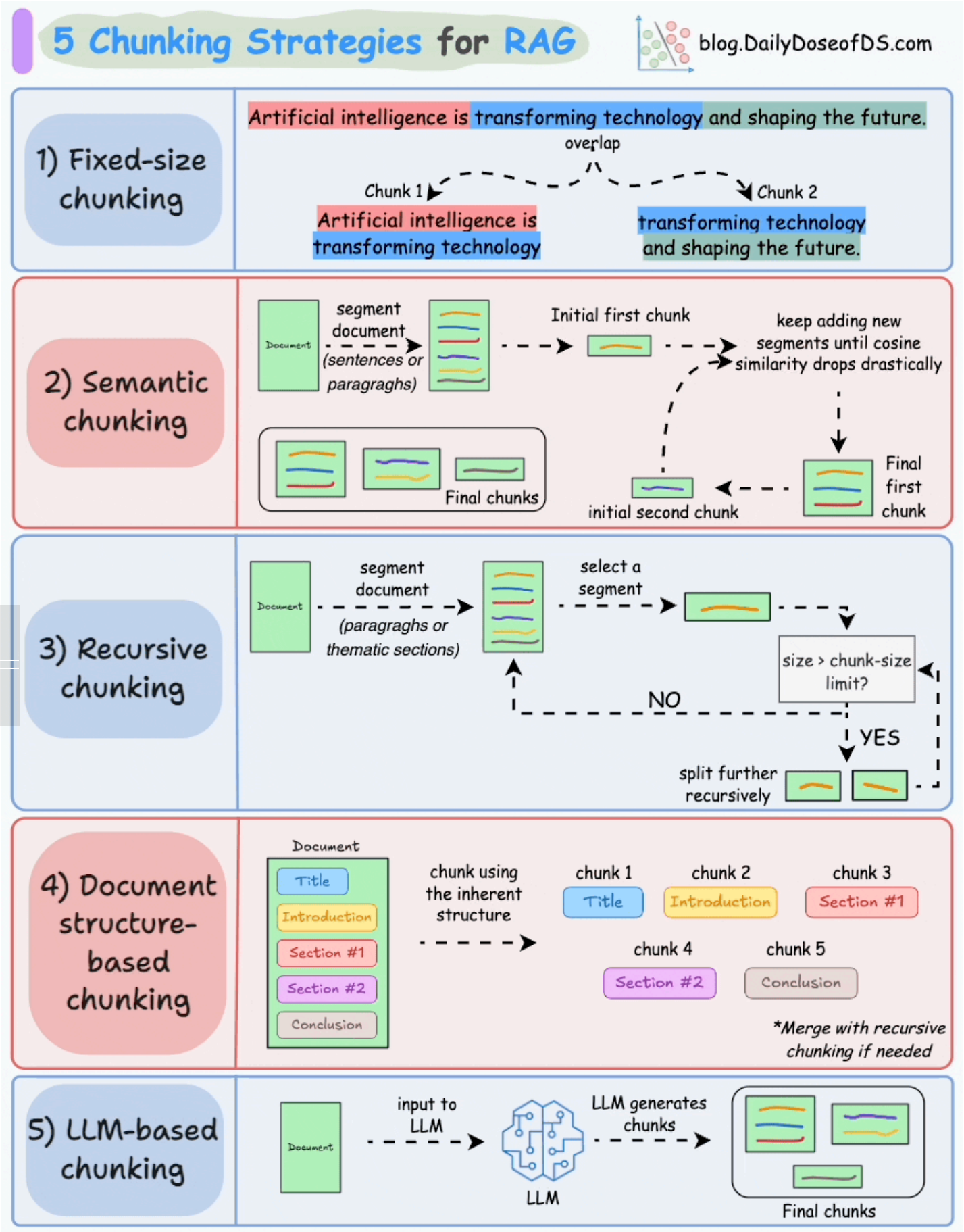
On a side note, we covered chunking strategies recently in the newsletter here:



5 Chunking Strategies For RAG

...explained in a single frame.

Daily Dose of Data Science • Avi Chawla

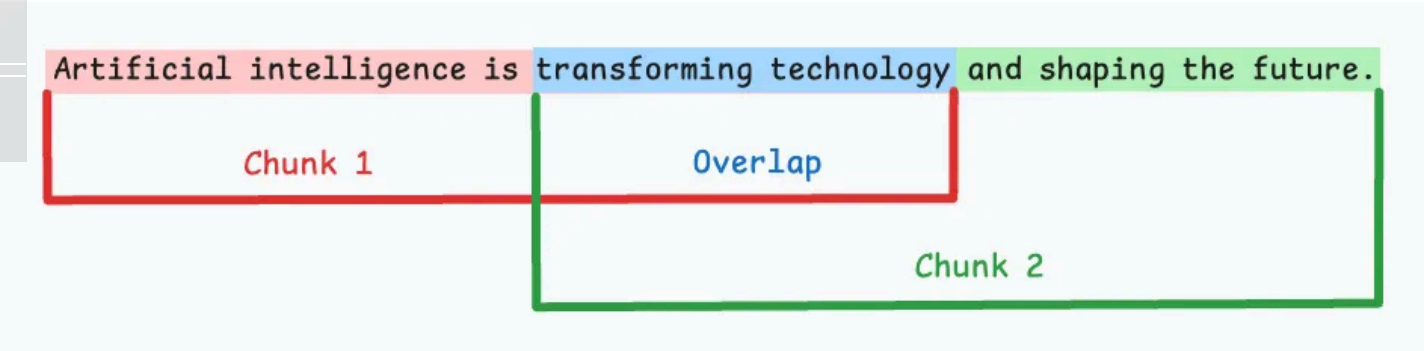


Coming back to the topic...

In your case, you can declare the chunking parameters when declaring the knowledge sources as follows:

Chunking

```
from crewai.knowledge.source.string_knowledge_source
 import StringKnowledgeSource

source = StringKnowledgeSource(
    content="Your content here",

    chunk_size=4000,      # Maximum size of each chunk (default: 4000)

    chunk_overlap=200     # Overlap between chunks (default: 200)
)
```



Since a direct split can disrupt the semantic flow, it is recommended to maintain some overlap between two consecutive chunks (the blue part above).

Knowledge access level

You have flexibility over who can access what knowledge.

As discussed above, this is controlled using the `knowledge_sources` parameter, which you can assign at:

- Agent level → The knowledge is only visible to that specific agent.

- Crew level → The knowledge is shared across all agents in the crew.

If you think about it, this design mirrors how information flows in real-world teams. Sometimes, only certain members have access to confidential documents. Other times, everyone in a team might share access to a central knowledge base.

Let's walk through both approaches using a practical example.

#1) Agent-level access

Suppose we're building a product assistant who knows details about an upcoming launch. We want only this agent to access the product summary—not others.

First, we create the knowledge source:

Define source

```
from crewai.knowledge.source.string_knowledge_source
 import StringKnowledgeSource

product_info = """Our new product, Beacon AI, launches in Q3. It's a
                  platform for automating enterprise workflows using
                  AI agents. Key selling points: secure deployment,
                  agent chaining, and seamless API integration."""

product_knowledge = StringKnowledgeSource(content=product_info)
```

Moving on, we assign it to a specific agent:

```python
from crewai import Agent, LLM

llm = LLM(model="gpt-4o")

product_analyst = Agent(
    role="Product Launch Specialist",
    goal="Answer internal questions about our upcoming product.",
    backstory="""You work on the go-to-market team and
                 know the product details inside out.""",
    knowledge_sources=[product_knowledge],
    llm=llm,
    verbose=True
)
```

Agent-level knowledge

Next, we create a task and run a Crew:

Run Crew

```python
from crewai import Task, Crew, Process

task = Task(
    description="What is Beacon AI and when does it launch?",
    expected_output="A short summary for internal use.",
    agent=product_analyst
)

crew = Crew(
    agents=[product_analyst],
    tasks=[task],
    process=Process.sequential,
    verbose=True
)

result = crew.kickoff()
print(result)
```

In this case, only the `product_analyst` agent can access that string-based knowledge.

#2) Crew-level access

Now let's say you want multiple agents (marketing, support, sales) to share access to the same product info. Instead of assigning it to each one individually, you attach it at the crew level.

We use the same knowledge source as before:

Define source

```
from crewai.knowledge.source.string_knowledge_source
 import StringKnowledgeSource

product_info = """Our new product, Beacon AI, launches in Q3. It's a
                  platform for automating enterprise workflows using
                  AI agents. Key selling points: secure deployment,
                  agent chaining, and seamless API integration."""

product_knowledge = StringKnowledgeSource(content=product_info)
```

Next, we define two agents (without individual knowledge access):

Agents

```
marketing_agent = Agent(
    role="Marketing Strategist",
    goal="Craft messaging around our product launch.",
    backstory="You translate product features into compelling stories.",
    llm=llm,
    verbose=True
)

support_agent = Agent(
    role="Customer Support Specialist",
    goal="Prepare FAQs and answers for product-related customer inquiries.",
    backstory="You make sure support teams are ready before launch.",
    llm=llm,
    verbose=True
)
```

Next, we define the tasks for both agents:

Tasks

```
marketing_task = Task(
    description="Create a short headline summarizing what Beacon AI does.",
    expected_output="A 1-line marketing headline.",
    agent=marketing_agent
)

support_task = Task(
    description="Draft an FAQ entry answering: What is Beacon AI?",
    expected_output="A short FAQ-style answer for users.",
    agent=support_agent
)
```

Finally, we attach knowledge at the crew level:

```
crew = Crew(
    agents=[marketing_agent, support_agent],
    tasks=[marketing_task, support_task],
    knowledge_sources=[product_knowledge],
    process=Process.sequential,
    verbose=True
)

result = crew.kickoff()
print(result)
```

Crew-level access

Both agents will now be able to retrieve and reason over the shared product summary—even though the knowledge wasn't attached individually.

In general:

- If the knowledge is agent-specific, pass it via `Agent(knowledge_sources=[...])`.
- If the knowledge is relevant to all, pass it via `Crew(knowledge_sources=[...])`.

This lets you control access without duplicating configuration—just like in real team environments.

Using real files as knowledge

Let's now go beyond simple string memory and explore how to work with different types of files using CrewAI's knowledge sources.

Imagine you're building a multi-agent system for a startup's internal operations. These agents should be able to:

- Answer questions about HR policy (text file),
- Summarize meeting notes (PDF),
- Analyze user feedback (CSV),
- Interpret financial data (Excel),
- And handle structured company info (JSON).

Each of these formats is supported out of the box by CrewAI. All you have to do is load them as knowledge sources and plug them into your agents or crews.

Before we start, create a folder named `./knowledge/` and place the following sample files inside:

- `hr_policy.txt`
- `meeting_notes.pdf`
- `feedback.csv`
- `company_info.json`

These are already available in the code shared above.

Text source

Let's start with the most straightforward example: a `.txt` file that contains your HR policy. This file contains leave policies, remote work guidelines, or reimbursement rules—all of which the agent will have access to.

knowledge > hr_policy.txt

```
Company HR Policy - Leave and Attendance

1. All full-time employees are entitled to 21 days of paid leave annually.
2. New employees are eligible for leave after completing their first 30 days.
3. Sick leave must be informed on the same day before 10 AM.
4. Remote work is allowed up to 3 days a week with manager approval.
5. Any leave exceeding 3 consecutive days must be supported by documentation.
```

You want an agent who can read this document and answer internal questions about employee leaves, sick policies, or remote work.

First, load the HR policy file as a knowledge source:

```
from crewai.knowledge.source.text_file_knowledge_source
 import TextFileKnowledgeSource

# This file contains the company's HR leave and attendance policy.
text_source = TextFileKnowledgeSource(
    file_paths=["hr_policy.txt"]
)
```

Location of .txt file in /knowledge folder

Next, create an agent that knows everything about your HR rules:

```
from crewai import Agent, LLM

llm = LLM(model="gpt-4o")

hr_agent = Agent(
    role="HR Policy Assistant",
    goal="Help employees understand HR guidelines.",
    backstory="""You are an experienced HR support assistant,
                 trained on company policy documents.""",
    knowledge_sources=[text_source],
    llm=llm
)
```

Specify knowledge

This agent is now equipped with the contents of `hr_policy.txt`.

Moving on, define a task for the agent:

Specify task

```
from crewai import Task

task = Task(
    description="What is the sick leave protocol for employees?",
    expected_output="A brief explanation of the sick leave process.",
    agent=hr_agent
)
```

Finally, assemble the crew and run the task:

Run Crew

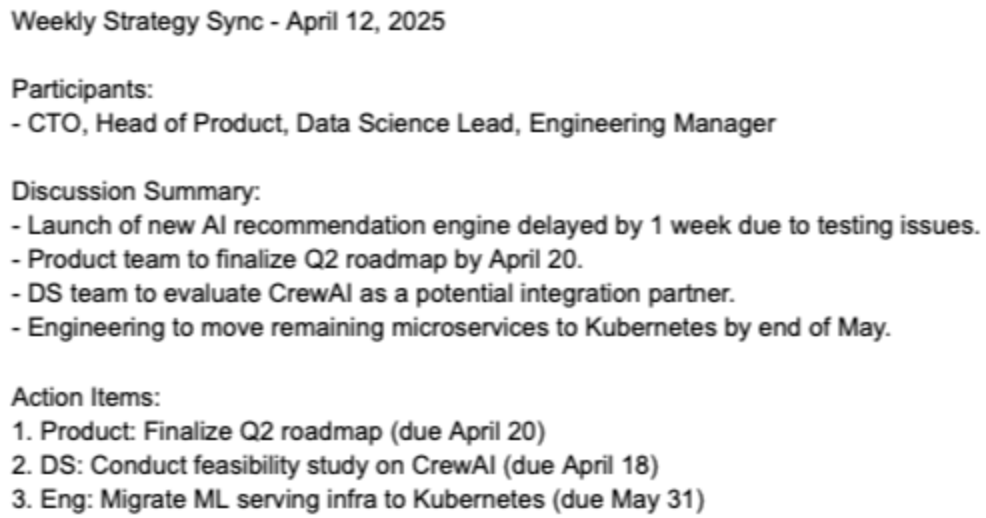
```
from crewai import Crew, Process

crew = Crew(
    agents=[hr_agent],
    tasks=[task],
    process=Process.sequential,
    verbose=True
)

result = crew.kickoff()
print(result)
```

We get the following response which is coherent with the text source:

```
Overriding of current TracerProvider is not allowed
# Agent: HR Policy Assistant
## Task: What is the leave policy for new employees?


# Agent: HR Policy Assistant
## Final Answer:
The leave policy for new employees is as follows: All full-time employees are enti


('The leave policy for new employees is as follows: All full-time employees '
 'are entitled to 21 days of paid leave annually. New employees become '
 'eligible for leave after completing their first 30 days of employment. If an '
 'employee needs to take sick leave, they must inform the company by 10 AM on '
 'the same day. Additionally, remote work is permitted for up to 3 days a week '
 'with the approval of their manager. For any leave that exceeds 3 consecutive '
 'days, employees are required to provide appropriate documentation.')
```

PDF source

Now let's say you have weekly PDF meeting notes with summaries and action items. You want your agent to read this file and extract the next steps.



Weekly Strategy Sync - April 12, 2025

Participants:
- CTO, Head of Product, Data Science Lead, Engineering Manager

Discussion Summary:
- Launch of new AI recommendation engine delayed by 1 week due to testing issues.
- Product team to finalize Q2 roadmap by April 20.
- DS team to evaluate CrewAI as a potential integration partner.
- Engineering to move remaining microservices to Kubernetes by end of May.

Action Items:
1. Product: Finalize Q2 roadmap (due April 20)
2. DS: Conduct feasibility study on CrewAI (due April 18)
3. Eng: Migrate ML serving infra to Kubernetes (due May 31)

First, load the `.pdf` meeting file:

```python
from crewai.knowledge.source.pdf_knowledge_source
 import PDFKnowledgeSource

pdf_source = PDFKnowledgeSource(
    file_paths=["meeting_notes.pdf"]
)
```

Location of .pdf file in /knowledge folder

Next, create a summarization agent:

```python
meeting_summarizer = Agent(
    role="Meeting Summary Specialist",
    goal="Extract and summarize action items from past meetings.",
    backstory="""You specialize in distilling meetings
                 into concise action points.""",
    knowledge_sources=[pdf_source],
    llm=llm
)
```

Pass knowledge

Moving on, define the task to summarize action items:

Create Task

```python
task = Task(
    description="List the action items from last week's meeting.",
    expected_output="A bullet list of 3-5 action items.",
    agent=meeting_summarizer
)
```

Finally, create and run the crew:

Run Crew

```
crew = Crew(
    agents=[meeting_summarizer],
    tasks=[task],
    process=Process.sequential,
    verbose=True
)

result = crew.kickoff()
print(result)
```

This produces the output shown below, which is aligned with the PDF input.

```
Overriding of current TracerProvider is not allowed
# Agent: Meeting Note Summarizer
## Task: Summarize the key action items from last week's meeting.


# Agent: Meeting Note Summarizer
## Final Answer:
- Product: Finalize Q2 roadmap (due April 20)
- DS: Conduct feasibility study on CrewAI (due April 18)
- Eng: Migrate ML serving infra to Kubernetes (due May 31)


('- Product: Finalize Q2 roadmap (due April 20)\n'
 '- DS: Conduct feasibility study on CrewAI (due April 18)\n'
 '- Eng: Migrate ML serving infra to Kubernetes (due May 31)')
```

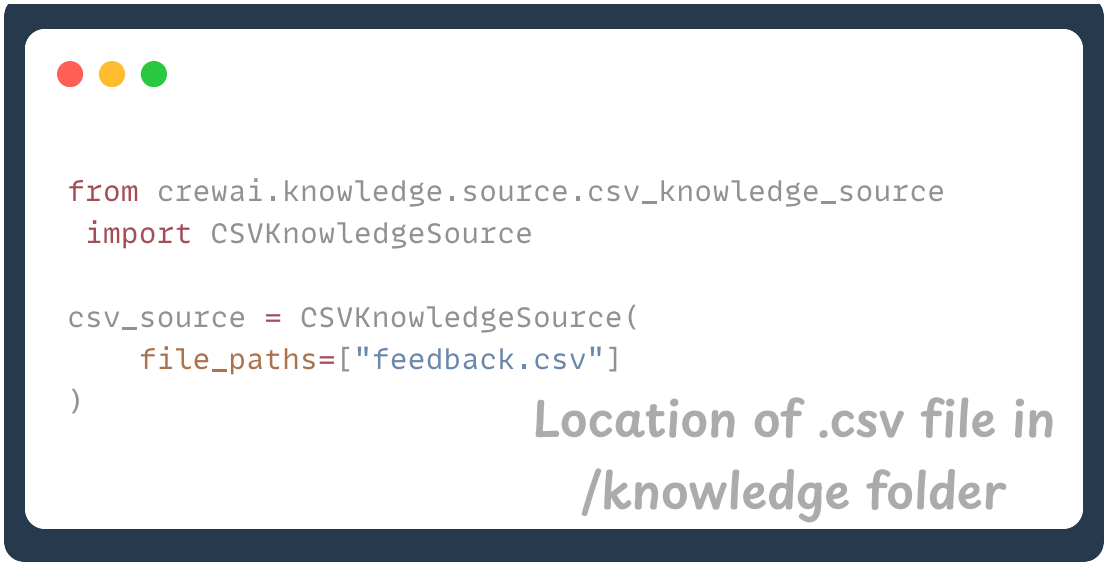
CSV source

Imagine you have a `.csv` file where users have submitted product feedback. Now, you want an agent to read that CSV and highlight recurring pain points.

knowledge > feedback.csv

```
id, user_id, comment, rating
1, 101, The UI feels slow on mobile, 3
2, 102, I love the new dashboard layout!, 5
3, 103, Notifications are too frequent and not customizable, 2
4, 104, Customer support was really helpful, 4
5, 105, It's hard to find the export option in reports, 3
```

Just like earlier, load the feedback CSV file:



Next, build a feedback analysis agent:

```
feedback_analyst = Agent(
    role="Feedback Analyst",
    goal="Analyze customer sentiment and complaints.",
    backstory="""You process raw user feedback and extract
                 recurring themes for the product team.""",
    knowledge_sources=[csv_source],
    llm=llm
)
```

Pass knowledge

Moving on, define the task to identify the top recurring user complaints:

Create Task

```
task = Task(
    description="What are the top 2 common issues users are facing?",
    expected_output="A summary of top 2 user pain points.",
    agent=feedback_analyst
)
```

Finally, create the crew and kickoff the task:

Run Crew

```
crew = Crew(
    agents=[feedback_analyst],
    tasks=[task],
    process=Process.sequential,
    verbose=True
)

result = crew.kickoff()
print(result)
```

This produces the output shown below, which is aligned with the CSV input.

```
Overriding of current TracerProvider is not allowed
# Agent: User Feedback Analyst
## Task: What are the three most common complaints users had last month?


# Agent: User Feedback Analyst
## Final Answer:
The three most common complaints users had last month are:

1. The UI feels slow on mobile.
2. Notifications are too frequent and not customizable.
3. It's hard to find the export option in reports.


('The three most common complaints users had last month are: \n'
 '\n'
 '1. The UI feels slow on mobile.\n'
 '2. Notifications are too frequent and not customizable.\n'
 '3. It's hard to find the export option in reports.')
```

JSON source

Finally, let's say your company logs org-level data in a structured JSON format—like departments, team sizes, and reporting lines.

knowledge > {} company_info.json > [ ] departments

```json
{
  "company_name": "TechNova Inc.",
  "departments": [
    {
      "name": "Product",
      "head": "Alice",
      "teams": [
        {"name": "UX Team", "members": 5},
        {"name": "Analytics Team", "members": 3}
      ]
    },
    {
      "name": "Engineering",
      "head": "Bob",
      "teams": [
        {"name": "Backend", "members": 6},
        {"name": "DevOps", "members": 4},
        {"name": "ML Engineering", "members": 5}
      ]
    }
  ],
  "employee_count": 62
}
```

First, load the structured company data:

```
from crewai.knowledge.source.json_knowledge_source
 import JSONKnowledgeSource

json_source = JSONKnowledgeSource(
    file_paths=["company_info.json"]
)
```

Location of JSON file in /knowledge folder

Next, build a company information expert:

```
org_analyst = Agent(
    role="Org Structure Specialist",
    goal="Answer questions about company departments and teams.",
    backstory="""You help HR and leadership make decisions based
                 on team sizes and reporting structure.""",
    knowledge_sources=[json_source],
    llm=llm
)
```

Pass knowledge

Moving on, define a task to list all teams in the engineering department:

Create Task

```
task = Task(
    description="List the teams under the Engineering department.",
    expected_output="A list of team names under Engineering.",
    agent=org_analyst
)
```

Finally, create the crew and launch the workflow:

Run Crew

```
crew = Crew(
    agents=[org_analyst],
    tasks=[task],
    process=Process.sequential,
    verbose=True
)

result = crew.kickoff()
print(result)
```

We get the following response which is coherent with the JSON source:

```
Overriding of current TracerProvider is not allowed
# Agent: Company Info Specialist
## Task: How many teams are working on the product and what are their names?


# Agent: Company Info Specialist
## Final Answer:
There are two teams working on the product within TechNova Inc.'s Product department. The teams are:
1. UX Team with 5 members.
2. Analytics Team with 3 members.


There are two teams working on the product within TechNova Inc.'s Product department. The teams are:
1. UX Team with 5 members.
2. Analytics Team with 3 members.
```

Each agent in these examples is powered by a different knowledge source:

- TXT for HR policy
- PDF for internal summaries
- CSV for customer insights

- JSON for company metadata

Next, let's explore embedding customization, memory cleanup, and knowledge source chaining for advanced workflows. Let me know when you want to dive into that.

Create a custom knowledge source

Until now, we've worked with knowledge that lives in files: PDFs, JSON, Excel, etc. But what if the data you need comes from a live API, a database, or an internal system?

To define your own knowledge sources by extending the `BaseKnowledgeSource` class and implementing the logic to fetch, process, and store relevant data.

Let’s walk through an example where we build a Weather Knowledge Source that fetches real-time weather summaries for a given city.

We want to build an AI agent that:

- Fetches the latest weather for a city.
- Summarizes the key information like temperature, condition, and wind.
- Answers natural language questions about the current forecast.

We’ll build:

- A custom knowledge source that fetches from the Open-Meteo API.
- An agent who reads from that source.
- A task that prompts the agent to answer a question.
- A crew that connects it all.

To create a custom knowledge source in CrewAI, you need to subclass `BaseKnowledgeSource` and override a few key methods that define how your data is fetched, processed, and stored.

Let's walk through the structure, line by line, using our real-world example: building a live weather knowledge source.

Imports

```
from crewai.knowledge.source.base_knowledge_source
 import BaseKnowledgeSource

from typing import Dict, Any
from pydantic import Field
import requests
```

- `BaseKnowledgeSource`: This is the abstract base class CrewAI uses for all knowledge sources. You'll extend it to define your own logic.
- `Field`: From Pydantic, used to describe configurable fields for your knowledge source.
- `requests`: To fetch data from an API (in our case, the Open-Meteo weather API).

Next, we define a new class `WeatherKnowledgeSource`, which inherits from `BaseKnowledgeSource`. This makes it compatible with the CrewAI framework.

Also, since every knowledge source can have parameters, we take in the `city` so that the agent can fetch data for a specific location. You can also define other parameters like `units`, `language`, or `API token` depending on your use case.

Define knowledge class

```
class WeatherKnowledgeSource(BaseKnowledgeSource):
    city: str = Field(description="City for which weather should be fetched")
```

Moving on, we implement `load_content()` method. This method is required. It defines how to fetch the content you want the agent to learn from.

Define knowledge class

```
class WeatherKnowledgeSource(BaseKnowledgeSource):
    city: str = Field(description="City for which weather should be fetched")

    def load_content(self) -> Dict[Any, str]:
        print(f"Fetching weather for {self.city}...")

        # Hardcoded coordinates for demo purposes (San Francisco)
        endpoint = "https://api.open-meteo.com/v1/forecast"
        params = {
                "latitude": 37.77,
                "longitude": -122.42,
                "current_weather": True
            }

        response = requests.get(endpoint, params=params)
        response.raise_for_status()
```

- We make a GET request to the Open-Meteo API.
- In this example, we hardcode coordinates for San Francisco. In production, you might use a geocoding API to dynamically convert the city name to coordinates.
- If the request fails, we raise a clean error message.

Once data is fetched, it must be converted into plain text that can be embedded.

Define knowledge class

```python
class WeatherKnowledgeSource(BaseKnowledgeSource):
    city: str = Field(description="City for which weather should be fetched")

    def load_content(self) -> Dict[Any, str]:
        print(f"Fetching weather for {self.city}...")

        # Hardcoded coordinates for demo purposes (San Francisco)
        endpoint = "https://api.open-meteo.com/v1/forecast"
        params = {
                "latitude": 37.77,
                "longitude": -122.42,
                "current_weather": True
            }

        response = requests.get(endpoint, params=params)
        response.raise_for_status()
        weather_data = response.json().get("current_weather", {})
        formatted = self.format_weather(weather_data)
        return {self.city: formatted}
```

We extract only the `current_weather` section, format it, and return it as a dictionary with the city name as the key. This is the expected structure: `{source_id: raw_text}`.

Next up, we have a helper function `format_weather()` to convert the JSON response into readable text:

Define knowledge class

```python
class WeatherKnowledgeSource(BaseKnowledgeSource):
    city: str = Field(description="City for which weather should be fetched")

    def load_content(self) -> Dict[Any, str]:
        ...

    def format_weather(self, data: dict) -> str:
        if not data:
            return "No weather data available."

        return (
            f"Current weather in {self.city}:\n"
            f"- Temperature: {data.get('temperature')}°C\n"
            f"- Wind Speed: {data.get('windspeed')} km/h\n"
            f"- Weather Code: {data.get('weathercode')}\n"
            f"- Time: {data.get('time')}"
        )
```

This method:

- Checks if there's any data.
- Extracts values like temperature, wind speed, and time.
- Returns it as a neatly formatted string that can later be embedded into vector space.

Finally, we implement the `add()` method:

Define knowledge class

```
class WeatherKnowledgeSource(BaseKnowledgeSource):
    city: str = Field(description="City for which weather should be fetched")

    def load_content(self) -> Dict[Any, str]:
        ...

    def format_weather(self, data: dict) -> str:
        ...

    def add(self) -> None:
        content = self.load_content()
        for _, text in content.items():
            chunks = self._chunk_text(text)
            self.chunks.extend(chunks)

        self._save_documents()
```

This is where you:

1. Load the content.
2. Chunk it using CrewAI's built-in `_chunk_text()` method (inherited from `BaseKnowledgeSource`).
3. Save those chunks to be used in semantic search during execution.

You don't need to manually embed anything—CrewAI takes care of that.

Now we create an agent who knows how to answer questions based on this weather data.

```
from crewai import Agent, LLM

weather_knowledge = WeatherKnowledgeSource(city="San Francisco")

weather_agent = Agent(
    role="Weather Reporter",
    goal="Answer questions about the current weather forecast.",
    backstory="""You are a friendly meteorologist who
                 provides real-time weather updates.""",
    knowledge_sources=[weather_knowledge],
    llm=LLM(model="gpt-4o", temperature=0.0),
    verbose=True
)
```

Custom knowledge source

Moving on, we define the Task and the Crew:

```
from crewai import Task, Crew, Process

task = Task(
    description="What is the current temperature and wind speed in SF?",
    expected_output="A concise weather summary for San Francisco.",
    agent=weather_agent
)

crew = Crew(
    agents=[weather_agent],
    tasks=[task],
    process=Process.sequential,
    verbose=True
)
```

Task and Crew

Finally, we run the Crew:

Run Crew

```
result = crew.kickoff()

print(result)
```

This example illustrates a few important takeaways about custom knowledge sources:

- `BaseKnowledgeSource`: Parent class you must extend to build custom loaders.
- `load_content()`: Required method that returns a dictionary of text content to embed.
- `add()`: Method to handle preprocessing (e.g., chunking and saving).
- You can fetch from any source: APIs, webhooks, and even internal services. Once built, your knowledge source is just like any other—add it to agents or crews.

Embedding customization

Let's now explore a more advanced (but powerful) feature of CrewAI knowledge sources—embedding customization.

By default, when you attach a knowledge source to a Crew or an Agent, CrewAI will embed the data using the same provider as your LLM (e.g., OpenAI or Ollama). But in many scenarios, you may want more control.

You might want to:

- Use a different provider for embeddings than the one used for generation.

- Use local embeddings (e.g., with Ollama) for better privacy.
- Optimize the embedding model for shorter latency, larger input sizes, or domain-specific encoding.

Let's walk through a real-world example using Ollama as the embedding provider while maintaining complete control over how and where the knowledge is stored.

Suppose you want to build a local assistant that helps new employees with common onboarding questions. All the FAQs are stored in a string or a file, but you want everything to run offline using Ollama, including the embedding model.

First, set up the local knowledge base.

Define knowledge source

```
from crewai.knowledge.source.string_knowledge_source
 import StringKnowledgeSource

# Internal onboarding FAQ
faq_content = """
- You can access your email via portal.company.com using your employee credentials.
- The standard work hours are from 9am to 6pm, Monday to Friday.
- All reimbursement requests must be submitted by the 5th of the following month.
- For any IT-related issues, contact support@company.com.
"""

# Create a string knowledge source
faq_knowledge = StringKnowledgeSource(content=faq_content)
```

Next, we define an embedding model served through Ollama for embedding the knowledge.

To do that, first pull the `nomic-embed-text` from Ollama as follows:

Command line

```
ollama pull nomic-embed-text
```

Next, define the embedding model config served through Ollama:

Define Ollama embedder

```
from crewai import LLM

# Configure Ollama for embeddings
ollama_embedder = {
    "provider": "ollama",
    "config": {
        "model": "nomic-embed-text",
        "api_url": "http://localhost:11434"
    }
}
```

Moving on, we'll create an HR assistant that answers onboarding-related questions:

Define Agent

```
from crewai import Agent

hr_faq_agent = Agent(
    role="HR Assistant",
    goal="Answer onboarding-related questions for new hires.",
    backstory="""You are a helpful assistant who knows everything
                 about internal policies and onboarding processes.""",
    llm=ollama_llm,
    allow_delegation=False,
    verbose=True
)
```

The agent will be given a user question (e.g., about work hours or reimbursements) and will use the knowledge to answer. So its task is defined below:

Define Task

```
from crewai import Task

task = Task(
    description="Answer this onboarding question: {question}",
    expected_output="An accurate answer based on HR documentation.",
    agent=hr_faq_agent
)
```

Now we wire everything up with a custom embedder and the knowledge source:

Run Crew

```
from crewai import Crew, Process

crew = Crew(
    agents=[hr_faq_agent],
    tasks=[task],
    knowledge_sources=[faq_knowledge],
    embedder=ollama_embedder,
    process=Process.sequential,
    verbose=True
)

result = crew.kickoff(inputs={
    "question": "What are the working hours and how do I get reimbursed?"
})

print(result)
```

We get the following response:

```
# Agent: HR Assistant
## Task: Answer this onboarding question: What are the working hours and how do I get reimbursed?


# Agent: HR Assistant
## Final Answer:
Our standard working hours are from 9:00 AM to 5:00 PM, Monday through Friday. However, flexible work

1. Complete the expense reimbursement form, available on the company intranet.
2. Attach all relevant receipts and documentation that support your claim.
3. Submit the completed form to your direct supervisor for approval.
4. Once approved, the form will be forwarded to the finance department for processing.

Reimbursements are typically processed within two pay cycles. If you have any questions or need assis


('Our standard working hours are from 9:00 AM to 5:00 PM, Monday through '
 'Friday. However, flexible working arrangements may be available based on '
 'management approval. To get reimbursed for approved expenses, please follow '
 'these steps: \n'
 '\n'
 '1. Complete the expense reimbursement form, available on the company '
 'intranet.\n'
 '2. Attach all relevant receipts and documentation that support your claim.\n'
 '3. Submit the completed form to your direct supervisor for approval.\n'
 '4. Once approved, the form will be forwarded to the finance department for '
 'processing.\n'
 '\n'
 'Reimbursements are typically processed within two pay cycles. If you have '
 'any questions or need assistance with the reimbursement process, please '
 'contact the HR department directly.')
```

In the above setup:

- We passed the onboarding policy as a string knowledge source.
- We used an embedding model served through Ollama locally.
- The agent queried the embedded memory and returned the answer.

While we have discussed this before, customizing embeddings is especially useful when:

- You want to run entirely on-prem or offline.
- You need to align embedding logic with your data structure (e.g., short fields vs. long paragraphs).
- You want to use a cheaper or faster model for embeddings while keeping your LLM unchanged.

With CrewAI, the embedding providers are supported— `openai` , `google` , `cohere` , `huggingface` , `azure` , `AWS bedrock` , `ollama` , and more.

You can choose the one that matches your stack, cost structure, or privacy requirements.

Conclusion, takeaways, and next steps

With that, we come to the end of Part 7 of the Agents crash course.

In this part, we explored how to equip agents with knowledge—giving them the ability to recall, reference, and reason over contextual information. This takes your agents from being prompt-bound to memory-augmented, allowing for far more robust and informed behaviors.

We covered the full spectrum:

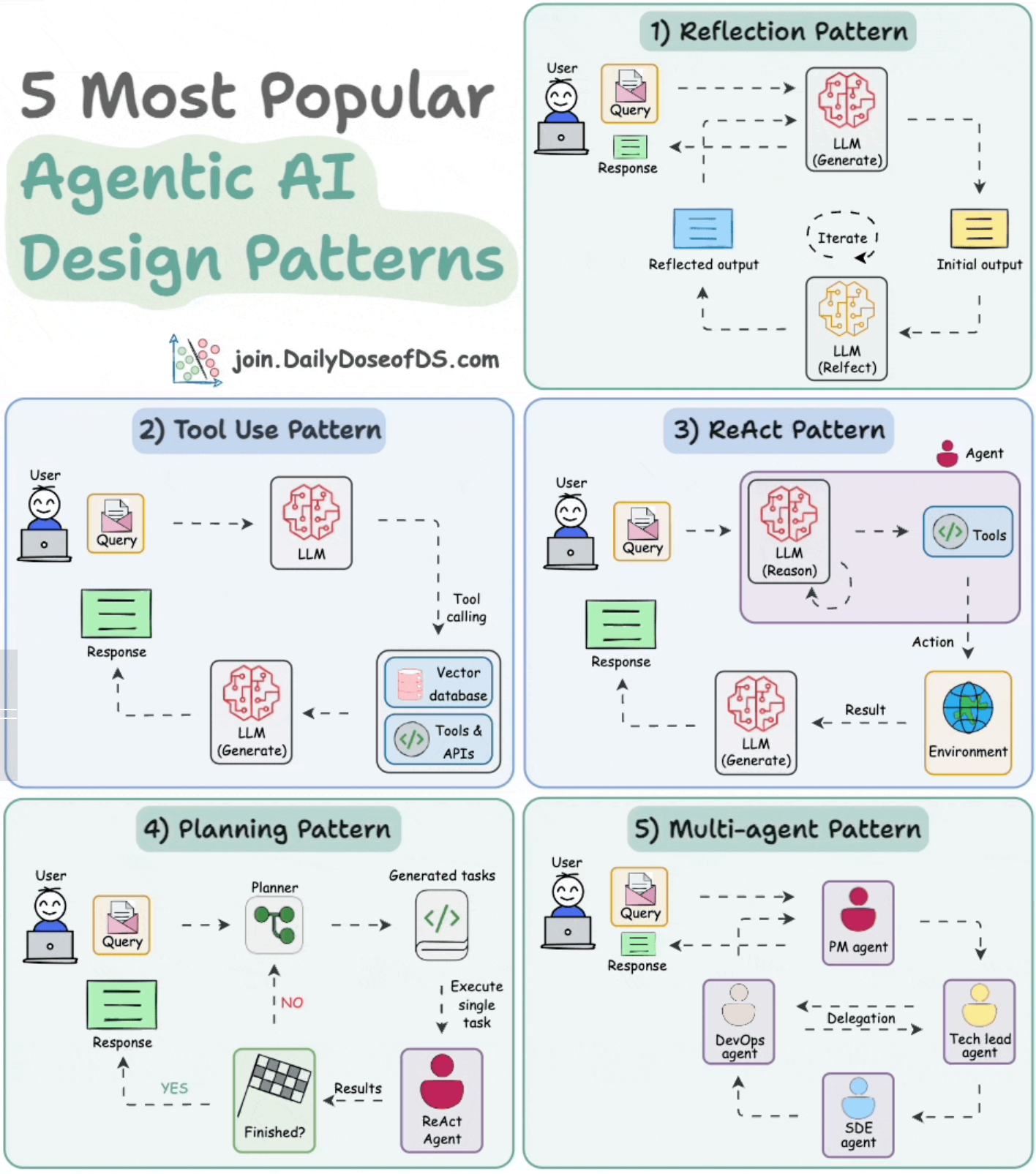
- How to use lightweight string-based knowledge sources
- How to chunk content and control how it's embedded
- How to control knowledge access at the agent or crew level
- How to work with real-world formats like text, PDF, CSV, JSON, and Excel
- How to define and plug in custom knowledge sources from external APIs or systems
- How to customize embedding behavior for privacy.

This gives you a complete foundation for turning your agents into intelligent, context-aware systems—capable of making decisions based on documents, datasets, and structured memory.

SOon, we'll take this further by looking at long-term memory, cross-task context retention, and how to design workflows that adapt and evolve across sessions.

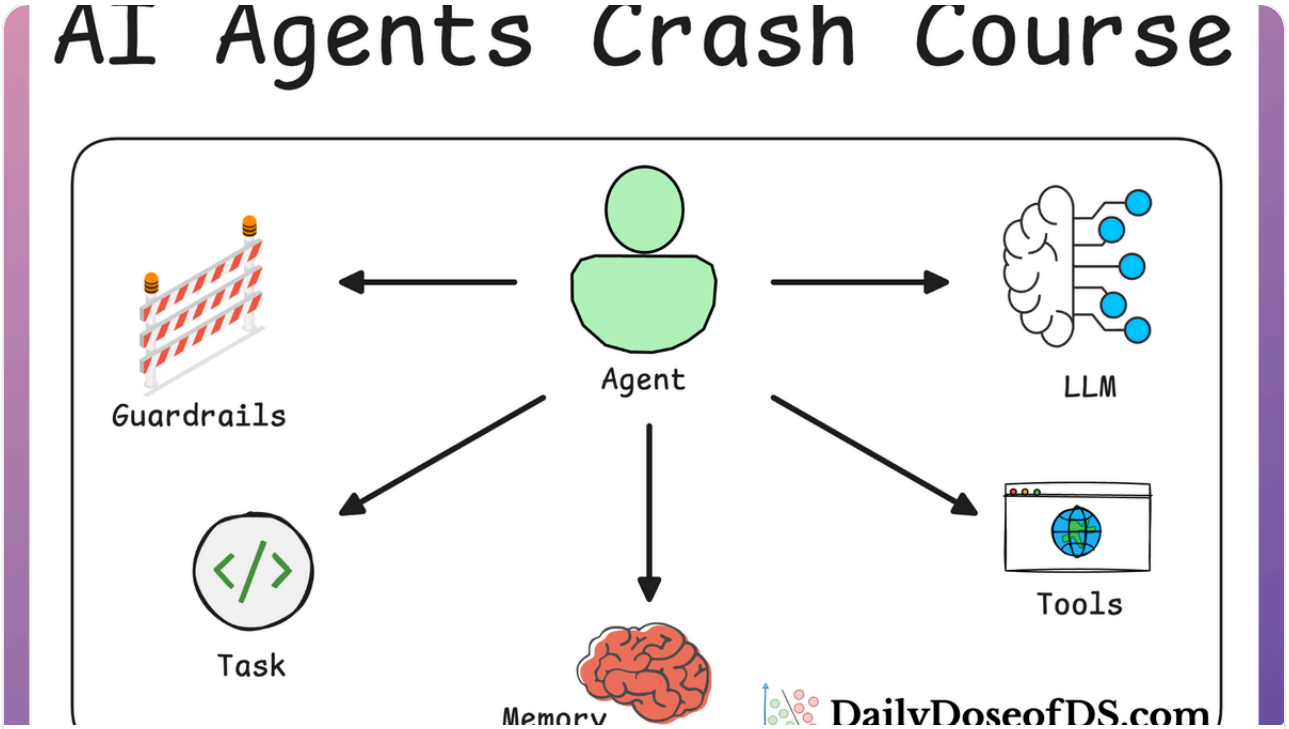
Moreover, in the upcoming parts, we have several other advanced agentic things planned for you:

- Building production-ready agentic pipelines that scale.
- Agentic RAG (Retrieval-Augmented Generation) – combining RAG with AI agents [we understood this a bit in this part too].
- Optimizing agents for real-world applications in business and automation.
- Building Agents around the Agentic patterns depicted below:



- and many many more.

Read the next part of this crash course here:



A Practical Deep Dive Into Memory for Agentic Systems (Part A)

AI Agents Crash Course—Part 8 (with implementation).

Daily Dose of Data Science • Avi Chawla

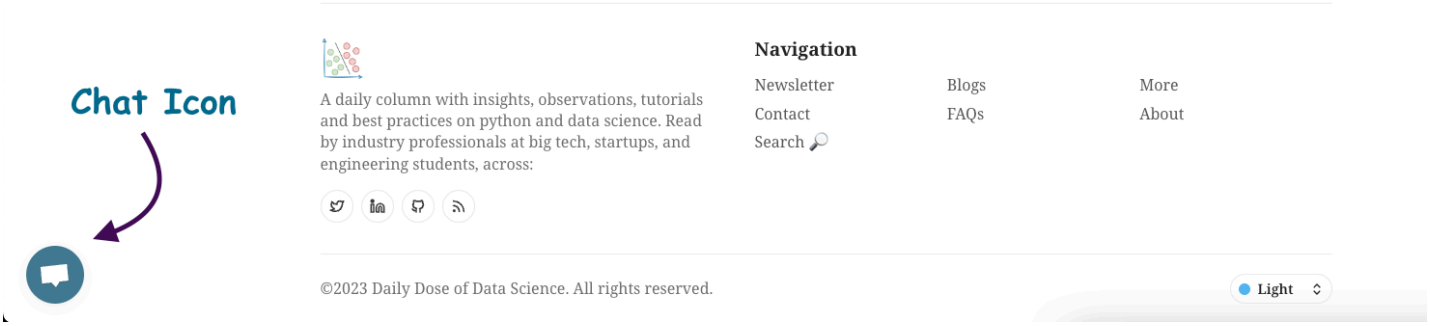
As always, thanks for reading!

Any questions?

Feel free to post them in the comments.

Or

If you wish to connect privately, feel free to initiate a chat here:



Connect via chat

Agents LLMs AI Agent Crash Course

Share this article

Read next

MCP Jun 15, 2025 • 22 min read

The Full MCP Blueprint: Building a Full-Fledged MCP Workflow using Tools, Resources, and Prompts

Model context protocol crash course—Part 4.

Avi Chawla, Akshay Pachaar

Agents Jun 8, 2025 • 20 min read

The Full MCP Blueprint: Building a Custom MCP Client from Scratch

Model context protocol crash course—Part 3.

Avi Chawla, Akshay Pachaar

Agents Jun 1, 2025 • 22 min read

The Full MCP Blueprint: Background, Foundations, Architecture, and Practical Usage (Part B)

Model context protocol crash course—Part 2.

Avi Chawla, Akshay Pachaar