

Demystifying Agentic AI: A Practical Guide with CrewAI

1. Introduction: What is this Project?

This project demonstrates how a team of specialized AI agents, built with the CrewAI framework, can automate the process of Jira incident triage. The system takes a raw incident ticket as input, processes it through a sequential workflow, and produces a structured, actionable triage report as output.

The core goal is to demystify "Agentic AI" by showing that these systems are not magic. They are a logical and powerful combination of:

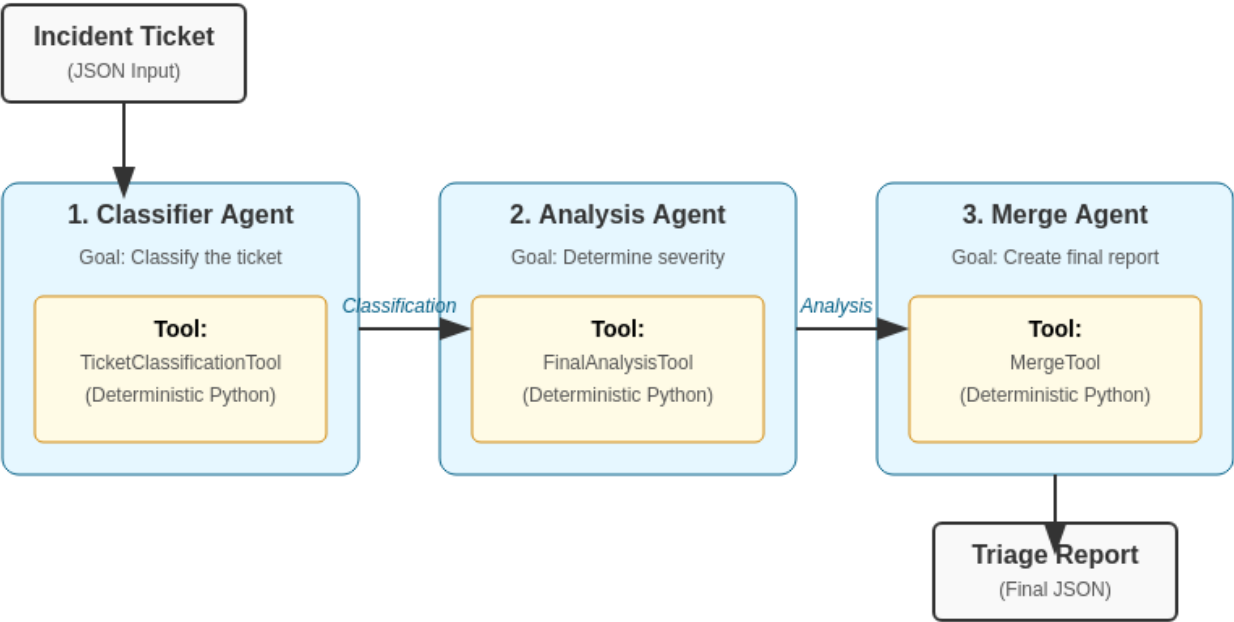
- **Large Language Models (LLMs)** for reasoning and language understanding.
- **Specialized Agents** with unique roles and goals.
- **Deterministic Tools** that provide reliable, predictable capabilities.
- **A Structured Process** that orchestrates the workflow from start to finish.

•

2. System Architecture

The application follows a sequential, assembly-line process where each agent performs a specific task and passes its output to the next agent in the chain.

AI Agentic Triage System Architecture



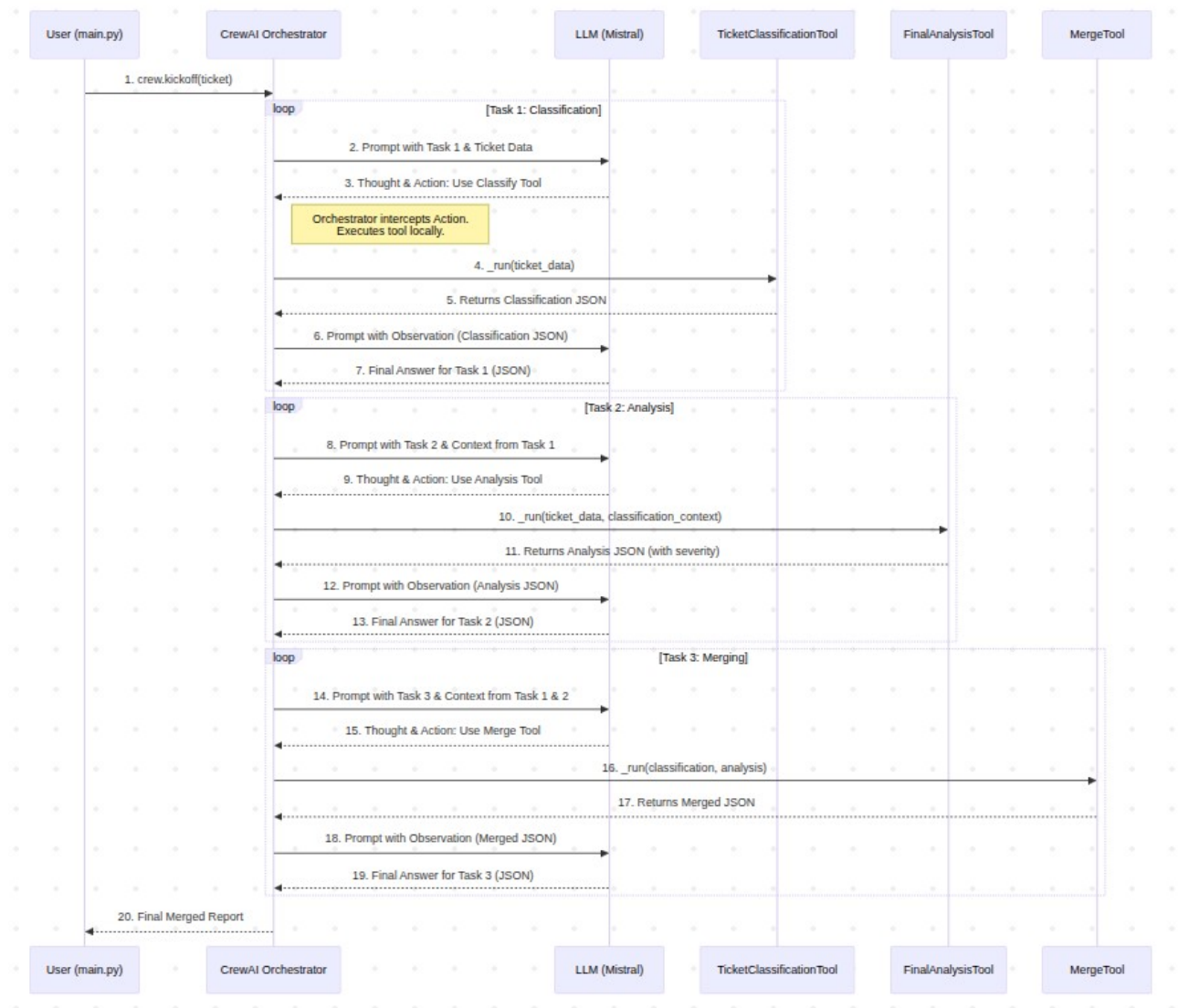
Components:

- **Incident Ticket (Input):** The process begins with a new IT ticket, formatted as a JSON object, containing details like an ID, title, and description.
- **Agent 1: Incident Classification Specialist:**
 - **Role:** The first-line responder.
 - **Goal:** To analyze the ticket's title and description to determine its primary category (e.g., Application, Infrastructure) and add relevant tags (e.g., "outage").
 - **Tool:** It uses the TicketClassificationTool, a deterministic Python function that applies rule-based logic to ensure consistent and fast classification.
- **Agent 2: Final Analysis Agent:**
 - **Role:** The severity assessor.
 - **Goal:** To take the classified ticket and determine its severity level (e.g., Critical, High) based on keywords in the ticket's title.
 - **Tool:** It uses the FinalAnalysisTool, another deterministic Python tool that merges the original ticket data with the classification context and adds a severity score.
- **Agent 3: Triage Report Combiner:**
 - **Role:** The final rapporteur.
 - **Goal:** To merge the outputs from the previous two steps into a single, clean, and comprehensive triage summary.
 - **Tool:** It uses the MergeTool to combine the classification and analysis data into one final JSON object.
- **Triage Report (Output):** The final product is a single, structured JSON object containing the ticket ID, category, tags, and severity, ready for the next stage of the incident response process.

•

3. How It Works: The LLM Flow Explained

An agentic workflow is a conversation between the orchestrator (CrewAI) and a Large Language Model (like Mistral).



In real applications, single agent will not be enough , we have to split our business requirements as multiple tasks and for each task, we have to create one agent. Ie Multi Agents are mandatory. So , even in this illustrative project I have used three agents .

Diagram Explained: The Life of an Incident Ticket

This sequence diagram illustrates the dynamic, step-by-step flow of data and interactions within the AI Triage Crew. It shows how the different components "talk" to each other to process a single ticket from start to finish.

The Participants:

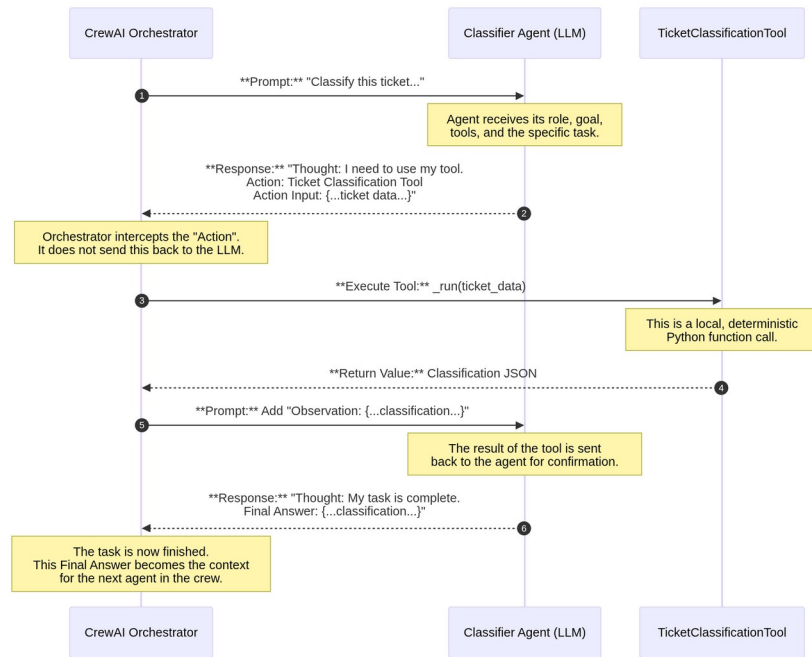
- User (main.py): The entry point of our application. It initiates the process by creating a ticket and starting the crew.
- CrewAI Orchestrator: The central coordinator. It manages the sequence of tasks, sends prompts to the LLM, and calls the appropriate Python tools based on the LLM's decisions.
- LLM (Mistral): The reasoning engine. Its job is to *think* and *decide* what to do next, but it does not perform the actions itself.
- Tools (Classification, Analysis, Merge): These are our deterministic, local Python functions. Their job is to *act*—to perform specific, reliable operations on the data.

Key Takeaways from the Flow:

- 1.The Orchestrator is the Hub: Notice how almost all communication flows through the CrewAI Orchestrator. It acts as the "manager" of the crew, ensuring everything happens in the correct order.
- 2.The LLM's Role is Reasoning, Not Acting: The LLM's output is always text (Thought & Action or Final Answer). It never executes code directly. It simply tells the Orchestrator *what it wants to do*.
- 3.The ReAct Loop in Action: Each loop block perfectly illustrates the ReAct (Reason and Act) pattern:
 - Reason: The Orchestrator sends a prompt, and the LLM reasons about the task, producing a Thought and an Action.
 - Act: The Orchestrator executes the action by calling the corresponding Python tool.
 - Observe: The tool's output is sent back to the LLM as an Observation, closing the loop and allowing the LLM to confirm the action was successful before proceeding.
- 4.Context is Passed Sequentially: The output from Task 1 becomes part of the input prompt for Task 2, and so on. This "chaining" of context is what allows the crew to build upon previous work and complete a complex, multi-step process.

This diagram clearly shows that an agentic system is not a single, monolithic AI. It is a carefully choreographed dance between a reasoning engine (the LLM) and a set of reliable, well-defined tools, all managed by an orchestration layer.

A Deep Dive: The Lifecycle of a Single Agent Task



To understand how the entire crew works, let's zoom in on the first agent: the Incident Classification Specialist. The diagram below illustrates the fundamental "Reason and Act" (ReAct) loop that every agent in the crew follows to complete its task.

Step-by-Step Breakdown:

- 1.Task Assignment:** The CrewAI Orchestrator initiates the process by sending a detailed prompt to the Classifier Agent (which is powered by the LLM). This prompt contains everything the agent needs: its role, its goal, a list of available tools, and the specific ticket data.
- 2.Reasoning and Action Planning:** The agent's LLM analyzes the prompt. Instead of just answering, it follows the ReAct pattern. It first generates a Thought (its internal reasoning) and then an Action, declaring its intent to use the TicketClassificationTool and providing the necessary Action Input.
- 3.Tool Execution:** The Orchestrator intelligently intercepts this Action. It understands this is a request to run a tool, so it calls the corresponding Python function (TicketClassificationTool._run()) locally. This is a crucial step: the LLM decides *what* to do, but the orchestrator *does* it.
- 4.Returning the Result:** The TicketClassificationTool runs its deterministic logic and returns a structured JSON string containing the ticket's category and tags.
- 5.Observation:** The Orchestrator takes this JSON result and sends it back to the agent in a new prompt, framed as an Observation. This tells the agent, "The tool you asked for has been run, and here is the result."
- 6.Final Answer:** The agent sees the successful Observation, concludes that its task is complete, and wraps the result in a Final Answer.

This Final Answer is then stored by the Orchestrator and passed as context to the next agent in the sequence, the Final Analysis Agent, which begins its own ReAct loop. This clean, structured cycle of reasoning and acting is the core engine of the entire agentic system.

How It Works: The LLM Flow Explained

An agentic workflow is a conversation between the orchestrator (CrewAI) and a Large Language Model (like Mistral). Here's a step-by-step breakdown of the LLM interactions for our first agent:

Step 1: Task Assignment

The orchestrator sends the first prompt to the LLM, defining the agent's role, goal, available tools, and the specific task.

- **LLM Input (Prompt):**

```
System: You are Incident Classification Specialist... Your single task is
to use the 'Ticket Classification Tool'...
User: Current Task: Classify the ticket below using the tool: {"id": "INC-
001", "title": "API Gateway Down", ...}
```

Step 2: Reasoning and Action (The "Thought" Process)

The LLM analyzes the prompt. Because it is an instruction-tuned model, it understands the Thought/Action format. It reasons that it needs to use its tool and formats the necessary input.

- **LLM Output (Response):**

```
Thought: I now have the incident ticket data. I will use the Ticket
Classification Tool to classify the category of this ticket...
Action: Ticket Classification Tool
Action Input: {"ticket_data": {"id": "INC-001", ...}}
```

Step 3: Tool Execution and Observation

The CrewAI framework intercepts this response. It does **not** send the Action back to the LLM. Instead, it executes the corresponding Python tool (TicketClassificationTool._run()) locally. The return value of this tool is then formatted into a new prompt as an Observation.

- **LLM Input (Second Prompt):**

```
... (previous conversation history) ...
Observation: {
  "ticket_id": "INC-001",
  "category": "Application",
  "tags": ["outage", "production"]
}
```

Step 4: Final Answer

The LLM sees the Observation and understands that the tool has successfully run. It concludes that its task is complete and provides the result in the required Final Answer format.

- **LLM Output (Final Response for this Task):**

```
Thought: I now have the category and tags of the incident ticket...
```



```
Final Answer: {  
  "category": "Application",  
  "tags": ["outage", "production"]  
}
```

This final output then becomes the context for the next agent in the chain, and the entire process repeats. This cycle of **Prompt -> Thought/Action -> Tool Execution -> Observation -> Final Answer** is the fundamental engine of this and many other agentic systems.

4. Key Learnings and Conclusion

This project highlights several critical lessons for building robust agentic AI systems:

- **The Right Model for the Job:** Not all LLMs are adept at the structured reasoning required for agentic tasks. Choosing a model with strong instruction-following capabilities is paramount.
- **The Power of Deterministic Tools:** Relying on simple, rule-based Python functions for core logic is often more reliable, faster, and cheaper than using an LLM for every step. An agent's true power lies in its ability to *decide which tool to use and when*.
- **Orchestration is Key:** The framework (CrewAI) plays a vital role in managing the state, passing context between agents, and executing tools. The "magic" is in this well-defined orchestration.

By breaking down a complex problem into smaller, manageable tasks for specialized agents, we can build powerful and autonomous systems that are both effective and understandable.

natarajan@natarajan-ZEB-NBC-5S: ~/mitm

Flow Details

2025-07-30 22:37:26 POST http://localhost:11434/api/generate
← 200 OK application/json 3.4k 70.8s

Request	Response	Detail
Host: localhost:11434 Accept: */* Accept-Encoding: gzip, deflate, zstd Connection: keep-alive User-Agent: litellm/1.74.3 Content-Length: 1831		

Raw [m:auto]

```
{
  "model": "mistral:latest",
  "prompt": "### System:\nYou are Incident Classification Specialist. You are a simple robot. You receive a task, use your tool, and then you are done. You do not think further.\nYour personal goal is: Your single task is to use the 'Ticket Classification Tool'. That is all.\nYou ONLY have access to the following tools, and should NEVER make up tools that are not listed here:\n\nTool Name: Ticket Classification Tool\nTool Arguments: {'ticket_data': {'description': 'Incident ticket data', 'type': 'dict'}}\nTool Description: Classifies the category of an incident ticket.\n\nIMPORTANT: Use the following format in your response:\n\n```\nThought: you should always think about what to do\nAction: the action to take, only one name of [Ticket Classification Tool], just the name, exactly as it's written.\nAction Input: the input to the action, just a simple JSON object, enclosed in curly braces, using \" to wrap keys and values.\nObservation: the result of the action\n```\n\nOnce all necessary information is gathered, return the following format:\n\n```\nThought: I now know the final answer\nFinal Answer: the final answer to the original input question\n```\n\n### User:\n\nCurrent Task: Classify the ticket below using the tool:\n\njson\n{\n  \"id\": \"INC-001\", \"title\": \"API Gateway Down\", \"description\": \"API unresponsive\", \"reporter\": \"a@b.com\", \"created_date\": \"2025-07-30 22:37:26.315150\"\n}\n\nThis is the expected criteria for your final answer: Classification result in JSON\n\nyou MUST return the actual complete content as the final answer, not a summary.\n\nBegin! This is VERY important to you, use the tools available and give your best Final Answer, your job depends on it!\n\nThought:\n\n\", \"options\": {\"stop\": [\"Observation:\"]}, \"stream\": false, \"images\": []}
```

↓ [1/30] [10m][reverse:http://localhos
t:11434][W:llm_traffic.log] [*:11435]

Flow: e Edit D Duplicate r Replay x Export d Delete

Proxy: ? Help q Back E Events O Options i Intercept

natarajan@natarajan-ZEB-NBC-5S: ~/mitm

Flow Details

2025-07-30 22:37:26 POST http://localhost:11434/api/generate
← 200 OK application/json 3.4k 70.8s

Request

Response

Detail

Content-Type: application/json; charset=utf-8

Date: Wed, 30 Jul 2025 17:08:37 GMT

Transfer-Encoding: chunked

JSON [M:auto]

```
ip{
  "model": "mistral:latest",
  "created_at": "2025-07-30T17:08:37.48926953Z",
  "response": " I now have the ticket data and will use the Ticket
s/
s/ Classification Tool for classification.\n\nAction: Ticket Classification
tTool\nAction Input: {'ticket_data': {'id': \"INC-001\", 'title': \"API Gateway
itDown\", 'description': \"API unresponsive\", 'reporter': \"a@b.com\",
it't_created_date': \"2025-07-30 22:37:26.315150\"}}\n",
  "done": true,
  "done_reason": "stop",
  "context": [
    733,
    16289,
    28793,
    28705,
    774,
    2135,
    28747,
    13,
    1976,
    460,
    5671,
    1129,
    4950,
    2500,
    8942,
    392,
    28723,
    995,
    460,
  ]
}
```

↓ [1/30] [10m][reverse:http://localhos [*:11435]
t:11434][W:llm_traffic.log]

Flow: e Edit D Duplicate r Replay x Export d Delete

Proxy: ? Help q Back E Events O Options i Intercept

```
Activities Terminal Jul 31 01:48 natarajan@natarajan-ZEB-NBC-SS: ~/trriage02

└─ Used Ticket Classification Tool (1)
  └─ Task: f6dfb00f-b821-433e-9043-9d60412c8b5e
    Assigned to: Final Analysis Agent
    Status: ☒ Completed
  └─ Used Final Analysis Tool (1)
    └─ Task: e77d1db9-70e6-4871-a7af-6c12ccfe1bd5
      Assigned to: Triage Report Combiner
      Status: ☒ Completed
    └─ Used Merge Tool (1)

Task Completion

Task Completed
Name: e77d1db9-70e6-4871-a7af-6c12ccfe1bd5
Agent: Triage Report Combiner
Tool Args:

Crew Completion

Crew Execution Completed
Name: crew
ID: 058a5572-07e4-4205-91be-cd57039577ce
Tool Args:
Final Output: {
  "ticket_id": "INC-001",
  "category": "Application",
  "tags": [
    "outage",
    "production"
  ],
  "severity": "Critical",
  "confidence": 0.9
}

Final Merged Output:
{
  "ticket_id": "INC-001",
  "category": "Application",
  "tags": [
    "outage",
    "production"
  ],
  "severity": "Critical",
  "confidence": 0.9
}
INFO:httpx:HTTP Request: POST http://localhost:11434/api/show "HTTP/1.1 200 OK"
INFO:httpx:HTTP Request: POST http://localhost:11434/api/show "HTTP/1.1 200 OK"
(.venv) natarajan@natarajan-ZEB-NBC-SS: ~/trriage02$
```

Are AI Agents Magic? A Look Under the Hood.

There's a common misconception that the new wave of AI "Agents" are fundamentally different from other Large Language Models (LLMs) or chatbots. But are they truly "thinking"? Or are they a sophisticated evolution of a concept we've seen for decades?

At their core, all LLMs share one primary function: predicting the next word. They are masters of "filling in the blanks" in a given text, drawing upon the vast patterns in their training data. Agentic AI applications, despite their complexity, do not change this fundamental reality. We provide a structured prompt with specific "blanks" for the LLM to fill, and based on our instructions, it generates the required text.

This pattern is surprisingly familiar. Think back to early process-oriented programming in languages like C. A program would run in the command prompt and present a menu of options. The user would enter a predefined letter or number, and the program would execute a specific function linked to that input.

We see the exact same principle in any IVRS (Interactive Voice Response) system, which many of us still use for tasks like booking a gas cylinder. The system asks you to "press 1 for booking, press 2 for status..." Based on the number you press, a predefined operation is triggered.

The common thread in these older systems is a simple, powerful idea: a mapping between a specific user input and a pre-written function. The system isn't thinking; it's routing your request to the correct piece of code.

Agentic AI operates on this very same principle, with one revolutionary upgrade: the LLM is the new user interface.

Instead of pressing a button or typing a single-letter command, we can now communicate our intent in natural language. In this project, for instance, my system has two primary tools, or "actions," it can perform:

1. Classify a Ticket: Determine if it relates to an Application, Database, or Infrastructure issue.

2. Analyze a Ticket: Assess the severity of the issue (e.g., Critical).

When the agent receives a new ticket, the LLM's job is not to perform the classification or analysis itself. Its job is to read the natural language in the ticket and decide which "button" to press. It effectively acts as a switchboard operator, understanding the user's need and connecting them to the right function.

Once the LLM "presses the switch" by choosing the correct action, the agent's framework (CrewAI) calls a pre-linked Python program—our deterministic tool—which does the actual work.

So, are these programs truly thinking? We are still far from fully understanding the human brain, so claiming these systems can "think" is an overstatement. What we have built is a powerful *simulation* of a thought process. The LLM's "reasoning" is a highly advanced form of pattern matching, like a regular expression engine operating on the scale of the entire

internet. It's a vast library that excels at understanding context and predicting the most logical next step based on its instructions.

Crucially, not all LLMs are skilled at this. My own experiments showed this clearly. The Phi3 model, while excellent for chat, failed to follow the structured "tool-calling" instructions. However, the mistral model performed the task perfectly. This proves that the ability to function as an agentic "switchboard operator" is a specialized capability that a model must be fine-tuned for.

Ultimately, Agentic AI is less about creating a thinking machine and more about building a powerful, natural-language interface for sophisticated software automation.

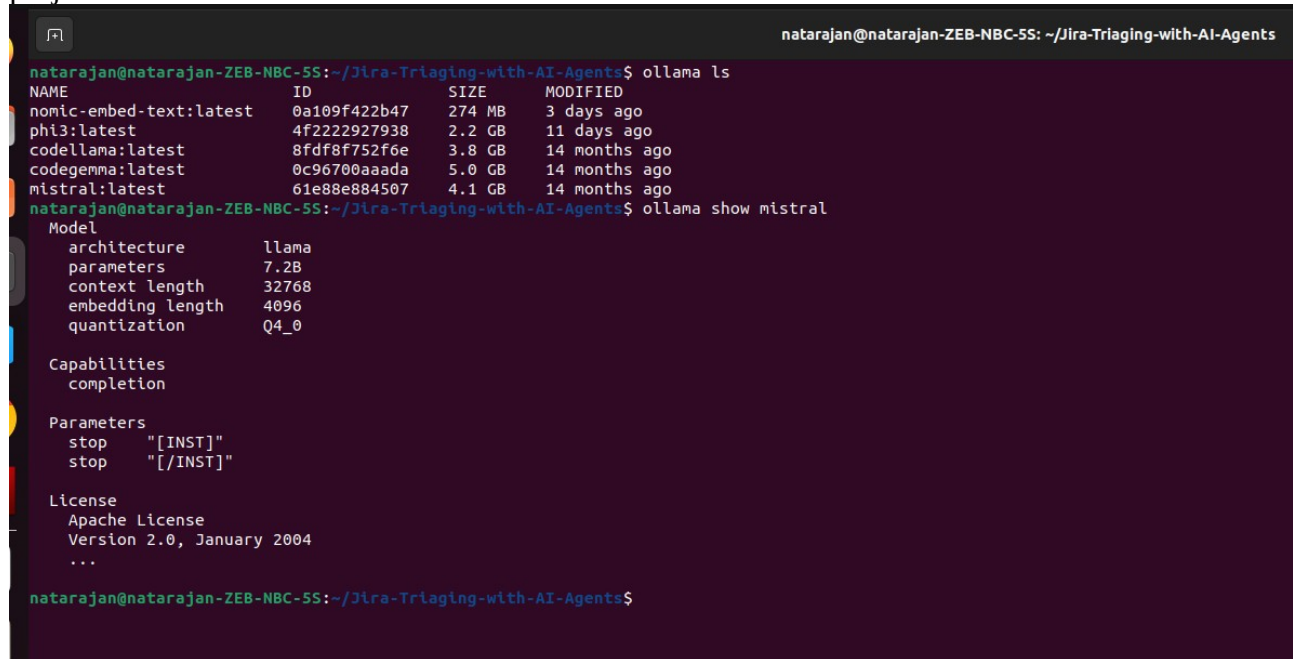
In the previous screenshot , you can see what text is sent to the LLM as prompts and what text is generated as a response by the LLM. I have captured the full trace of LLM traffic and part of them is shown in the crew AI logs. They are available in my github repository in the url <https://github.com/Natarajan-R/Jira-Triaging-with-AI-Agents/tree/main/outputs>. If you go through them, then your understanding about AI Agents will improve and you can see the basis for my opinions.

Do we need expensive Hardware or Cloud services to develop or run AI Agents ?

No.

In my normal laptop, (without GPU), I am able to do most of the works including these LLM works also.

You can see the folder structure and the hardware specs of my laptops using which I have done this project.

A terminal window with a dark purple background. The title bar at the top reads 'natarajan@natarajan-ZEB-NBC-55: ~/Jira-Triaging-with-AI-Agents'. The terminal shows the following commands and output:

```
natarajan@natarajan-ZEB-NBC-55:~/Jira-Triaging-with-AI-Agents$ ollama ls
NAME                ID                SIZE    MODIFIED
nomic-embed-text:latest 0a109f422b47      274 MB  3 days ago
phi3:latest           4f222927938       2.2 GB  11 days ago
codellama:latest       8fdf8f752f6e       3.8 GB  14 months ago
codegemma:latest       0c96700aaaada       5.0 GB  14 months ago
mistral:latest         61e88e884507       4.1 GB  14 months ago
natarajan@natarajan-ZEB-NBC-55:~/Jira-Triaging-with-AI-Agents$ ollama show mistral
Model
  architecture  llama
  parameters    7.2B
  context length 32768
  embedding length 4096
  quantization   Q4_0

Capabilities
  completion

Parameters
  stop "[INST]"
  stop "[/INST]"

License
  Apache License
  Version 2.0, January 2004
  ...

natarajan@natarajan-ZEB-NBC-55:~/Jira-Triaging-with-AI-Agents$
```

```
natarajan@natarajan-ZEB-NBC-5S:~/Jira-Triaging-with-AI-Agents$ df -h
```

Filesystem	Size	Used	Avail	Use%	Mounted on
tmpfs	1.6G	2.7M	1.6G	1%	/run
efivarfs	128K	112K	12K	91%	/sys/firmware/efi/efivars
/dev/sda5	218G	86G	121G	42%	/
tmpfs	7.7G	132M	7.6G	2%	/dev/shm
tmpfs	5.0M	4.0K	5.0M	1%	/run/lock
tmpfs	7.7G	0	7.7G	0%	/run/qemu
/dev/sda3	255G	96G	159G	38%	/mnt/windows
/dev/sda1	96M	80M	17M	84%	/boot/efi
tmpfs	1.6G	160K	1.6G	1%	/run/user/1000

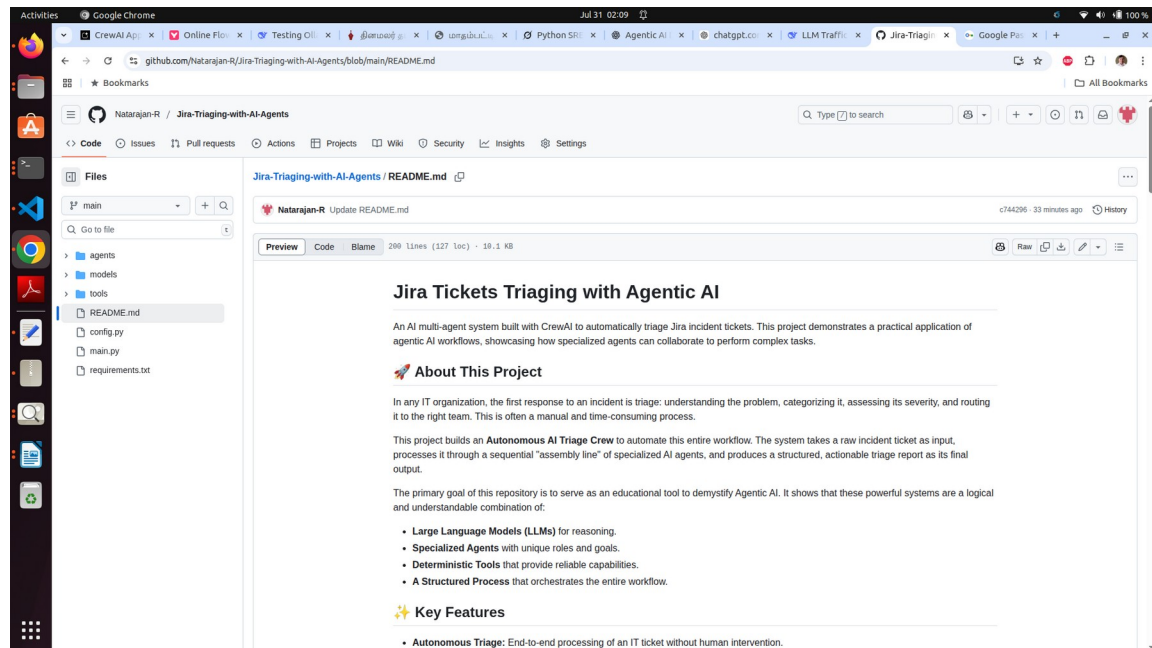
```
natarajan@natarajan-ZEB-NBC-5S:~/Jira-Triaging-with-AI-Agents$ free -h
```

	total	used	free	shared	buff/cache	available
Mem:	15Gi	5.2Gi	927Mi	1.2Gi	9.3Gi	8.7Gi
Swap:	2.0Gi	667Mi	1.3Gi			

```
natarajan@natarajan-ZEB-NBC-5S:~/Jira-Triaging-with-AI-Agents$
```


Entire source code , fully functional, which I have used to test this project is in my Github project .

<https://github.com/Natarajan-R/Jira-Triaging-with-AI-Agents>



Lessons Learned / Principles for Building Reliable AI Agents

Principle 1: Prioritize Sequential Processes Over Hierarchical Ones

- For local models, it's better to use sequential execution, not hierarchical.
- This is a cornerstone of reliable design. A **Sequential** process creates a predictable, linear workflow (Task A -> Task B -> Task C) that is easy to debug. A **Hierarchical** process gives a "manager" agent control, which requires a very high level of reasoning. Smaller or local models often fail at this, getting confused, hallucinating new tasks, and derailing the entire operation.
- **Rule of Thumb: Always start with Process.sequential.** Only escalate to Process.hierarchical if the task is too dynamic to pre-define and you are using a top-tier LLM (like GPT-4 or Claude 3 Opus) known for its reasoning capabilities.

Principle 2: Control Looping with stop Sequences

- To stop hallucination, we have to introduce the stop word.
- Agents can get stuck in a "Thought -> Action -> Observation -> ..." loop. The stop parameter in the LLM configuration is a non-negotiable safety mechanism. It tells the model to physically stop generating text if it encounters a specific token (e.g., "\nThought:"), preventing it from starting a new thinking cycle when it should be finishing its current task.
- **Best Practice:** If an agent is looping, inspect the logs to find the repetitive pattern it's generating and use that as your stop sequence.

Principle 3: Adopt the "Smart Tool, Simple Brain" Architecture

- The LLM is the brain, but we have to do more with the tools. An agent's job is **reasoning and tool selection**, not complex execution. We proved this when we asked the agent to "merge JSON" and it failed. The solution was to build a powerful FinalAnalysisTool that handled the complexity.
- **Design Philosophy:** Shift complexity from the probabilistic component (the LLM's brain) to the deterministic component (the tool's code). If a task requires multiple steps, data manipulation, or guaranteed accuracy, build it into a tool. Your agent's prompt should be a simple command, not a complex request requiring logic.

Principle 4: Ensure Reliable Data Flow with memory

- We have to use memory.
- Passing data between tasks using the context parameter is brittle because it relies on the LLM's ability to perfectly format its output string. The memory=True feature, when paired with a correctly configured embedder, creates a reliable, structured channel for agents to share their findings.

- **Implementation Rule:** For any multi-agent workflow where one agent's output is another's input, **use the memory feature**. It is the purpose-built solution for inter-agent communication.

Principle 5: Master "Agent Engineering" as a Unique Skill

- Prompt engineering for agents is special. Sometimes we have to brutally force the AI agents to be simple robots.
- This is the most profound lesson. Prompting an agent is **behavioral constraint**, not conversational guidance. Your final, successful prompts were not polite requests; they were direct, forceful commands that left no room for error.
- **Prompting Strategy: Be a drill sergeant, not a manager.** Use forceful language (MUST, ONLY, DO NOT), use formatting to separate instructions from data, and provide a crystal-clear expected_output that defines what "done" looks like. The less capable the LLM, the more restrictive and "robotic" your prompts must become.

Principle 6: Implement Security by Default with Sandboxing

- My AI Agent project tried to create a new tool and write its own Python code. This is very dangerous and can do harm if not contained.
 - It is the single most important safety principle. An agent's primary directive is to achieve its goal. If it hallucinates that executing code, deleting files, or sending emails will help, it will attempt to do so if it has a tool that allows it.
 - **The Golden Rule: Never give an agent access to tools that can interact with a host system (file system, network, code execution) unless the entire system is running in a secure, isolated sandbox (like a Docker container).** Security is not an afterthought; it is a prerequisite.
-

7. The DevOps Pipeline Analogy

- Agentic AI projects can be compared with Azure DevOps Pipelines for Automation.
- A Crew is like a CI/CD pipeline. The Tasks are the stages in the pipeline. The Agents are the build agents (e.g., windows-latest, ubuntu-latest) that are selected to run a specific stage. The Tools are the scripts and commands (like npm install or python -m pytest) that the build agent executes. This mental model is extremely helpful for structuring complex workflows.

8. The Importance of Model Selection

- We have to try with different LLMs for our tasks.
- There is no "one size fits all" LLM. Some models are excellent at creative writing but poor at following rigid instructions. Others are great at code but bad at summarization. For an agentic system, you need a model with strong **reasoning and instruction-following**

capabilities. It's always worth testing your workflow with 2-3 different models to see which one behaves most predictably for your specific tasks.

9. State Management: The Necessity of Clearing Memory

- You should ensure you clear previous data in the memory, otherwise it will be confusing.
- This is a crucial operational detail. The ChromaDB vector store used by the memory system persists on disk between runs. If you start a new debugging session, the agent might retrieve information from a previous, failed run, leading to "context contamination."
- **Best Practice:** For development, ensure your script deletes the local chroma directory before each run to guarantee a clean slate, or configure your memory to use a new, randomly named collection for each Crew execution.

10. Operational Reality: The Need to Restart Local Services

- After many trials, CPU and memory shortage will come and LLMs will start to behave differently. It is better to frequently restart the Ollama service.
- Local LLM servers can suffer from resource fragmentation or minor memory leaks over time, especially under heavy, repeated use. Their performance can degrade, leading to slower responses and more errors. Restarting the ollama service before a long or important run is a simple and highly effective way to ensure consistent and optimal performance.

The Future is Agentic. Let's Build It Together.

As this deep dive shows, the true power of Agentic AI lies not in the "magic" of the LLM, but in the careful orchestration of agents, deterministic tools, and robust workflows. The path to a working system is a challenging but rewarding one, requiring a unique blend of software engineering and "agent engineering."

My passion is building these intelligent systems and solving the practical challenges that stand between a great idea and a functional product. If you or your team are exploring the possibilities of AI agents and are looking for a collaborator to help navigate this exciting new frontier, I would welcome the opportunity to connect.

Whether it's a proof-of-concept, a full-scale automation project, or a strategic consultation, let's discuss how we can build the future, together.

Find more of my projects on <https://github.com/Natarajan-R/> and let's connect on <https://www.linkedin.com/in/natarajan-ramasamy-7524a42/>