Chapter 1

Agent Framework Implementation

The development of a custom Multi-Agent framework was undertaken after careful analysis of existing solutions, including CrewAI[1] and LangGraph[2]. These established platforms offer features for orchestrating LLM agents, yet neither fully addressed the requirements set by the research objectives described in **??**. Building a new framework became necessary to enable flexible communication with multiple end-users, referred to here as Process Workers, and to support agent autonomy as articulated by the Gaia methodology [3]. Unlike workflow-oriented systems, agents in this design retain the ability to choose their next actions and select with whom to interact, reflecting a more realistic and dynamic approach to coordination.

This chapter outlines the rationale and main design principles behind the developed framework. Several core requirements guided the implementation:

- **Human interaction interface:** Agents must be able to initiate a dialog with users, ask clarifying questions, and request additional context when needed.
- **Tool integration:** The framework should enable agents to employ specialized tools or functions, extending their capabilities and equipping them with helpful tools.
- **Configurable agent structures with communication:** The framework needs to support the construction of complex agent hierarchies and facilitate message-passing between agents.
- **Model agnosticism:** The framework should not be tied to any specific LLM provider or architecture.
- **High observability:** The research demands detailed monitoring of agent behavior, internal state, and message exchanges. This requirement ensures the system not only produces experimental results, but also captures the granular data needed for analysis and troubleshooting.

1.1 Components

To ensure model agnosticism and adaptability, the agent implementation is divided into modular components. The architecture separates language model functionality from the agent logic. An abstract LLM class standardizes the interface to various model providers, offering a unified set of methods and protocols. The Agent class, then, is defined independently and receives an LLM as a dependency through injection. This architecture enables the exchange or upgrade of models without altering agent logic. Figure 1.1 gives an overview of this split. Each component is described in detail below.

[1] CrewAI Documentation
[2] LangGraph Documentation

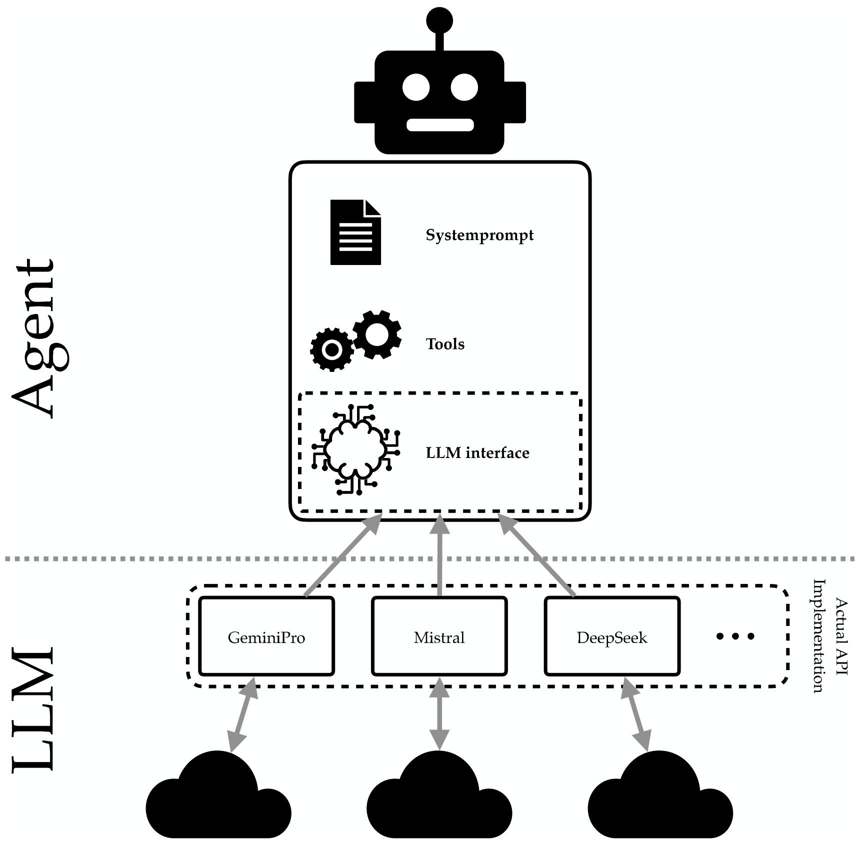


Figure 1.1: Overview of the Agent architecture, where the actual implementation of an agent is independent of the LLM.

1.1.1 Agents

Agents are the primary conceptual units within the framework. Building on the definition provided in **??**, each agent combines a system prompt, a set of tools, and an underlying LLM. The agent's behavior, operational boundaries, and ability to orchestrate tool use are governed by these elements.

Therefore the key properties of an agent include:

- **Name:** A unique natural language identifier for clear communication and routing among agents in Multi-Agent contexts.
- **System Prompt:** An instruction set that defines the agent's goals, constraints, and operational context.
- **Tools:** A list of functions the agent can utilize to perform tasks or gather information beyond language-only operations.
- **LLM:** The injected large language model, which serves as the decision-making core and generator of output based on the system prompt and with the help of tools.
- **Max Iterations:** A limit on agent invocations within a single request, preventing infinite loops and facilitating control over computational resources.

In addition to these configurable aspects, every agent manages an internal memory. This memory accumulates all messages exchanged during an interaction, forming a traceable record of conversation and task progress. It serves both as context for ongoing work and as an environment grounding the agent's actions.

Agents operate on a straightforward interaction loop. When a message is received, it is passed to the LLM, which processes the input and generates a response. If the response requires a tool call, the specified tool is executed, and the result is provided back to the agent. This exchange continues iteratively until the agent outputs a final natural language response, indicating task completion. Each cycle is logged for transparency and analysis. This process supports both simple single-agent interactions and more complex collaborations among multiple agents.

1.1.2 Large Language Models

At the core of each agent lies an LLM, which is responsible for interpreting prompts and generating agent output. The framework enforces a consistent structure for LLM interaction, so agents can operate with different models without rewriting core logic.

Inputs to an LLM are structured as follows:

- **System Prompt:** An initial directive that establishes task guidelines and role context.
- **Messages:** An ordered list capturing the ongoing conversation, including all user and model exchanges, tool calls, and tool results.
- **List of Tools:** Available tools that the model can instruct to call in its response, depending on the tool mode.
- **Tool Mode:** A flag specifying how the LLM should respond, either with a tool call, natural language output, or by making the choice autonomously.

Four main message types are defined:

- **User Text Message:** Natural language input.
- **Model Text Message:** Output from the LLM in natural language.
- **Model Tool Call:** Structured request to invoke a tool, with arguments.
- **Tool Response:** Output or error resulting from a tool execution.

Notably, the term ”user” in the LLM interface refers to any entity providing input, not necessarily a human user, in Multi-Agent systems, this can be another agent as well.

Message sequences strictly alternate between user and model roles, as required by some LLM providers. This approach maintains compatibility and modularity across diverse API implementations.

The tool_mode parameter governs the expected model response:

- **auto:** The LLM may choose between responding with a natural language message or a tool call.
- **force:** The LLM must return a tool call signature.
- **none:** Only natural language responses are permitted.

The LLM processes all context and configuration before returning a single output, following the specified constraints. This standardized interface ensures reliable, robust agent behavior, and simplifies future model integration.

1.1.3 Tools

Tools are central to empowering agents with capabilities beyond text generation. Leveraging recent function-calling features in LLMs, tools allow agents to execute structured operations. This could involve interacting with external APIs, performing computations, or routing messages between agents.

Each tool is defined by several attributes:

- **Name:** Unique function name for invocation.
- **Description:** A concise summary describing the function and intended usage, guiding the LLM in the appropriate selection.
- **Parameters:** Argument specifications for the tool, each described by:
  - *Name:* Parameter identifier
  - *Type:* Accepted data type (e.g., string, integer)
  - *Description (optional):* Supplementary details clarifying expected input or allowed values
  - *Required flag:* Indicates mandatory or optional input
- **Function:** The callable implementation that performs the tool's core action.

Internally, tools are registered and presented to the LLM following the schema expected by each provider. The structure of tool definitions ensures that invocation requests are well-formed and valid. By clearly describing each tool and its parameters, the framework facilitates error checking before execution, enabling robust feedback and retrial mechanisms. This systematic design makes agents more reliable and extends their functionality, supporting both flexibility and operational transparency.

1.2 Composing Multi-Agent Systems

To form a Multi-Agent system, individual agents must cooperate and communicate effectively. In the proposed framework, this is achieved by enabling agents to establish direct connections with one another. Each agent is aware of which peers it can interact with and may choose to send messages as needed, based on its own reasoning and current context. There is no overarching, centralized orchestrator controlling the flow of these interactions. See Figure 1.2 for an example of how the agents collaborate.

This communication structure aligns with the choreography collaboration pattern commonly used in microservices architectures [1]. The essential characteristic of this approach is that the agent originating a message determines the recipient, initiating a direct exchange. When an agent sends a message to another agent, it is effectively opening a communication channel; the recipient will address the request, returning a response. During its response process, the recipient agent may use tools or even contact additional agents to compile the required information or complete the task before replying.

This system relies on the use of unique agent names to facilitate direct and unambiguous communication. Each agent refers to other agents by their designated names, ensuring messages are accurately routed and tracked within the system.

To simplify these connections, the framework incorporates principles from the Gaia methodology, specifically the acquaintance model as defined by Wooldridge et al. [3]. Before runtime, the set of agent-to-agent communication pathways is already specified, this reduces complexity and supports predictable system behavior. This predefinition establishes which agents any given agent can reach out to during operation.

Additionally, using interaction models from Gaia, the expected content and structure of exchanged messages are determined in advance. Each agent can define the acceptable formats for incoming messages from each peer, as well as the structure of its outgoing responses. This attention to message specifications ensures that agents interact according to well-defined protocols, enabling seamless cooperation and reducing the likelihood of errors arising from miscommunication.

Overall, this compositional approach enables scalable, flexible Multi-Agent systems where agents coordinate dynamically, yet within a rigorously defined set of interaction constraints. It supports both simple and complex interaction patterns, facilitating experimentation with various agent collaboration scenarios.

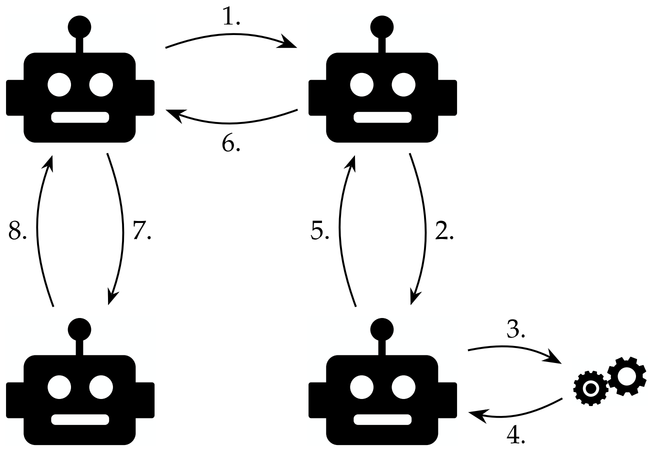


Figure 1.2: Visualization of a sample trace where the four agents get invoked by each other and solve the task collectively using the choreography pattern. The numbers on the arrows show the order in which the messages are triggered.

1.3 Handling Multiple Users

A central requirement of the framework is its ability to support interactions with multiple external users. This capability is important in the context of this research, where agents must communicate with several distinct individuals. However, most LLM interfaces are designed for one-on-one communication, with a simple alternation between user and model turns. While some LLM providers allow user tagging in message histories, depending on such vendor-specific features would undermine the model-agnostic design goal of the framework.

The challenge becomes more pronounced in a Multi-Agent environment. When an agent delegates tasks to another agent, it is passed to the agent as the user message. Without a clear mechanism for specifying the recipient and sender of a message, the dialogue context can become ambiguous, and the conversation history confusing for the LLM to interpret.

To address this, the framework employs its extensible tool system. Each agent that interfaces with external users is equipped with a dedicated user interface tool. This tool standardizes how agents interact with these external users by requiring two key parameters: the unique identifier of the external user and the message or question to be relayed. Upon invocation, the tool sends the prompt to the specified external user and records their reply for the agent's use. If the identifier is invalid or unavailable, the tool provides a clear error message, allowing the agent to handle the issue gracefully, such as by retrying or informing another agent.

This solution preserves straightforward user-agent exchanges, whether dealing with a single user or managing conversations with many participants. In cases where there is only one external user, the requirement to specify an identifier is removed for simplicity.

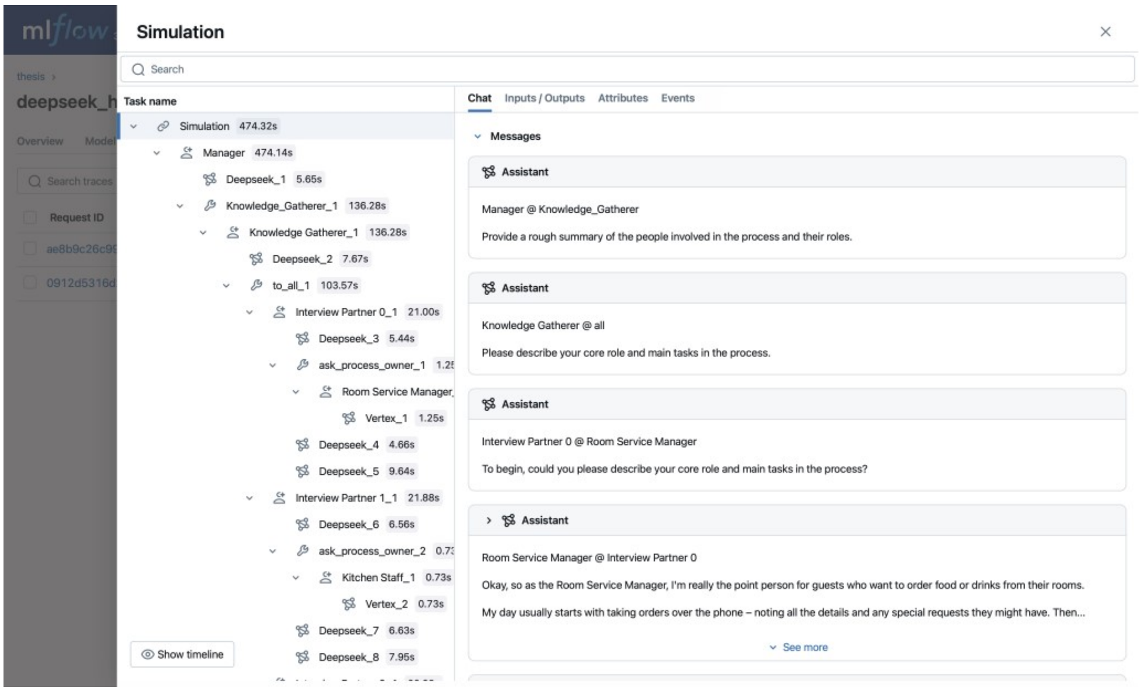
1.4 Observability

Robust observability is essential to ensure the framework's utility in research and development. To meet this requirement, the framework integrates MLflow [3], an open-source platform well

[3]https://mlflow.org/

suited for experiment tracking and artifact management. MLflow was selected for its ease of deployment, comprehensive API, and capacity to operate locally, making it accessible and adaptable for this research environment.

Every interaction with the LLM is systematically logged, capturing metadata such as token usage for each response. The framework also records all tool invocations, including their input parameters and any errors encountered during execution, whether originating from the tool logic itself or from the underlying LLM API. By leveraging the MLflow API, all agent messages are captured and organized per session, enabling complete traceability for each experimental run.



```
Task name
Simulation 474.32s
  Manager 474.14s
    Deepseek_1 5.65s
    Knowledge_Gatherer_1 136.28s
      Knowledge Gatherer_1 136.28s
        Deepseek_2 7.67s
        to_all_1 103.57s
          Interview Partner 0_1 21.00s
            Deepseek_3 5.44s
            ask_process_owner_1 1.2
              Room Service Manager
                Vertex_1 1.25s
            Deepseek_4 4.66s
            Deepseek_5 9.64s
          Interview Partner 1_1 21.88s
            Deepseek_6 6.56s
            ask_process_owner_2 0.7
              Kitchen Staff_1 0.73s
                Vertex_2 0.73s
            Deepseek_7 6.63s
            Deepseek_8 7.95s
```

Messages

Assistant

Manager @ Knowledge_Gatherer

Provide a rough summary of the people involved in the process and their roles.

Assistant

Knowledge Gatherer @ all

Please describe your core role and main tasks in the process.

Assistant

Interview Partner 0 @ Room Service Manager

To begin, could you please describe your core role and main tasks in the process?

Assistant

Room Service Manager @ Interview Partner 0

Okay, so as the Room Service Manager, I'm really the point person for guests who want to order food or drinks from their rooms.

My day usually starts with taking orders over the phone – noting all the details and any special requests they might have. Then...

Figure 1.3: Screenshot of the MLflow [2] user interface, shown is the trace inspector, where all the messages belonging to one trace are collected and can be inspected.

The MLflow user interface, shown in Figure 1.3, provides an intuitive way to inspect the full message trace alongside the associated metadata. Researchers can review the entire exchange for a given session, which facilitates analysis, troubleshooting, and reproducibility.

Additionally, MLflow enables logging of artifacts generated during execution. For example, process models or other outputs produced in the course of a run are stored as associated artifacts, bound to the session trace. This tight coupling of trace and artifacts significantly enhances the transparency and inspectability of experimental results. It allows researchers to link outcomes directly to the context and sequence of agent interactions that produced them, supporting deeper insight and rigorous evaluation of system behavior.