Agentic Data Workflow System

Ahmed Shaheer, Abbas Bukhari

January 20, 2025

Abstract

Data engineering is a critical function for modern organizations, enabling the seamless flow of data from sources to sinks for analytics, reporting, and machine learning. However, building and managing data pipelines is a complex, multi-stage process requiring expertise in tools, configurations, and logical workflows. Traditionally, data engineers rely on manual methods to select tools, namely choosing from a selection of legacy tools, and/or adopting the industry standard without heed to implementation compatibility. This approach is time-consuming and prone to errors because it is based off the limited knowledge of fledgling teams. The process of understanding the compatibility of various sources with tools, implementing their configuration and ensuring they adapt well to the task at hand is indeed cumbersome.

An Agentic data workflow system offers a promising alternative, automating pipeline construction by systematically prompting users for input and leveraging a knowledge-driven approach to select and configure tools for each stage of the pipeline. The configurations are dynamically set based on the functional constraints defined by the user. An end to end pipeline is deployable that is both interconnected with its various components and uses heuristics to select and use only the essential tools. This type of Agentic system would have a human in the loop based approach to determining optimality of pipeline construction based on parameters given by the user and design logic based on ideas of tool compatibility, scalability, functional constraints and cost effectiveness.

1 Introduction

The process of creating a custom pipeline tailored to user requirements is based on a deterministic framework that emphasizes rule-based decision-making.[1] Unlike AI-driven systems, this approach relies on structured inputs provided by users through carefully designed prompts. These prompts guide users to make specific choices, ensuring that the outcomes are wholly deterministic. The process begins with users submitting their requirements through a form, designed as a drop-down menu or questionnaire. The goal is simplicity for the user and ease of understanding. The questionnaire itself would have questions that segue or transition well to other implementations. This initial step acts as the foundation for further customization as a rough framework for the pipeline can thus be established.

Once the inputs are received, an agent is employed to shortlist appropriate tools for each stage of the pipeline. This logic is responsible for identifying the optimal tools and designing an initial pipeline structure based on a rule-based system. The shortlisted tools are then presented to the user, who remains an integral part of the decision-making process. Through a collaborative "human-in-the-loop" approach, users can finalize the selected tools and configure additional settings as needed, though configurations are entirely optional, as untouched configurations will naturally fallback to default values employed by the service. In some cases, supplementary prompts are provided to assist users in making more informed decisions.

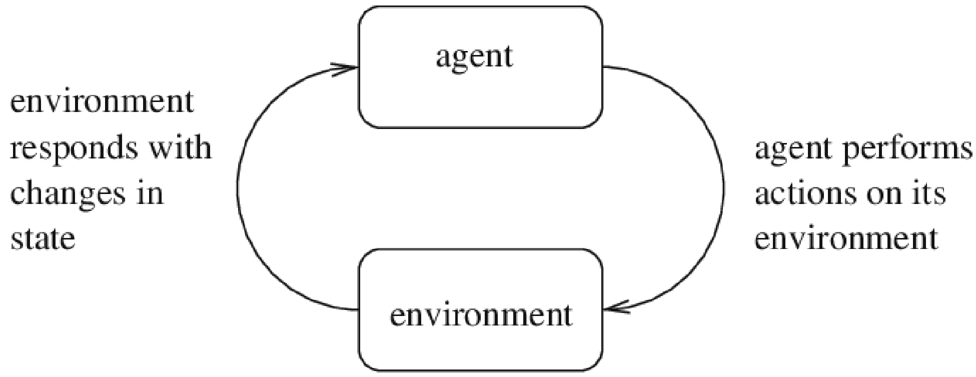

Figure 1: How an Agent Interacts and Perceives a Problem

The tools and services, defined by the user are then deployed using docker containers deployed simultaneously over a docker compose network. All ports used by the containers and services, though not all services, are all exposed, their dependencies are installed and the UI of all the tools are made available on the host's device. The use of docker containers all but assures that the implementation is applicable over a range of operating systems, with varying software and hardware requirements [2]. The docker-compose functionality allows for the creation of a volume that stores data which persists even after the container is taken down; this ensures that the user is in full control of his data. Services that are deployed come packaged with health checks that are implemented when the containers are built so that no faulty service will disturb the user's workflow.

Root privileges are required for certain Linux-based operating systems when deploying Docker containers because Docker operates at the system level and requires access to critical resources to manage containerization effectively. Specifically, Docker interacts directly with the Linux kernel's features, such as namespaces, control groups (cgroups), and storage drivers, to isolate and allocate system resources to containers. These operations inherently demand elevated permissions to ensure security and stability, as they involve modifications to system-level files, network interfaces, and process management. Additionally, the creation and management of Docker volumes, which store persistent data, often necessitate root access to set appropriate ownership and access controls on the host filesystem. Therefore, having sudo privileges ensures that Docker can perform these tasks seamlessly and securely, facilitating the deployment and operation of containers while maintaining the underlying integrity of the host system.

The final step involves testing the functionality of the pipeline and documenting all stages of the process comprehensively. This documentation ensures transparency and allows for future reference and iterative improvements. The structured nature of this workflow guarantees a consistent and user-driven customization process while maintaining a high level of adaptability and precision.

Building data pipelines with aid from an agent involves multiple stages, including selecting the nature of the agent, limiting the scope of the project by short-listing essential industry-based tools, conversing with data engineers about essential elements of design considerations, and determining the optimal way to deploy a reliable connection of service elements. These stages present the following challenges:

1. Selecting agent architecture tailored to project scope and objectives.
2. Addressing disjointed, incompatible industry tools for seamless pipeline integration.
3. Collaborating with engineers to ensure robust, scalable design elements.
4. Establishing reliable, efficient connections for service elements and deployment.

The proof-of-concept developed in this study focuses on demonstrating how deterministic agentic systems can be used to optimize healthcare workflows, such as automating industry-scale pipeline deployment, providing efficient tool/service recommendations, and assisting data engineers in decision-making. Different deterministic agent architectures are explored, including Finite State Machines, and hierarchical approaches (Behavior Tree Model), to determine the most efficient way to manage the increasing complexity and scalability requirements of data ecosystems.

Finally, this paper highlights the key challenges associated with Agentic workflow deployment, particularly in the areas of scalability, agent optimization, and system orchestration. Solutions are proposed to address these issues, focusing on the integration of hierarchical agent structures, adaptive learning algorithms, and resource-efficient scheduling mechanisms. Leveraging edge computing in conjunction with cloud-based orchestration are effective by dint of reducing latency and distributing computational loads effectively.[?] Another proposed solution is the incorporation of predictive analytics to anticipate workflow demands and preemptively allocate resources, ensuring seamless operation under varying workloads.

2 Scenario Agent

The concept of an "agent" as a design methodology predates modern developments in large language models (LLMs), with its origins tracing back to the 1990s. Historically, agents were defined as active entities capable of autonomous action and communication within multi-agent systems, leveraging formal agent communication languages (ACLs). These languages required agents to adhere to standardized protocols, ensuring mutual understanding. While effective in niche applications, the reliance on formal standards limited widespread adoption due to the complexity of implementation and communication constraints.

The advent of LLM-based agents marks a paradigm shift, allowing agents to communicate in natural language without requiring rigid formal standards. This capability is transformative, as it abstracts away the complexities of traditional communication protocols and enables "common-sense reasoning" about novel situations—something that was previously unattainable. Earlier systems were capable of formal reasoning when augmented with tools like Prolog, but they lacked the flexibility to reason about unstructured, real-world scenarios.

From a design perspective, agents diverge significantly from object-oriented programming (OOP) methodologies. While OOP structures are passive, serving as containers of logic that execute commands as dictated by program flow, agents are inherently active entities. They autonomously pursue goals, respond to dynamic environments, and collaborate with other agents to achieve complex objectives. This fundamental distinction differentiates their suitability for dynamic and scalable systems.

In the context of this project, deterministic agent-driven systems build upon this foundational idea by integrating the adaptability and reasoning capabilities of modern agents with a rule-based deterministic framework. By treating pipeline components as agents, these systems enable dynamic interaction and intelligent coordination[?]. For example, wrapping traditional microservices as agents capable of natural language communication creates opportunities for adaptive workflows that were previously infeasible in purely microservice-based architectures. These agentic pipelines ensure flexibility, scalability, and a human-in-the-loop design process, aligning with the project's goal of creating an efficient and user-driven system for data pipeline construction.

2.1 Deterministic Agent

The evolution from automation to AI workflows to AI agents reveals how deterministic and probabilistic elements converge. Automation, built on Boolean logic, excels at predefined, rule-based tasks. AI workflows extend this by incorporating LLMs into deterministic pipelines, enabling more complex, flexible tasks. AI agents go even further, performing non-deterministic, adaptive tasks autonomously, simulating human-like reasoning and behavior.[3]

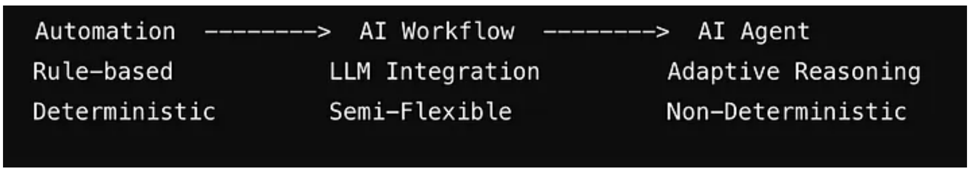

Figure 2: Degree of an Agent's Determinsim

This evolutionary spectrum reflects the growing role of probabilistic systems in addressing complex problems. However, as these systems become more capable, they also become less predictable. This is where deterministic patterns — structured, repeatable approaches to application design — play a critical role in ensuring that AI systems remain reliable and scalable.[4]

2.2 Deterministic Patterns as Scaffolding for Probabilistic Systems

Deterministic software patterns provide essential scaffolding to channel the variability of LLMs productively. They achieve this through several mechanisms:

- **Encapsulation of Probabilistic Behavior** By isolating LLM outputs within deterministic workflows, variability is processed and validated predictably. For instance, an LLM generating recommendations might have its outputs assessed using deterministic scoring mechanisms to ensure alignment with business requirements.

- **Scalability** Deterministic patterns enable the creation of modular, reusable components that simplify development and scaling. This modularity ensures that workflows can handle increasing complexity without sacrificing reliability.

- **Quality Assurance** Deterministic workflows embed guardrails, such as confidence thresholds and post-processing logic, to ensure that LLM outputs meet predefined standards of quality and relevance.

2.3 Types of Deterministic Agents

Deterministic agents operate within predefined frameworks to ensure predictable and repeatable outcomes. Among these, rule-based agents rely on explicitly defined conditions to guide their actions. Each type of agent offers unique advantages and implementation strategies tailored to specific use cases.[?]

Aspect	Deterministic Patterns	Probabilistic Systems (LLMs)
Behavior	Predictable, repeatable	Creative, flexible, variable
Strengths	Reliability, ease of testing	Handles complexity, adapts to new scenarios
Weaknesses	Limited to predefined rules	Potential for unpredictable or undesired outcomes
Use Cases	Input validation, routing logic	Language understanding, reasoning, summarization
Integration Strategy	Serve as guardrails and scaffolding for LLMs	Address nuanced and open-ended tasks

Figure 3: Difference Between Deterministic and LLM based Agents

2.3.1 Rule-Based Agents

Rule-based agents function by executing actions based on a set of predefined rules or conditions[5]. These agents are implemented using logic-based programming or configuration files that define the conditions and corresponding actions. For instance, in a data pipeline, a rule-based agent can validate user inputs by ensuring they meet specified criteria, such as format or completeness, before passing them to subsequent stages. This type of agent excels in structured, well-defined environments where tasks are predictable.

2.3.2 Finite State Machine (FSM) Agents

Finite state machine agents represent workflows as a series of states, transitioning between them based on defined inputs or events. These agents are implemented by mapping each state to specific actions and defining clear transition conditions. For example, an FSM agent in a pipeline might start with input validation, transition to data processing upon successful validation, and then move to data storage. FSM agents are particularly useful for managing sequential workflows with strict order requirements.[6]

2.3.3 Hierarchical Agents

Hierarchical agents operate within a layered structure, where higher-level agents manage or oversee lower-level agents to coordinate complex tasks. Implementations often involve a supervisory agent that delegates specific subtasks to subordinate agents, which execute them independently. In a pipeline context, a hierarchical agent might oversee stages such as data ingestion, transformation, and storage, assigning each stage to specialized agents while ensuring inter-stage coordination. This approach is ideal for large-scale, multi-component systems that require scalability and task division.

2.4 The Role of Deterministic Systems in This Project

This project leverages deterministic systems as the foundation for agent-driven pipeline construction, aligning with the principles outlined above. By integrating agents into a structured framework, the system ensures that variability in tool recommendations, user inputs, and configurations is processed within a repeatable and predictable architecture. The deterministic nature of the system facilitates:

1. Agents recommend tools based on rule-based logic tailored to user-defined constraints, ensuring compatibility and scalability.

2. A human-in-the-loop approach enables users to refine workflows without requiring deep technical expertise, promoting transparency and control.

3. The pipeline is constructed using deterministic logic, enabling seamless integration of tools and services within a Docker-based environment.

An agent, in the context of this project, is a deterministic algorithm designed to automate and optimize workflow tasks by interacting with users and other agents in natural language. Unlike traditional scripts or programs, which execute predefined tasks without flexibility or contextual understanding, agents are equipped with the ability to interpret user inputs in plain English and adapt their actions accordingly. This eliminates the need for users to have technical expertise in coding or infrastructure setup.

Agents excel in dynamic environments by facilitating communication between themselves and responding to feedback to adjust their actions. This capability is not inherently "intelligent" in the sense of autonomous decision-making but is driven by finely-tuned deterministic prompts and logical decision frameworks. For example, in this project, an agent can recommend tools for data pipeline creation, refine those recommendations based on user input, and collaborate with other agents, such as validators, to ensure the pipeline adheres to functional constraints and user-defined requirements.

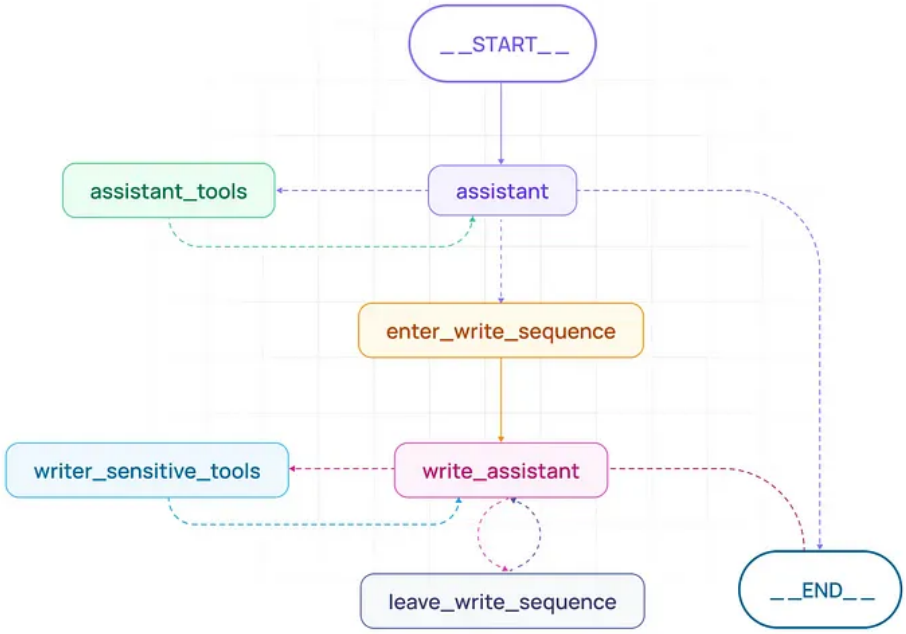

Figure 4: Agent Setup

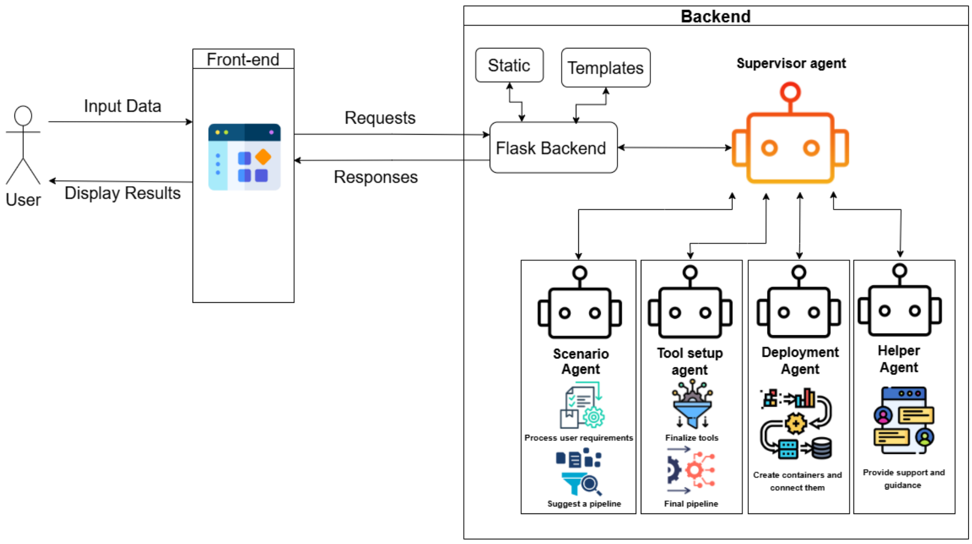

Figure 5: Architecture Diagram

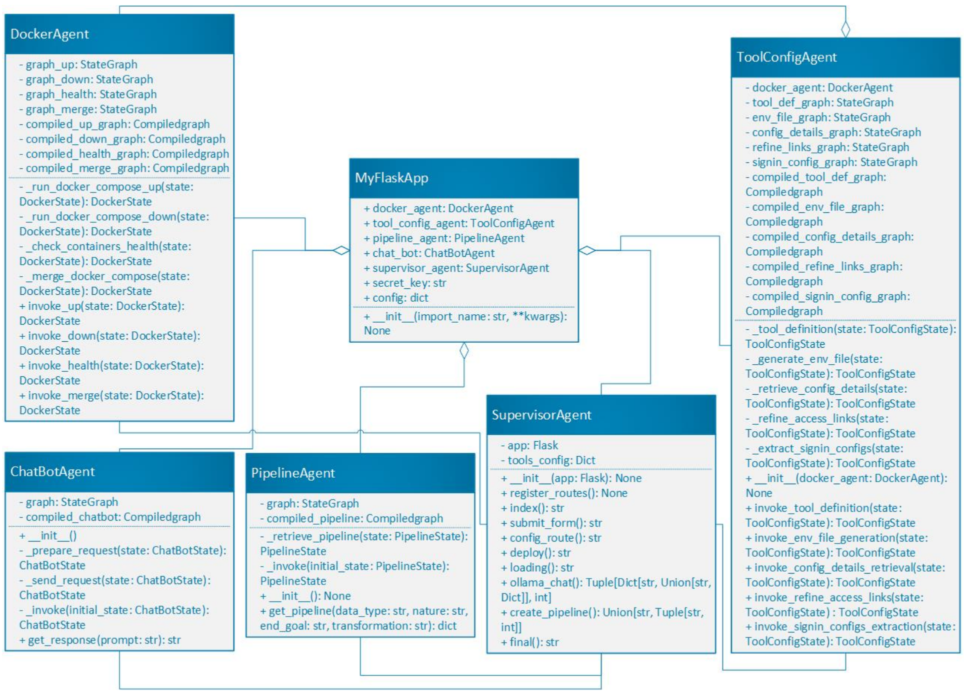

Figure 6: Class Diagram

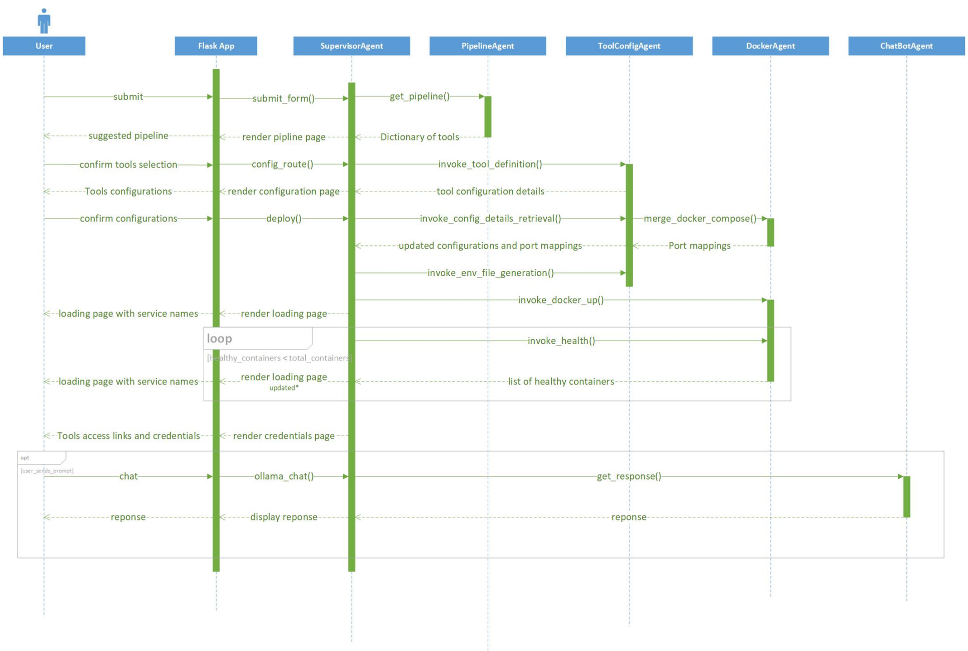

Figure 7: Sequence Diagram

This approach is particularly advantageous when constructing complex workflows, as agents act as guides, simplifying tool selection, configuration, and deployment. They empower users to oversee the process in a human-in-the-loop framework, where decisions are transparent, modifiable, and guided by deterministic rules rather than opaque machine learning processes. By integrating agents into the pipeline creation process, the project ensures a scalable, user-friendly system where users maintain control and oversight without requiring advanced technical skills.

3 Tool and Service Definition

3.1 Containerized Tool Deployment using Docker

Docker is a platform for developing, deploying, and managing applications using containerization. Containers are lightweight, standalone executable units that package an application and its dependencies, ensuring consistent behavior across different environments. Unlike virtual machines, containers share the host operating system's kernel, making them more efficient in terms of performance and resource utilization.[7] Docker provides tools to create, manage, and deploy containers seamlessly, making it a popular choice for modern development workflows.

Docker Compose is a tool specifically designed to manage multi-container applications. It allows developers to define and configure multiple services within a single YAML file, simplifying the orchestration of interconnected containers. [8] By abstracting the complexities of coordinating multiple containers, Docker Compose allows for the replicate entire application environments effortlessly, enhancing collaboration, portability, and scalability.

3.2 Docker Compose in the Context of this Project

Docker Compose enables the definition and coordination of multi-container applications in a single YAML file. This consolidated approach simplifies the orchestration of services, ensuring consistency across environments. For instance, in projects involving agent-driven data pipelines. This ensures that all pipeline components—such as ingestion, processing, and storage services—are interconnected and operational with minimal manual intervention.

Let the following terms be defined:

- T: The set of tools to merge, where $T = \{t_1, t_2, \ldots, t_n\}$.
- S_t: The set of services defined for tool t, — $S_t = \{s_{t1}, s_{t2}, \ldots, s_{tm}\}$.
- V_t: The set of volumes defined for tool t, — $V_t = \{v_{t1}, v_{t2}, \ldots, v_{tp}\}$.
- P_t: The set of ports defined for tool t, — $P_t = \{p_{t1}, p_{t2}, \ldots, p_{tq}\}$.
- $\mathcal{C}$: The merged Docker Compose file structure, where:

$$\mathcal{C} = (\mathcal{S}, \mathcal{V}, \mathcal{P}, \mathcal{N})$$

3.2.1 Initialization

The portability of applications' configuration files allows for collaboration among development, operations, and stakeholder teams. By sharing a consistent configuration, teams can work on identical setups, reducing discrepancies and thus production time. This collaborative efficiency is particularly advantageous in deterministic workflows where precision and consistency are critical.

The merged Compose structure is initialized as:

$$\mathcal{C} = (\mathcal{S} = \emptyset, \mathcal{V} = \emptyset, \mathcal{P} = \emptyset, \mathcal{N} = data\_pipeline\_network).$$

3.2.2 Service Name Conflicts

It also optimizes development by caching container configurations, enabling rapid environment adjustments without recreating unchanged containers. This efficiency supports iterative development processes, such as refining agent-driven pipelines, where quick adjustments and testing cycles are essential. It allows for the user to seamlessly add another tool or service to an existing pipeline.

For any $s_{t_i} \in S_t$, if $\exists s \in \mathcal{S} | s = s_{t_i}$, rename s_{t_i} to s'_{t_i} where:

$$s'_{t_i} = tool \oplus s_{t_i}.$$

3.2.3 Environment Variables

Within its YAML configuration files, it will support all environment variables allowing users to adapt the application for diverse deployment scenarios. For instance, in our agentic system, environment-specific configurations can be defined to accommodate varying computational resources or service requirements across development, testing, and production environments.

For any environment variable e in S_t, define:

$$e = \{(k_1, v_1), (k_2, v_2), \ldots\}$$

where k_i are keys and v_i are corresponding values. If:

$$\exists (k, v) | v \neq v' fork,$$

report a conflict:

$$ConflictVar|k : \{v, v'\}.$$

The algorithm ensures a conflict-free, deterministic merge of Docker Compose files from multiple tools while maintaining compatibility and granularity. The merged structure $\mathcal{C}$ encapsulates all services, volumes, ports, and networks, providing a unified deployment configuration.

4 Walkthrough Guide

4.1 Installation

Dependencies:

- Python 3.8+
- Docker and Docker Compose Plugin
- Install Python dependencies using: `pip install -r requirements.txt`

Set Up:

- Clone the repository.
- Navigate to the project root.

4.2 Running the Application

Launch: Run the application with `python app.py`. **Access:** Open `http://127.0.0.1:5000/` in your browser.

4.3 Prompt Page

The **Prompt Page** collects essential data from users, such as data type, use case, batch or streaming requirements, and transformation needs. The layout is intuitive and beginner-friendly, ensuring ease of use for data engineers.

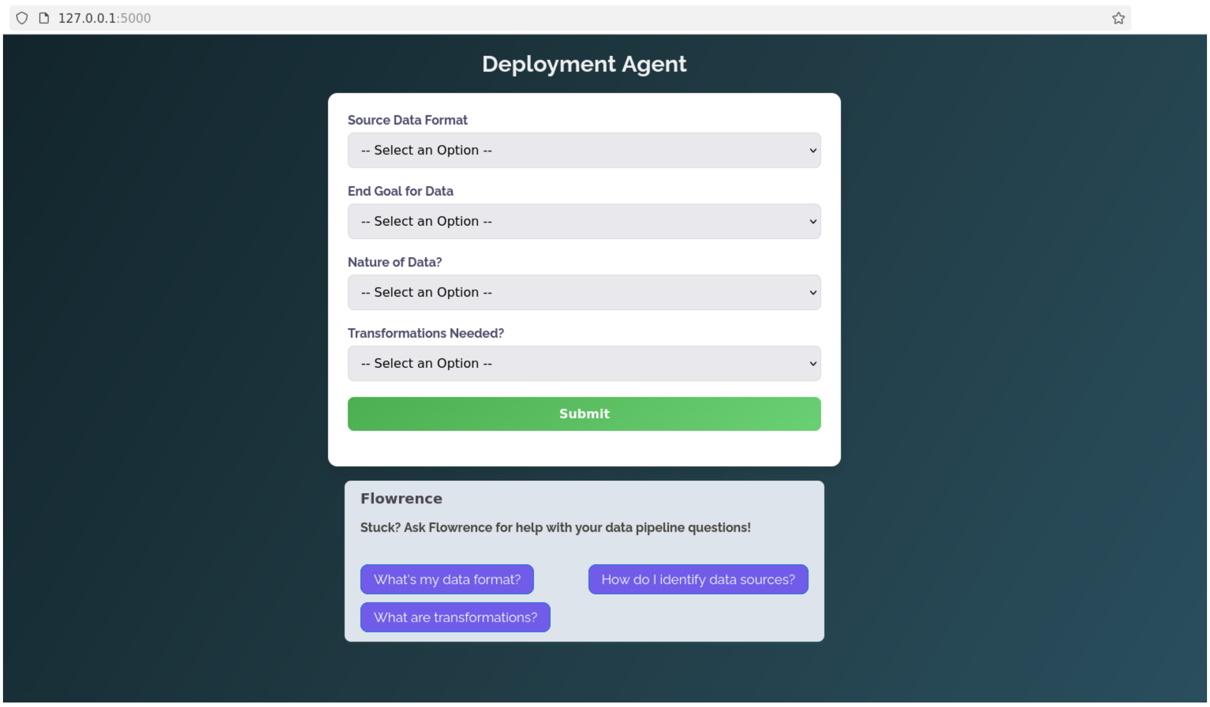

Figure 8: Snapshot of the Prompt Page, where users provide essential details about their data and intended pipeline use.

Below the main input fields, a chatbot is available to assist users, alongside FAQs addressing common questions about data pipelines.

4.4 Tool Selection Page

After submitting the form, users are redirected to the **Tool Selection Page**, which visually represents the recommended pipeline using a flowchart. The pipeline includes icons and flow arrows for ingestion, processing, and storage tools.

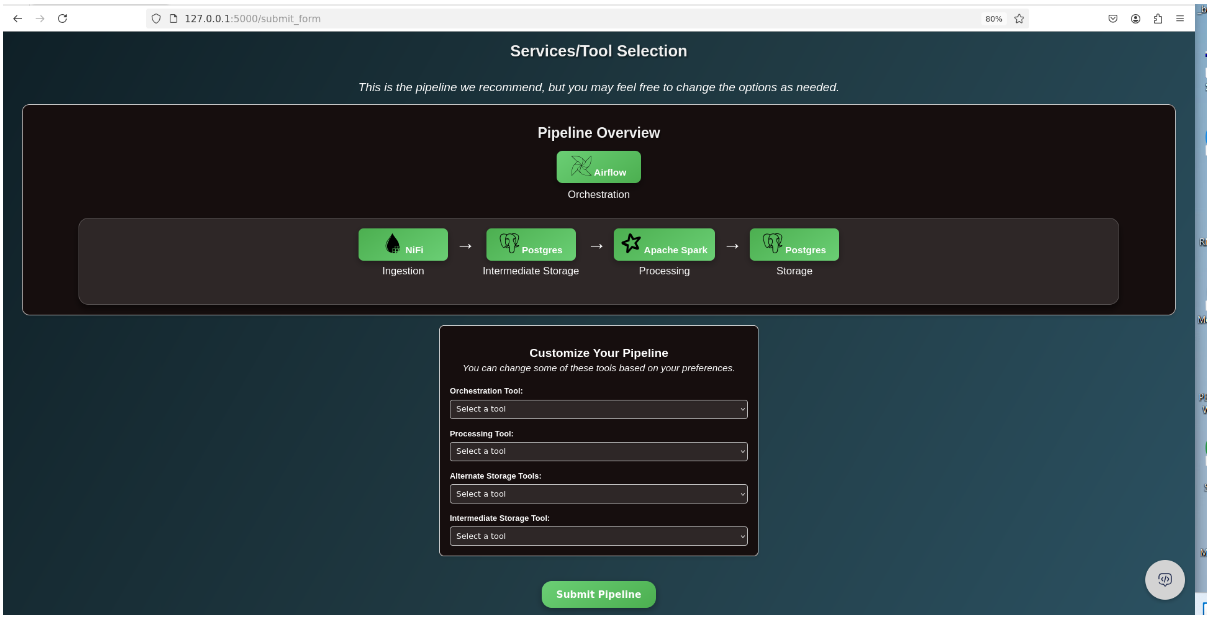

Figure 9: Snapshot of the Tool Selection Page.

Dropdown menus at the bottom allow users to select or configure alternative tools, subject to compatibility. The chatbot provides additional insights about tool recommendations.

4.5 Configurations Page

The **Configurations Page** displays tools the user can configure, keeping advanced options hidden to ensure simplicity. Default values are applied for unmodified configurations, minimizing complexity for beginners.

A chatbot is available to assist users with configuration decisions, ensuring informed choices.

4.6 Access Page

The **Access Page** serves as a central hub for managing deployed pipeline services. It lists service ports, links to user interfaces, and access credentials.

Users can reset the pipeline by clicking the **Home** button, which displays a warning before termination. Resetting redirects users back to the Prompt Page to create a new pipeline.

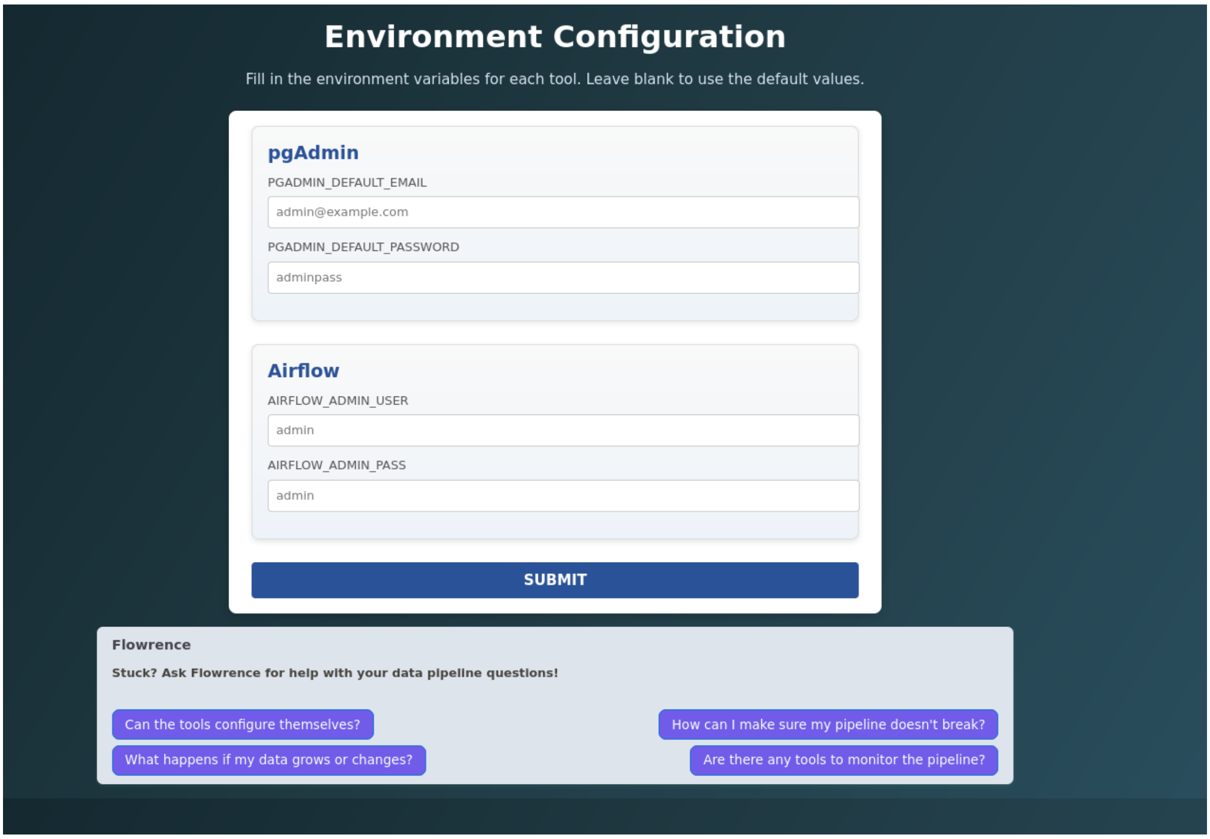

Figure 10: Snapshot of the Configurations Page, showing options for customizing the tools with default values.

4.7 Troubleshooting Guidelines

- **Docker Startup:** Ensure `docker_manager.run_docker_compose()` runs without errors.
- **Health Checks:** Extend `check_containers_health()` for new containers with unique healthcheck needs.
- **Docker Permission:** Ensure you can run Docker commands without errors.
- **Port Conflicts:** Check logs if auto-incremented host ports fail.
- **No Containers:** Verify Docker logs in the console or run `docker ps`.

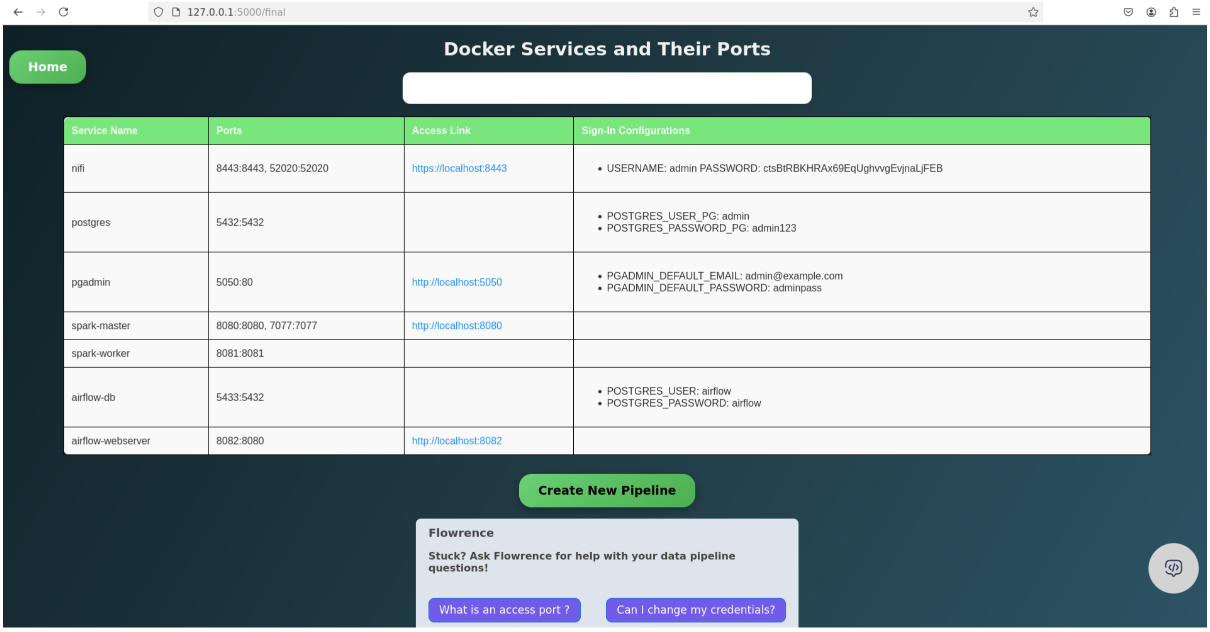

Figure 11: Snapshot of the Access Page, showing running services and access links.

5 Conclusion

The Agentic Data Workflow System represents a significant advancement in the automation of data pipeline construction and deployment. By leveraging deterministic patterns and rule-based decision-making, the system ensures predictable, scalable, and user-friendly solutions for data engineers, regardless of their expertise level. The integration of modular agents capable of interacting dynamically with users not only streamlines tool selection and configuration but also empowers users through a human-in-the-loop framework.

The use of containerization through Docker and Docker Compose guarantees platform independence and efficient resource management, while health checks and conflict resolution mechanisms ensure reliability and robustness.

This proof-of-concept demonstrates the potential of combining deterministic design with modern agentic methodologies, paving the way for future innovations in workflow automation. With further refinements and enhancements, the system can become an indispensable resource for organizations seeking to optimize their data pipeline ecosystems.

References

[1] Russell, S. J., Norvig, P., "Artificial Intelligence: A Modern Approach," 4th ed., Pearson, 2020.

[2] Weiss, G., "Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence," 2nd ed., MIT Press, 2013.

[3] Jennings, N. R., Sycara, K., "Automated Agents for Intelligent Workflow Management," *AI Magazine*, vol. 20, no. 1, pp. 11–27, 1999.

[4] Fowler, M., "Refactoring: Improving the Design of Existing Code," 2nd ed., Addison-Wesley, 2018.

[5] Cheng, B., et al., "Predictive Analytics for Scalable Intelligent Systems," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 51, no. 1, pp. 123–136, 2021.

[6] Harel, D., "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, 1987.

[7] Docker Inc., "Docker Documentation," *Docker*, [Online]. Available: `https://docs.docker.com`. [Accessed: Jan. 19, 2025].

[8] Docker Inc., "Compose Documentation," *Docker Compose*, [Online]. Available: `https://docs.docker.com/compose/`. [Accessed: Jan. 19, 2025].

Appendix

A1 Design Principles

1.1 Single Responsibility Principle (SRP)

Each class is responsible for a distinct concern (e.g., Docker operations, tool configurations, pipeline logic). This ensures the code is easier to maintain, test, and extend.

1.2 Encapsulation

Implementation details (e.g., Docker commands or environment file writing) are hidden within classes. Other parts of the system interact with these classes through defined methods without requiring knowledge of the internal workings.

1.3 Composition over Inheritance

Classes hold references to other classes (e.g., `ToolConfigManager` references `DockerManager`) instead of relying on inheritance. This enhances flexibility and avoids tightly coupled code.

1.4 Layered Architecture

The system is organized into distinct layers:

- **Presentation Layer:** Flask routes/controllers handle user interactions.
- **Application/Logic Layer:** Manages logic through classes like `DockerManager`, `ToolConfigManager`, `PipelineManager`, and `ChatBotManager`.
- **Infrastructure:** Handles configurations (`config.json`, `.env`, Docker Compose).

1.5 PEP8 and Readability

The project adheres to Python's PEP8 style guidelines, ensuring consistent naming conventions, indentation, and code clarity.

A2 Overall Architecture

2.1 Flask Application (Presentation Layer)

Handles HTTP requests, manages user sessions, and serves templates. Routes direct user actions (e.g., `GET /config`, `POST /deploy`) to appropriate manager methods.

2.2 Manager Classes (Application Logic Layer)

- **DockerManager:** Manages Docker operations, including container orchestration and health checks.
- **ToolConfigManager:** Handles tool definitions, environment variable management, and Compose file merging.

- **PipelineManager:** Generates recommended pipeline configurations based on user input.
- **ChatBotManager:** Interacts with an external LLM or inference API to provide responses to user queries.

2.3 Infrastructure

- `config.json`: Stores default environment variable values.
- `Docker Compose Templates`: Define tool-specific Compose configurations.
- `.env`: Generated at runtime to store environment variables.

A3 Components and Their Interactions

3.1 Flask Controller / Main Application

Responsibilities:

- Runs the Flask server (`app.run(debug=True)`).
- Renders templates (e.g., `index.html`, `config.html`).
- Orchestrates user actions by invoking manager methods.

Interactions:

- Instantiates manager classes.
- Calls `PipelineManager.get_pipeline()` to recommend a pipeline.
- Invokes Docker operations via `DockerManager`.
- Uses `ChatBotManager.infer()` for AI-driven user queries.

3.2 DockerManager

Responsibilities:

- Orchestrates containers with `docker-compose up/down`.
- Merges Compose files, resolves port conflicts, and checks container health.

Interactions:

- Coordinates with `ToolConfigManager` for Compose merging.
- Reports container statuses to the front-end.

3.3 ToolConfigManager

Responsibilities:

- Defines tools based on pipeline configuration.
- Generates environment files and resolves tool configurations.
- Builds direct links to containers for user access.

Interactions:

- Reads from `config.json` to define environment variables.
- Calls `DockerManager.merge_docker_compose()` for final Compose setup.

3.4 PipelineManager

Responsibilities:

- Recommends pipeline steps based on user input (e.g., ingestion, storage, processing).

Interactions:

- Generates a pipeline dictionary that is passed to the front-end for user confirmation.

3.5 ChatBotManager

Responsibilities:

- Sends prompts to an external API and retrieves responses.

Interactions:

- Processes user queries and generates AI-driven responses.

A4 Proof-of-Concept Code

chat-bot-agent.py

```python
from typing_extensions import TypedDict
from langgraph.graph import StateGraph, START, END

class PipelineState(TypedDict):
    data_type: str
    nature: str
    end_goal: str
    transformation: str
    pipeline: dict

class PipelineAgent:
    def __init__(self):
        """
        Initializes the PipelineAgent by defining the state and building the graph.
        """
        self.graph = StateGraph(PipelineState)
        self.graph.add_node("RetrievePipeline", self._retrieve_pipeline)
        self.graph.add_edge(START, "RetrievePipeline")
        self.graph.add_edge("RetrievePipeline", END)

        self.compiled_pipeline = self.graph.compile()

    def _retrieve_pipeline(self, state: PipelineState) -> PipelineState:
        """
        Retrieves the pipeline configuration based on the state values.
        """
        pipelines = {
            "batch": {
                "structured": {
                    "storage": {
                        "yes": {
                            'Ingestion': 'NiFi',
                            'Storage': 'Postgres',
                            'Processing': 'Apache Spark',
                            'Intermediate Storage': 'Postgres'
                        },
                        "no": {
                            'Ingestion': 'NiFi',
                            'Storage': 'Postgres'
                        }
                    },
                    "dashboard": {
                        "yes": {
                            'Ingestion': 'NiFi',
                            'Storage': 'Postgres',
                            'Processing': 'Apache Spark',
                            'Intermediate Storage': 'Postgres',
                            'Visualization': 'Apache Superset'
                        },
                        "no": {
                            'Ingestion': 'NiFi',
```

```
                            'Storage': 'Postgres',
                            'Visualization': 'Apache Superset'
                        }
                    },
                    "graph": {
                        "yes": {
                            'Ingestion': 'NiFi',
                            'Storage': 'Neo4j',
                            'Processing': 'Apache Spark',
                            'Intermediate Storage': 'Postgres'
                        },
                        "no": {
                            'Ingestion': 'NiFi',
                            'Storage': 'Neo4j',
                            'Processing': 'Apache Spark',
                            'Intermediate Storage': 'Postgres'
                        }
                    }
                },
                "semi-structured": {
                    "storage": {
                        "yes": {
                            'Ingestion': 'NiFi',
                            'Storage': 'Mongo',
                            'Processing': 'Apache Spark',
                            'Intermediate Storage': 'Mongo'
                        },
                        "no": {
                            'Ingestion': 'NiFi',
                            'Storage': 'Mongo'
                        }
                    },
                    "dashboard": {
                        "yes": {
                            'Ingestion': 'NiFi',
                            'Storage': 'Postgres',
                            'Processing': 'Apache Spark',
                            'Intermediate Storage': 'Mongo',
                            'Visualization': 'Apache Superset'
                        },
                        "no": {
                            'Ingestion': 'NiFi',
                            'Storage': 'Postgres',
                            'Processing': 'Apache Spark',
                            'Intermediate Storage': 'Mongo',
                            'Visualization': 'Apache Superset'
                        }
                    },
                    "graph": {
                        "yes": {
                            'Ingestion': 'NiFi',
                            'Storage': 'Neo4j',
                            'Processing': 'Apache Spark',
                            'Intermediate Storage': 'Mongo'
                        },
                        "no": {
                            'Ingestion': 'NiFi',
                            'Storage': 'Neo4j',
```

```
                            'Processing': 'Apache Spark',
                            'Intermediate Storage': 'Mongo'
                        }
                    }
                },
                "mixture": {
                    "storage": {
                        "yes": {
                            'Ingestion': 'NiFi',
                            'Structured': 'Postgres',
                            'Semi': 'Mongo',
                            'Processing': 'Apache Spark',
                            'Final Structured': 'Postgres',
                            'Final Semi': 'Mongo'
                        },
                        "no": {
                            'Ingestion': 'NiFi',
                            'Structured': 'Postgres',
                            'Semi': 'Mongo'
                        }
                    },
                    "dashboard": {
                        "yes": {
                            'Ingestion': 'NiFi',
                            'Structured': 'Postgres',
                            'Semi': 'Mongo',
                            'Processing': 'Apache Spark',
                            'Storage': 'Postgres',
                            'Visualization': 'Apache Superset'
                        },
                        "no": {
                            'Ingestion': 'NiFi',
                            'Structured': 'Postgres',
                            'Semi': 'Mongo',
                            'Processing': 'Apache Spark',
                            'Storage': 'Postgres',
                            'Visualization': 'Apache Superset'
                        }
                    },
                    "graph": {
                        "yes": {
                            'Ingestion': 'NiFi',
                            'Structured': 'Postgres',
                            'Semi': 'Mongo',
                            'Processing': 'Apache Spark',
                            'Storage': 'Neo4j'
                        },
                        "no": {
                            'Ingestion': 'NiFi',
                            'Structured': 'Postgres',
                            'Semi': 'Mongo',
                            'Processing': 'Apache Spark',
                            'Storage': 'Neo4j'
                        }
                    }
                }
            },
            "streaming": {
```

```
                "structured": {
                    "storage": {
                        "yes": {
                            'Ingestion': 'Apache Kafka',
                            'Processing': 'Apache Flink',
                            'Storage': 'Postgres'
                        },
                        "no": {
                            'Ingestion': 'Apache Kafka',
                            'Storage': 'Postgres'
                        }
                    },
                    "dashboard": {
                        "yes": {
                            'Ingestion': 'Apache Kafka',
                            'Processing': 'Apache Flink',
                            'Storage': 'Postgres',
                            'Visualization': 'Apache Superset'
                        },
                        "no": {
                            'Ingestion': 'Apache Kafka',
                            'Storage': 'Postgres',
                            'Visualization': 'Apache Superset'
                        }
                    },
                    "graph": {
                        "yes": {
                            'Ingestion': 'Apache Kafka',
                            'Storage': 'Neo4j',
                            'Processing': 'Apache Flink'
                        },
                        "no": {
                            'Ingestion': 'Apache Kafka',
                            'Storage': 'Neo4j',
                            'Processing': 'Apache Flink'
                        }
                    }
                },
                "semi-structured": {
                    "storage": {
                        "yes": {
                            'Ingestion': 'Apache Kafka',
                            'Processing': 'Apache Flink',
                            'Storage': 'Mongo'
                        },
                        "no": {
                            'Ingestion': 'Apache Kafka',
                            'Storage': 'Mongo'
                        }
                    },
                    "dashboard": {
                        "yes": {
                            'Ingestion': 'Apache Kafka',
                            'Processing': 'Apache Flink',
                            'Storage': 'Postgres',
                            'Visualization': 'Apache Superset'
                        },
                        "no": {
```

```
                            'Ingestion': 'Apache Kafka',
                            'Processing': 'Apache Flink',
                            'Storage': 'Postgres',
                            'Visualization': 'Apache Superset'
                        }
                    },
                    "graph": {
                        "yes": {
                            'Ingestion': 'Apache Kafka',
                            'Storage': 'Neo4j',
                            'Processing': 'Apache Flink'
                        },
                        "no": {
                            'Ingestion': 'Apache Kafka',
                            'Storage': 'Neo4j',
                            'Processing': 'Apache Flink'
                        }
                    }
                },
                "mixture": {
                    "storage": {
                        "yes": {
                            'Ingestion': 'Apache Kafka',
                            'Processing': 'Apache Flink',
                            'Structured': 'Postgres',
                            'Semi': 'Mongo',
                            'Final Structured': 'Postgres',
                            'Final Semi': 'Mongo'
                        },
                        "no": {
                            'Ingestion': 'Apache Kafka',
                            'Structured': 'Postgres',
                            'Semi': 'Mongo'
                        }
                    },
                    "dashboard": {
                        "yes": {
                            'Ingestion': 'Apache Kafka',
                            'Processing': 'Apache Flink',
                            'Storage': 'Postgres',
                            'Visualization': 'Apache Superset'
                        },
                        "no": {
                            'Ingestion': 'Apache Kafka',
                            'Processing': 'Apache Flink',
                            'Storage': 'Postgres',
                            'Visualization': 'Apache Superset'
                        }
                    },
                    "graph": {
                        "yes": {
                            'Ingestion': 'Apache Kafka',
                            'Processing': 'Apache Flink',
                            'Structured': 'Postgres',
                            'Semi': 'Mongo',
                            'Storage': 'Neo4j'
                        },
                        "no": {
```

```python
                            'Ingestion': 'Apache Kafka',
                            'Structured': 'Postgres',
                            'Semi': 'Mongo',
                            'Storage': 'Neo4j',
                            'Processing': 'Apache Flink'
                        }
                    }
                }
            }
        }
        # Retrieve the pipeline configuration
        pipe = pipelines[state['nature']][state['data_type']][state['end_goal']][state['transformation']]

        # Add alternate tools for flexibility
        if pipe.get("Semi") == "Mongo" or pipe.get("Structured") == "Postgres":
            pipe.update({
                'Alternate Structured Storage Tools': ["MySQL", "Hadoop Standalone"],
                'Alternate Semi-Structured Storage Tools': ["Cassandra", "Hadoop Standalone"]
            })
        if pipe.get("Storage") == "Postgres":
            pipe.update({'Alternate Final Storage Tools': ["MySQL", "Hadoop Standalone"]})
        if pipe.get("Storage") == "Mongo":
            pipe.update({'Alternate Final Storage Tools': ["Cassandra", "Hadoop Standalone"]})
        if pipe.get("Final Structured") == "Postgres":
            pipe.update({'Alternate Final Structured Storage Tools': ["MySQL", "Hadoop Standalone"]})
        if pipe.get("Final Semi") == "Mongo":
            pipe.update({'Alternate Final Semi Storage Tools': ["Cassandra", "Hadoop Standalone"]})
        if pipe.get('Intermediate Storage') == "Postgres":
            pipe.update({'Alternate Intermediate Storage Tools': ["MySQL", "Hadoop Standalone"]})
        elif pipe.get("Intermediate Storage") == "Mongo":
            pipe.update({'Alternate Intermediate Storage Tools': ["Cassandra", "Hadoop Standalone"]})
        if pipe.get('Processing') == "Apache Flink":
            pipe.update({'Alternate Processing Tools': ["Apache Spark"]})
        if pipe.get('Processing') == "Apache Spark":
            pipe.update({'Alternate Processing Tools': ["Apache Flink"]})

        # Add orchestration options
        pipe.update({'Orchestration': 'Airflow'})
        pipe.update({'Alternate Orchestration': ['Prefect']})

        state['pipeline'] = pipe
        return state

    def _invoke(self, initial_state: PipelineState) -> PipelineState:
        """
        Invokes the compiled pipeline graph with the given initial
```

```
state.

        Args:
            initial_state (PipelineState): The initial state to pass through the graph.

        Returns:
            PipelineState: The final state after processing.
        """
        return self.compiled_pipeline.invoke(initial_state)

    def get_pipeline(self, data_type: str, nature: str, end_goal: str, transformation: str) -> dict:
        """
        Generates a pipeline configuration based on the given parameters.

        Args:
            data_type (str): Type of data to process.
            nature (str): Nature of the pipeline (batch/streaming).
            end_goal (str): Purpose of the pipeline (storage/dashboard/graph).
            transformation (str): Whether transformations are required.

        Returns:
            dict: The generated pipeline configuration.
        """
        initial_state = PipelineState(
            data_type=data_type,
            nature=nature,
            end_goal=end_goal,
            transformation=transformation,
            pipeline={}
        )
        result = self._invoke(initial_state)
        return result['pipeline']

# Example usage
if __name__ == "__main__":
    pipeline_agent = PipelineAgent()
    pipeline = pipeline_agent.get_pipeline("structured", "batch", "dashboard", "yes")
    print(pipeline)
```

Listing 1: The Backend functionality that allows user to communicate with chatbot Agent

docker-agent.py

```
from typing_extensions import TypedDict
from langgraph.graph import StateGraph, START, END
import subprocess
import json
import os
import shutil
import yaml
from typing import List, Dict, Tuple, Union

```

```python
class DockerState(TypedDict):
    command: str
    tool_names: List[str]
    base_directory: str
    ports: Dict[str, List[str]]
    compose_file_path: str
    success: bool
    error: str

class DockerAgent:
    """
    Handles Docker commands (up/down) and merges Docker Compose files.
    """

    def __init__(self) -> None:
        """
        Initializes the DockerAgent by defining the state and building
    the graph.
        """
        self.graph_up = StateGraph(DockerState)
        self.graph_down = StateGraph(DockerState)
        self.graph_health = StateGraph(DockerState)
        self.graph_merge = StateGraph(DockerState)

        # Define nodes for `docker compose up`
        self.graph_up.add_node("RunDockerComposeUp", self.
    _run_docker_compose_up)
        self.graph_up.add_edge(START, "RunDockerComposeUp")
        self.graph_up.add_edge("RunDockerComposeUp", END)
        self.compiled_up_graph = self.graph_up.compile()

        # Define nodes for `docker compose down`
        self.graph_down.add_node("RunDockerComposeDown", self.
    _run_docker_compose_down)
        self.graph_down.add_edge(START, "RunDockerComposeDown")
        self.graph_down.add_edge("RunDockerComposeDown", END)
        self.compiled_down_graph = self.graph_down.compile()

        # Define nodes for checking container health
        self.graph_health.add_node("CheckContainersHealth", self.
    _check_containers_health)
        self.graph_health.add_edge(START, "CheckContainersHealth")
        self.graph_health.add_edge("CheckContainersHealth", END)
        self.compiled_health_graph = self.graph_health.compile()

        # Define nodes for merging Docker Compose files
        self.graph_merge.add_node("MergeDockerCompose", self.
    _merge_docker_compose)
        self.graph_merge.add_edge(START, "MergeDockerCompose")
        self.graph_merge.add_edge("MergeDockerCompose", END)
        self.compiled_merge_graph = self.graph_merge.compile()

    def _run_docker_compose_up(self, state: DockerState) -> DockerState
    :
        """
        Runs the `docker compose up` command to build and start all
    services.
        """
```

```
        try:
            command = "docker compose up --build"
            result = subprocess.run(f"sudo {command}", shell=True,
    check=True)
            state["success"] = True
            state["error"] = ""
        except subprocess.CalledProcessError as e:
            state["success"] = False
            state["error"] = str(e)
        return state

    def _run_docker_compose_down(self, state: DockerState) ->
    DockerState:
        """
        Runs the `docker compose down` command to stop and remove
    containers, networks, and volumes.
        """
        try:
            command = "docker compose down -v"
            result = subprocess.run(f"sudo {command}", shell=True,
    check=True)
            state["success"] = True
            state["error"] = ""
        except subprocess.CalledProcessError as e:
            state["success"] = False
            state["error"] = str(e)
        return state

    def _check_containers_health(self, state: DockerState) ->
    DockerState:
        """
        Checks the health status of all running Docker containers.
        """
        ps_cmd = ["sudo", "docker", "ps", "-q"]
        try:
            result = subprocess.run(ps_cmd, capture_output=True, text=
    True, check=True)
            container_ids = result.stdout.strip().split()
            if not container_ids:
                state["success"] = False
                state["ports"] = {}
                return state

            healthy_containers = []
            for container_id in container_ids:
                inspect_cmd = ["sudo", "docker", "inspect",
    container_id]
                try:
                    inspect_result = subprocess.run(
                        inspect_cmd, capture_output=True, text=True,
    check=True
                    )
                    data = json.loads(inspect_result.stdout)
                    container_info = data[0] if data else {}
                    container_name = container_info.get("Name", "").
    lstrip("/")
                    state_info = container_info.get("State", {})
                    health_status = state_info.get("Health", {}).get("
```

```
Status")
                if health_status == "healthy":
                    healthy_containers.append(container_name)
            except (json.JSONDecodeError, subprocess.CalledProcessError):
                continue

        state["success"] = len(healthy_containers) == len(container_ids)
        state["ports"] = healthy_containers
    except subprocess.CalledProcessError as e:
        state["success"] = False
        state["error"] = str(e)
    return state


def _merge_docker_compose(self, state: DockerState) -> DockerState:
    """
    Merges Docker Compose files from multiple tools into a single Compose file.

    """
    # Initialize the merged Compose structure
    try:
        merged_compose = {
            'version': '3.9',
            'services': {},
            'networks': {
                'data_pipeline_network': {}
            },
            'volumes': {}
        }

        allocated_ports = {}
        port_assignments = {}
        temp_merged_volumes = "temp_merged_volumes"
        os.makedirs(temp_merged_volumes, exist_ok=True)
        all_service_envs = {}
        used_service_names = set()

        for tool_name in state["tool_names"]:
            tool_dir = os.path.join(state["base_directory"], tool_name.lower())
            compose_file_path = os.path.join(tool_dir, "docker-compose.yml")

            # --- 1. Read the Tool's docker-compose.yml ---
            if not os.path.exists(compose_file_path):
                print(f"Warning: Compose file not found for tool '{tool_name}': {compose_file_path}")
                continue

            with open(compose_file_path, 'r') as f:
                try:
                    tool_compose = yaml.safe_load(f)
                except yaml.YAMLError as e:
                    print(f"Error parsing YAML for tool '{tool_name}': {e}")
```

```
                    continue

            if not tool_compose or 'services' not in tool_compose:
                print(f"Warning: No services defined in {compose_file_path}")
                continue

            # --- 2. Check if the tool has a custom Dockerfile ---
            has_dockerfile = os.path.exists(os.path.join(tool_dir, "Dockerfile"))

            # --- 3. Merge Services ---
            for service_name, service_config in tool_compose['services'].items():
                if service_name in used_service_names:
                    new_service_name = f"{tool_name.lower()}_{service_name}"
                    print(f"Service name conflict detected for '{service_name}'. "
                          f"Renaming to '{new_service_name}'")
                else:
                    new_service_name = service_name

                used_service_names.add(new_service_name)
                merged_compose['services'][new_service_name] = service_config.copy()

                # Handle custom build or image references
                if has_dockerfile and 'build' in service_config:
                    build_value = service_config['build']
                    if isinstance(build_value, dict):
                        merged_compose['services'][new_service_name]['build']['context'] = tool_dir
                    elif isinstance(build_value, str):
                        merged_compose['services'][new_service_name]['build'] = {
                            'context': os.path.join(tool_dir, build_value)
                        }
                    else:
                        print(f"Warning: 'build' format for '{new_service_name}' is unrecognized.")
                else:
                    if 'image' not in service_config:
                        print(f"Warning: Service '{new_service_name}' has no 'image' or 'build' definition.")

                # --- 4. Handle Volume Mounts ---
                if 'volumes' in service_config:
                    new_volume_list = []
                    for vol in service_config['volumes']:
                        if isinstance(vol, str):
                            parts = vol.split(':')
                            if len(parts) == 2:
                                source, dest = parts
                                if source.startswith('./') or os.path.isdir(os.path.join(tool_dir, source)):
                                    abs_source_path = os.path.join(
```

```
tool_dir, source)
                                    merged_volume_dir = os.path.join(
                                        temp_merged_volumes,
                                        f"{tool_name.lower()}_{service_name}_{os.path.basename(source.strip('./'))}"
                                    )
                                    if os.path.exists(abs_source_path):
                                        if os.path.isdir(abs_source_path):
                                            try:
                                                shutil.copytree(abs_source_path, merged_volume_dir, dirs_exist_ok=True)
                                            except OSError:
                                                print(f"Skipping copy for {abs_source_path}: Directory not found.")
                                        else:
                                            if os.path.isfile(abs_source_path):
                                                os.makedirs(merged_volume_dir, exist_ok=True)
                                                shutil.copy2(abs_source_path, merged_volume_dir)
                                            else:
                                                print(f"Skipping {abs_source_path}, not found.")
                                    new_volume_list.append(f"{os.path.abspath(merged_volume_dir)}:{dest}")
                                else:
                                    # Named volume, prefix with tool+service
                                    volume_name = f"{tool_name.lower()}_{service_name}_{source}"
                                    merged_compose['volumes'][volume_name] = {}
                                    new_volume_list.append(f"{volume_name}:{dest}")
                            else:
                                new_volume_list.append(vol)
                        elif isinstance(vol, dict):
                            for k, v in vol.items():
                                volume_name = f"{tool_name.lower()}_{service_name}_{k}"
                                merged_compose['volumes'][volume_name] = {}
                                new_volume_list.append({volume_name: v})
                        else:
                            new_volume_list.append(vol)

                    merged_compose['services'][new_service_name]['volumes'] = new_volume_list

                # Add the service to the shared network
                merged_compose['services'][new_service_name]['networks'] = ['data_pipeline_network']

```

```
                # --- 5. Handle Port Conflicts ---
                if 'ports' in service_config:
                    updated_ports = []
                    for port_mapping in service_config['ports']:
                        host_port = None
                        container_port = None

                        if isinstance(port_mapping, str):
                            parts = port_mapping.split(':')
                            if len(parts) == 2:
                                host_port_str, container_port_str = parts
                                host_port = int(host_port_str)
                                container_port = int(container_port_str)
                        elif isinstance(port_mapping, int):
                            host_port = port_mapping
                            container_port = port_mapping
                        elif (isinstance(port_mapping, dict)
                            and 'published' in port_mapping
                            and 'target' in port_mapping):
                            host_port = int(port_mapping['published'])
                            container_port = int(port_mapping['target'])

                        if host_port is not None:
                            while host_port in allocated_ports:
                                host_port += 1  # increment until free

                            allocated_ports[host_port] = new_service_name

                            if isinstance(port_mapping, str):
                                updated_ports.append(f"{host_port}:{container_port}")
                                port_assignments.setdefault(new_service_name, []).append(f"{host_port}:{container_port}")
                            elif isinstance(port_mapping, int):
                                updated_ports.append(host_port)
                                port_assignments.setdefault(new_service_name, []).append(str(host_port))
                            elif isinstance(port_mapping, dict):
                                new_port_mapping = {
                                    'published': host_port,
                                    'target': container_port
                                }
                                for extra_key in ['protocol', 'mode']:
                                    if extra_key in port_mapping:
                                        new_port_mapping[extra_key] = port_mapping[extra_key]

                                updated_ports.append(new_port_mapping)
                                port_assignments.setdefault(new_service_name, []).append(f"{host_port}:{container_port}")
```

```
                    merged_compose['services'][new_service_name]['ports'] = updated_ports

                # --- 6. Handle depends_on references ---
                if 'depends_on' in service_config:
                    depends_config = service_config['depends_on']
                    if isinstance(depends_config, dict):
                        new_depends = {}
                        for dep_service, dep_config in depends_config.items():
                            if dep_service in used_service_names:
                                new_depends[dep_service] = dep_config
                            else:
                                new_depends[dep_service] = dep_config
                        merged_compose['services'][new_service_name]['depends_on'] = new_depends
                    elif isinstance(depends_config, list):
                        new_depends_list = []
                        for dep_service in depends_config:
                            if dep_service in used_service_names:
                                new_depends_list.append(dep_service)
                            else:
                                new_depends_list.append(dep_service)
                        merged_compose['services'][new_service_name]['depends_on'] = new_depends_list

                # --- 7. Collect environment variables ---
                env_vars = merged_compose['services'][new_service_name].get('environment', {})
                if isinstance(env_vars, list):
                    env_dict = {}
                    for env_item in env_vars:
                        if '=' in env_item:
                            k, v = env_item.split('=', 1)
                            env_dict[k] = v
                    env_vars = env_dict

                if not isinstance(env_vars, dict):
                    env_vars = {}

                all_service_envs[new_service_name] = env_vars

        # Detect environment variable conflicts
        env_conflicts = {}
        for service_name, envs in all_service_envs.items():
            for k, v in envs.items():
                env_conflicts.setdefault(k, {}).setdefault(v, []).append(service_name)

        for env_var, values_dict in env_conflicts.items():
            if len(values_dict) > 1:
                print(f"Warning: Conflict detected for ENV variable '{env_var}':")
```

```python
                    for val, services in values_dict.items():
                        print(f"  Value '{val}' set by services: {services}")

            # Write the merged Docker Compose file
            with open("docker-compose.yml", 'w') as outfile:
                yaml.dump(merged_compose, outfile, sort_keys=False, indent=2)

            state["success"] = True
            state["ports"] = port_assignments
        except Exception as e:
            state["success"] = False
            state["error"] = str(e)
        return state



    # PUBLIC METHODS TO INVOKE GRAPHS
    def invoke_up(self, state: DockerState) -> DockerState:
        return self.compiled_up_graph.invoke(state)

    def invoke_down(self, state: DockerState) -> DockerState:
        return self.compiled_down_graph.invoke(state)

    def invoke_health(self, state: DockerState) -> DockerState:
        return self.compiled_health_graph.invoke(state)

    def invoke_merge(self, state: DockerState) -> DockerState:
        return self.compiled_merge_graph.invoke(state)

    # Example usage
if __name__ == "__main__":
    docker_agent = DockerAgent()
    initial_state = DockerState(
        command="",
        tool_names=[],
        base_directory="docker_templates",
        ports={},
        compose_file_path="",
        success=False,
        error=""
    )
    result = docker_agent.invoke_up(initial_state)
    print(result)
```

Listing 2: An Agent that quickly reads pipeline configuration details submitted by user and launches all relevant services while also avoiding conflicts and ensuring service health

tool-config-agent.py

```python
from typing_extensions import TypedDict
from langgraph.graph import StateGraph, START, END
from typing import Dict, List, Tuple, Union
import os

class ToolConfigState(TypedDict):
```

```
    pipeline_dict: Dict[str, Union[str, List[str]]]
    tools_config: Dict[str, Dict]
    config: Dict[str, Dict]
    tool_names: List[str]
    ports: Dict[str, List[str]]
    services_dict: Dict[str, Dict]
    env_file_path: str
    updated_config: Dict[str, Dict]

class ToolConfigAgent:
    """
    Manages tool definitions, environment files, and other configurations.
    """

    def __init__(self, docker_agent) -> None:
        """
        Initializes the ToolConfigAgent with a DockerAgent instance.

        Args:
            docker_agent (DockerAgent): An instance of DockerAgent.
        """
        self.docker_agent = docker_agent
        self.tool_dependencies = {
            "airflow-webserver": ["airflow-db", "airflow-init", "airflow-scheduler", "airflow-webserver"],
            "jobmanager": ["jobmanager", "taskmanager"],
            "zoo1": [ "zoo1", "kafka1", "kafka-schema-registry", "kafka-rest-proxy", "kafka-connect"],
            "spark-master": ["spark-master", "spark-worker"],
            "superset": ["superset-metadata-db", "superset"],
            "namenode": ["namenode", "datanode"],
            "mongo-express": ["mongo", "mongo-express"],
            "phpmyadmin": ["mysql", "phpmyadmin"],
            "neo4j": ["neo4j"],
            "nifi": ["nifi"],
            "pgadmin": ["postgres", "pgadmin"],
            "prefect-orion": ["prefect-orion", "prefect-worker"],
            "cassandra" : ["cassandra"],
            "conduktorDB" : ["conduktorDB", "conduktor-console"]
        }
        # Graph for tool definition
        self.tool_def_graph = StateGraph(ToolConfigState)
        self.tool_def_graph.add_node("DefineTools", self._tool_definition)
        self.tool_def_graph.add_edge(START, "DefineTools")
        self.tool_def_graph.add_edge("DefineTools", END)
        self.compiled_tool_def_graph = self.tool_def_graph.compile()

        # Graph for generating environment files
        self.env_file_graph = StateGraph(ToolConfigState)
        self.env_file_graph.add_node("GenerateEnv", self._generate_env_file)
        self.env_file_graph.add_edge(START, "GenerateEnv")
        self.env_file_graph.add_edge("GenerateEnv", END)
        self.compiled_env_file_graph = self.env_file_graph.compile()

        # Graph for retrieving configuration details
```

```
        self.config_details_graph = StateGraph(ToolConfigState)
        self.config_details_graph.add_node("RetrieveConfig", self._retrieve_config_details)
        self.config_details_graph.add_edge(START, "RetrieveConfig")
        self.config_details_graph.add_edge("RetrieveConfig", END)
        self.compiled_config_details_graph = self.config_details_graph.compile()

        # Graph for refining access links
        self.refine_links_graph = StateGraph(ToolConfigState)
        self.refine_links_graph.add_node("RefineLinks", self._refine_access_links)
        self.refine_links_graph.add_edge(START, "RefineLinks")
        self.refine_links_graph.add_edge("RefineLinks", END)
        self.compiled_refine_links_graph = self.refine_links_graph.compile()

        # Graph for extracting sign-in configurations
        self.signin_config_graph = StateGraph(ToolConfigState)
        self.signin_config_graph.add_node("ExtractSigninConfigs", self._extract_signin_configs)
        self.signin_config_graph.add_edge(START, "ExtractSigninConfigs")
        self.signin_config_graph.add_edge("ExtractSigninConfigs", END)
        self.compiled_signin_config_graph = self.signin_config_graph.compile()

        # Graph for extracting sign-in configurations
        self.all_services_graph = StateGraph(ToolConfigState)
        self.all_services_graph.add_node("ExtractAllDependencies", self._all_services)
        self.all_services_graph.add_edge(START, "ExtractAllDependencies")
        self.all_services_graph.add_edge("ExtractAllDependencies", END)
        self.compiled_all_services_graph = self.all_services_graph.compile()

    # NODE IMPLEMENTATIONS
    def _tool_definition(self, state: ToolConfigState) -> ToolConfigState:
        selected_tools = {key: value for key, value in state["pipeline_dict"].items() if value}
        shortlisted_tools = {}
        for type_object, tool in selected_tools.items():
            if isinstance(tool, list):
                pass
            else:
                tool = tool.replace(" ", "")
                shortlisted_tools.update({tool: state["tools_config"][tool]})
        state["updated_config"] = shortlisted_tools
        return state

    def _generate_env_file(self, state: ToolConfigState) -> ToolConfigState:
        with open(state["env_file_path"], "w") as file:
            for service, details in state["updated_config"].items():
                if "EnvironmentVariables" in details:
```

```
                    file.write(f"# {service} Environment Variables\n")
                    for key, value in details["EnvironmentVariables"].items():
                        file.write(f"{key}={value}\n")
                    file.write("\n")
        return state

    def _retrieve_config_details(self, state: ToolConfigState) -> ToolConfigState:
        tool_names = state["tool_names"]
        updated_config = {}

        for tool_name in tool_names:
            if tool_name in state["tools_config"]:
                tool_config = state["tools_config"][tool_name]
                updated_env_vars = {}

                for var_name, default_value in tool_config["EnvironmentVariables"].items():
                    input_value = None  # Placeholder for form data
                    updated_env_vars[var_name] = input_value or default_value

                updated_config[tool_name] = {"EnvironmentVariables": updated_env_vars}

        docker_state = {
            "command": "",
            "tool_names": tool_names,
            "base_directory": "docker_templates",
            "ports": {},
            "compose_file_path": "docker-compose.yml",
            "success": False,
            "error": "",
        }

        docker_state = self.docker_agent.invoke_merge(docker_state)

        if not docker_state["success"]:
            state["updated_config"] = {}
            state["ports"] = {}
            raise RuntimeError(f"Error merging Docker Compose files: {docker_state['error']}")
        print(f'These are the updated configs')
        state["updated_config"] = updated_config
        state["ports"] = docker_state["ports"]
        return state

    def _refine_access_links(self, state: ToolConfigState) -> ToolConfigState:
        services_with_ports = [
            "airflow-webserver", "jobmanager", "conduktor-console", "spark-master",
            "superset", "namenode", "mongo-express", "phpmyadmin",
            "neo4j", "nifi", "pgadmin", "prefect-orion",
        ]
        extracted_services = {}
        for service in services_with_ports:
```

```python
            if service in state["ports"]:
                if service == "nifi":
                    extracted_services[service] = state["ports"][service][1].split(":")[0]
                else:
                    extracted_services[service] = state["ports"][service][0].split(":")[0]
        state["updated_config"]["access_links"] = extracted_services
        print(f'these are the extracted services --------------------{extracted_services}')
        return state

    def _extract_signin_configs(self, state: ToolConfigState) -> ToolConfigState:
        env_params_dict = {
            'airflow-db': ['POSTGRES_USER', 'POSTGRES_PASSWORD'],
            'superset-metadata-db': ['POSTGRES_USER_SUPERSET', 'POSTGRES_PASSWORD_SUPERSET'],
            'airflow-webserver': ['AIRFLOW_ADMIN_USER', 'AIRFLOW_ADMIN_PASS'],
            'mongo': ['MONGO_INITDB_ROOT_USERNAME', 'MONGO_INITDB_ROOT_PASSWORD'],
            'mongo-express': ['ME_CONFIG_BASICAUTH_USERNAME', 'ME_CONFIG_BASICAUTH_PASSWORD'],
            'mysql': ['MYSQL_USER', 'MYSQL_PASSWORD'],
            'phpmyadmin': ['PMA_USER', 'PMA_PASSWORD'],
            'neo4j': ['NEO4J_AUTH'],
            'postgres': ['POSTGRES_USER_PG', 'POSTGRES_PASSWORD_PG'],
            'pgadmin': ['PGADMIN_DEFAULT_EMAIL', 'PGADMIN_DEFAULT_PASSWORD']
        }
        signin_configs = {}
        if not os.path.exists(state["env_file_path"]):
            return state

        env_dict = {}
        with open(state["env_file_path"], "r") as file:
            for line in file:
                line = line.strip()
                if not line or line.startswith("#"):
                    continue
                key, value = line.split("=", 1)
                env_dict[key.strip()] = value.strip()

        for service, config_needed in state["services_dict"].items():
            service_configs = {param: env_dict.get(param, None) for param in env_params_dict.get(service, [])}
            signin_configs[service] = service_configs
        state["updated_config"]["signin_configs"] = signin_configs

        return state



    def _all_services(self, state: ToolConfigState):
        filtered_dependencies = {}
        for key in state['ports'].keys():
```

```python
            if key in self.tool_dependencies:
                filtered_dependencies[key] = self.tool_dependencies[key
    ]
        state['ports'] = filtered_dependencies

        return state

    # PUBLIC METHODS TO INVOKE GRAPHS
    def invoke_tool_definition(self, state: ToolConfigState) ->
    ToolConfigState:
        return self.compiled_tool_def_graph.invoke(state)

    def invoke_env_file_generation(self, state: ToolConfigState) ->
    ToolConfigState:
        return self.compiled_env_file_graph.invoke(state)

    def invoke_config_details_retrieval(self, state: ToolConfigState)
    -> ToolConfigState:
        return self.compiled_config_details_graph.invoke(state)

    def invoke_refine_access_links(self, state: ToolConfigState) ->
    ToolConfigState:
        return self.compiled_refine_links_graph.invoke(state)

    def invoke_signin_configs_extraction(self, state: ToolConfigState)
    -> ToolConfigState:
        return self.compiled_signin_config_graph.invoke(state)

    def invoke_all_services(self, state: ToolConfigState) ->
    ToolConfigState:
        return self.compiled_all_services_graph.invoke(state)
```

Listing 3: A class that handles mmiscellaneous extraction of configration details/ports/access links and environment variables