

**Report**

**Design and Development of a Prototype**

**For**

**Blackboard based reliable Agentic AI Framework**

**Centre of Excellence for Distributed Artificial Intelligence**

**NetEdge Computing SolutionsPvt. Ltd.**

NetEdge Tower, C – 56/39, Sector - 62

NOIDA – 201 301

**Table of Content**

[1. Introduction 2](#_Toc626943476)

[2. Shared Blackboard Layer 4](#_Toc1632953082)

[3. Centralized Repository 5](#_Toc1004999632)

[4. Dynamic Linked List Implementation 6](#_Toc1255857249)

[4.1. Introduction 6](#_Toc323742959)

[4.2. Structure of the Implementation 6](#_Toc311076420)

[4.2.1 Solution Class (Node Representation) 6](#_Toc546496761)

[4.2.2 LinkedList Class 6](#_Toc1024418108)

[4.2.3 Dynamic Linked List Management 7](#_Toc157473585)

[4.3. Implementation Details 7](#_Toc1332524139)

[4.4. Example Execution Output 7](#_Toc727080536)

[5. Database Documentation: Agent Management System 8](#_Toc1575567379)

[5.1. Introduction 8](#_Toc1900214212)

[5.2. Database Structure 8](#_Toc1435630096)

[5.2.1 Database Name:  agent\_flow 8](#_Toc751555434)

[5.2.2 Tables: 8](#_Toc505412320)

[5.3. Table Description 8](#_Toc1386527199)

[5.3.1 Agents Table: agent\_info 8](#_Toc1858163504)

[5.3.2 Tags Table: Tag\_info 8](#_Toc45988712)

[5.3.3 Agents Tags Table:  Agent\_tags 9](#_Toc1178782225)

[5.3.4 Knowledge Source Table: knowledge\_source 9](#_Toc2037044169)

[5.4. Installation and Setup with XAMPP 9](#_Toc1575867612)

[5.5. Future Scope: Accessing Database with Python 10](#_Toc540316662)

[6. Communication Layer 10](#_Toc613353862)

[7. Message Format for Communication Between Client and Server 12](#_Toc1856304298)

[7.1. Request Message Format (Client → Server) 12](#_Toc905857978)

[7.2. Response Message Format (Server → Client) 13](#_Toc1807338227)

[7.3. Summary of Communication Flow 13](#_Toc1198044374)

[8. Intelligent Agents 14](#_Toc180690019)

[8.1. System Architecture 14](#_Toc17904230)

[8.2. Message Flow 15](#_Toc1131912903)

[8.3. Knowledge Source (Decision Tree) 15](#_Toc1326726628)

[8.4. Implementation Considerations 15](#_Toc2112292810)

[8.5. Conclusion 15](#_Toc444664039)

[9. Knowledge Source 16](#_Toc1309612354)

[9.1. Introduction 16](#_Toc729183687)

[9.2. Objectives 16](#_Toc328330297)

[9.3. System Architecture 17](#_Toc1998086224)

[9.3.1 Workflow of the Knowledge Source 17](#_Toc943278317)

[9.4. Future Enhancements 17](#_Toc1372842077)

[9.5. Conclusion 17](#_Toc828414256)

[10. Message Assembler 18](#_Toc787991884)

[10.1. Introduction 18](#_Toc439224796)

[10.2. System Architecture 18](#_Toc134941489)

[10.2.1 Data Flow in the System 18](#_Toc1256955)

[10.3. Working Mechanism 19](#_Toc1511071875)

[10.4. Future Enhancements 19](#_Toc393553503)

[11. Message disassembler 20](#_Toc451598363)

[11.1. Introduction 20](#_Toc955963716)

[11.2. System Architecture 20](#_Toc984748328)

[11.2.1 Data Flow in the System 20](#_Toc1424716253)

[11.3. Working Mechanism 21](#_Toc545529137)

[11.4. Future Enhancements 21](#_Toc1207621464)

[12. Central Repository Implementation 21](#_Toc1434583010)

[12.1. Introduction 21](#_Toc1159969312)

[12.2. System Workflow 21](#_Toc514369316)

[12.2.1 Parsing Incoming Message 21](#_Toc1660749948)

[12.2.2 Storing Parsed Data in Instant Solution 21](#_Toc1039468916)

[12.2.3 Forwarding Messages to Agents 21](#_Toc2133210626)

[12.2.4 Mock Agent Server for Testing 21](#_Toc2088003585)

[12.3. Implementation Details 21](#_Toc2101047472)

[12.3.1 Parsing Incoming Data (receive\_data) 21](#_Toc1144052862)

[12.3.2 Storing Data in Instant Solution 21](#_Toc808698817)

[12.3.3 Forwarding Messages to Agents 21](#_Toc1711333582)

[12.3.4 Mock Agent Servers 21](#_Toc698434104)

[12.3.5 Main Execution 21](#_Toc907692123)

[12.4. Conclusion 21](#_Toc826901585)

[13. Conclusion 21](#_Toc549365489)

# Introduction

The **Blackboard-Based Reliable Agentic AI Framework** is a multi-layered AI system designed to facilitate reliable, transparent, and efficient problem-solving through intelligent agents. The system is divided into three primary layers:

1. **User Experience Layer**
2. **Shared Blackboard Layer**
3. **Intelligent Agent Layer**

Currently, the focus of development is on the **Shared Blackboard Layer**, which acts as the central coordination hub for agentic AI interactions.

# Shared Blackboard Layer

The **Shared Blackboard Layer** is responsible for managing communication between various agents and ensuring reliable decision-making. This layer is further divided into the following components:

* **Regulatory Layer**: Ensures compliance with reliability standards, including trust, transparency, unbiasedness, and explainability.
* **Reliable Agents**: Designed to adhere to AI ethics and ensure fair decision-making.
* **Centralized Repository**: Stores intermediate solutions for efficient problem resolution.
* **Communication Layer**: Facilitates data exchange between agents and the repository.

# Centralized Repository

The **Centralized Repository** serves as a structured storage unit for solutions ents. Its key characteristics include:

* Storage of intermediate solutions as **JSON object**.
* Each JSON object contains:
* **Data** related to the problem.
* **Tags** that define which agent can solve the problem.
* All generated JSON object are stored in a **database** for easy retrieval and processing.
* **Unique Port Assignment**: Each agent is assigned a unique port number for identification.
* **Tag-Based Query System**: Each agent is assigned specific tags, allowing it to determine the problems it can solve based on metadata in the JSON objects.
* **Continuous Operation**: The system operates iteratively, processing data until no packets remain.
* **Parallel Data Reception**: Ensures efficient data handling and real-time response to new problem-solving tasks.
* **Automatic Restart Mechanism**: If new packets arrive, the system reactivates to process them without manual intervention.
* The repository is integrated with the **Communication Layer** to ensure seamless agent interaction.

# Dynamic Linked List Implementation

## Introduction

This report presents an implementation of a dynamic linked list system where multiple linked lists can be created and managed dynamically. The code provides a structured way to insert, remove, and print nodes in each linked list. The implementation uses Python classes to encapsulate the behavior of nodes and linked lists, allowing efficient data management.

## Structure of the Implementation

### Solution Class (Node Representation)

* Each node in the linked list is represented by an instance of the `Solution` class.
* It contains two attributes:
* `data`: Stores the value of the node.
* `next`: A pointer to the next node in the list.

### LinkedList Class

* This class provides functionalities to manage a singly linked list.
* Methods included:
* `\_\_init\_\_()`: Initializes an empty linked list.
* `insert(data)`: Adds a new node with the given data at the end of the list.
* `remove(key)`: Removes a node containing the given key from the list.
* `print\_list()`: Returns a string representation of the linked list.

### Dynamic Linked List Management

* A list (`array\_of\_linked\_lists`) stores multiple linked list instances.
* The `add\_linked\_list()` function dynamically creates a new linked list and appends it to the list.
* Each linked list is named sequentially as `A1`, `A2`, `A3`, etc.

## Implementation Details

* Three linked lists (`A1`, `A2`, `A3`) are dynamically created and populated with values.
* Nodes are inserted into each linked list sequentially.
* The `remove` method is tested by deleting specific nodes from `A1` and `A3`.
* The linked lists are then printed to display their final structure.

## Example Execution Output

Linked List A1: 1 -> 3 -> None   
Linked List A2: 4 -> 5 -> None   
 Linked List A3: 6 -> 7 -> 9 -> None

This implementation demonstrates how linked lists can be dynamically created and managed in Python. The modular approach ensures reusability and easy modification. The addition of a `remove` method enhances functionality, making it a comprehensive solution for linked list operations.

**Pseudocode**

CLASS Solution:

FUNCTION **init**(data):

SET self.data = data

SET self.next = NULL

CLASS LinkedList:

FUNCTION **init**():

SET self.head = NULL

FUNCTION insert(data):   
FUNCTION remove(key):

FUNCTION print\_list():

# Database Documentation: Agent Management System

## Introduction

The Agent Management System is designed to store and manage information about agents and their assigned port numbers, along with the associated tags. The database is created and managed using MySQL via XAMPP.

## Database Structure

### Database Name:  agent\_flow

### Tables:

* **Agents:** Stores details of agents and their assigned port numbers.
* **Tags**: Stores unique tag names.

## Table Description

### Agents Table: agent\_info

|  |  |  |  |
| --- | --- | --- | --- |
| **Column Name** | **Data Type** | **Constraints** | **Description** |
| agent\_id | INT | PRIMARY KEY, AUTO\_INCREMENT(1) | Unique identifier for each agent. |
| agent\_name | VARCHAR(255) | NOT NULL | Name of the agent. |
| port\_number | INT | UNIQUE, NOT NULL | Port number assigned to the agent. |

### Tags Table: Tag\_info

|  |  |  |  |
| --- | --- | --- | --- |
| Column Name | Data Type | Constraints | Description |
| tag\_id | INT | PRIMARY KEY, AUTO\_INCREMENT(1) | Unique identifier for each tag. |
| tag\_name | VARCHAR(255) | UNIQUE, NOT NULL | Unique name of the tag. |

**Ayush 11/02/2025**

### Agents Tags Table:  Agent\_tags

|  |  |  |
| --- | --- | --- |
| Column Name | Data Type | Constraints |
| agent\_id | INT | FOREIGN KEY REFERENCES Agents(agent\_id) ON DELETE CASCADE |
| tag\_id | INT | FOREIGN KEY REFERENCES Tags(tag\_id) ON DELETE CASCADE |
| **Primary Key** | (agent\_id, tag\_id) | Ensures unique agent-tag pairs |

1. The Agent\_Tags table acts as a bridge between Agents and Tags to represent the many-to-many relationship.
2. The foreign key constraints in Agent\_Tags ensure that an agent cannot be assigned to a non-existent tag and vice versa.
3. ON DELETE CASCADE ensures that if an agent or tag is deleted, all corresponding entries in Agent\_Tags are also removed.

***Priyanka Gupta(25/03/2025)***

### Knowledge Source Table: knowledge\_source

The Knowledge Source is backed by a MySQL database (intelligent\_agent\_db) that stores the rules dynamically.

|  |  |  |  |
| --- | --- | --- | --- |
| Column Name | Data Type | Constraints | Description |
| condition | VARCHAR(255) | PRIMARY KEY, NOT NULL | Tags used as conditions to trigger an action. |
| action | VARCHAR(255) | NOT NULL | The action executed when the condition is met. |

#### Sample Data:

|  |  |
| --- | --- |
| Condition (Tags) | Action |
| tag1 | action1 |
| tag2 | action2 |
| tag3 | action3 |
| tag4 | action4 |
| tag1\_tag2 | action5 |
| tag2\_tag3 | action6 |
| tag3\_tag4 | action7 |
| tag1\_tag3 | action8 |
| tag2\_tag4 | action9 |

## Installation and Setup with XAMPP

1. Install and start XAMPP.

2. Open phpMyAdmin from the XAMPP control panel.

3. Create a new database named `agent\_flow`.

4. Run the provided SQL script to create the tables.

## Future Scope: Accessing Database with Python

To interact with this database using Python, the `MySQL Connector` package can be used. Below is a sample Python script to connect and retrieve data:

import mysql.connector   
   
# Connect to MySQL database   
conn = mysql.connector.connect(   
    host='localhost',   
    user='root',   
    password='',     
    database='Agent\_flow'   
)   
   
cursor = conn.cursor()   
   
# Fetch agents data   
cursor.execute("SELECT \* FROM Agents")   
for row in cursor.fetchall():   
    print(row)   
   
# Close connection   
cursor.close()   
conn.close() 

# Communication Layer

The **Communication Layer** enables efficient and reliable communication between agents and the centralized repository. The key components are:

* Establish communication between agents and the repository.

***Priyanka Gupta(13/03/2025)***

# Message Format for Communication Between Client and Server

## Request Message Format (Client → Server)

The client sends a **JSON message** to the server, structured as follows:

<message>

{

    <tags>

    "tag": "TAG\_1,TAG\_2,TAG\_3......TAG\_N"

    </tags>

    <data>

    "data": {

        "param1","param2",……

    </data>

    }

}

</message>

**Fields Explanation:**

* **tag (string, required):** Stores names of tags.
* **body (object):** Contains any parameters or data needed for processing.

## Response Message Format (Server → Client)

The server processes the request and returns a JSON response:

<message>

{

    "status": "success",

    "response": {

        "data": {

            "result": "Processed value"

        }

    }

}

</message>

**Fields Explanation:**

* **status (string):** Indicates the success or failure of the operation ("success" or "error").
* **response (object,):** Contains the results of the operation.
* **data (object):** The computed result or output, if applicable.

## Summary of Communication Flow

1. **Client → Server:**
2. Sends a JSON message with a tag and data.
3. **Server:**
4. Parses the request and passes it to the **Knowledge Source** for rule-based processing.
5. If a valid rule is found, executes the action and generates a response.
6. **Server → Client:**
7. Sends a JSON response containing the status and result of the operation.

***Priyanka Gupta(10/03/2025)***

# Intelligent Agents

The purpose is to develop an intelligent agent capable of processing messages between a central repository server and client agents. The system leverages a rule-based decision tree to determine appropriate actions based on incoming messages. The architecture ensures efficient message handling and decision-making while supporting API integrations.

## System Architecture

The intelligent agent consists of the following core components:

* **Communication Layer:** Handles message reception and transmission.
* **Message Processing Module:** Parses incoming JSON messages, extracts tags, and routes them to the Knowledge Source.
* **Knowledge Source (Decision Tree):** Implements predefined rules to determine the necessary actions.
* **Action Execution Module:** Calls system functions or external APIs based on the rule evaluation.
* **Sending Queue:** Stores processed responses and sends them back to the server.

## Message Flow

1. The **server** sends a JSON message to the client agent.
2. The **client agent’s communication layer** receives the message.
3. The **message processing module** extracts components (tag and body).
4. The **knowledge source** applies rule-based decision-making based on the tag(s).
5. If an action is required, the **action execution module** performs a function call (internal or external API call).
6. The function’s response (JSON format) is added to the **sending queue**.
7. The **sending queue** transmits the result back to the server.

## Knowledge Source (Decision Tree)

* Each tag corresponds to a specific rule.
* Rules define actions such as:
* **Internal function calls** (e.g., data retrieval, processing computations)
* **External API calls** (e.g., fetching data from a third-party service)
* **No action**, if no relevant rule exists.

Example Decision Tree:

IF tag = "fetch\_user\_data" THEN call getUserData()

IF tag = "process\_transaction" THEN call processTransaction()

IF tag = "log\_event" THEN storeLog()

## Implementation Considerations

* **Scalability:** The rule-based system should efficiently handle a growing number of messages.

1. **Error Handling:** Implement logging and exception handling to ensure robustness.

* **Security:** Ensure secure message transmission and API interactions.

## Conclusion

This intelligent agent efficiently processes messages using a structured decision tree approach. It ensures seamless communication between the server and client agents while providing flexibility in executing internal and external actions. Future enhancements may include machine learning for adaptive decision-making.

***Priyanka Gupta(24/03/2025)***

# Knowledge Source

## Introduction

The **Knowledge Source** is a fundamental component of the **Intelligent Agent System** that acts as a **rule-based decision-making system**. It determines the appropriate **actions** based on the **tags** associated with each message. These actions can involve **API calls, function executions, or further message processing**. The **Knowledge Source** is implemented as a **database-driven decision tree**, ensuring flexibility, scalability, and efficient rule execution.

## Objectives

The primary objectives of the **Knowledge Source** are:  
 **Store predefined rules** in a structured database.  
 **Retrieve rules dynamically** based on incoming message tags.  
 **Execute specific actions** based on rule conditions.  
 **Return processed results** for further processing or final output.

## System Architecture

The **Knowledge Source** acts as an **intelligent decision-making hub** that interacts with **agents** to execute rule-based operations.

### Workflow of the Knowledge Source

1. **Agents** send messages containing **tags** and **body** to the **Knowledge Source** for rule execution.
2. **The Knowledge Source fetches predefined rules** from a database (**MySQL using XAMPP**).
3. **It matches the incoming tags** with the stored rules and determines the corresponding action.
4. **The assigned action is executed**, and the result is generated.
5. **The result is returned to the agent**, which then forwards it for further processing or final transmission to the **central repository**.

## Future Enhancements

🔹 **Machine Learning-Based Rule Suggestion:**

* Instead of manually defining rules, the system can **learn from historical data** and **suggest new rule mappings**.

🔹 **Automated Tag Assignment for Future Processing:**

* The system will dynamically assign **next required tags** based on **processing history**.
* If a message processed by **Agent1** with tag1 requires further processing by **Agent2**, the system will **automatically assign** tag2 to it.

## Conclusion

The **Knowledge Source** is a **critical decision-making component** in the **Intelligent Agent System**. It dynamically retrieves **rules from a database**, executes **predefined actions**, and returns the results.

With **future enhancements** like **machine learning-based rule suggestion** and **automated tag assignment**, the **Knowledge Source** will evolve into a **self-learning and adaptive AI-driven decision engine**.

***Priyanka Gupta(20/03/2025)***

# Message Assembler

## Introduction

In an intelligent multi-agent system, messages flow between agents and a central repository to process and exchange data dynamically. The **Message Assembler** is a critical component responsible for:

1. Collecting processed results from different agents.
2. Combining results with associated tags.
3. Formatting the message into a structured JSON format.
4. Storing messages in a queue.
5. Sending assembled messages to the central repository for the next processing iteration.

## System Architecture

The **Message Assembler** integrates into the **Intelligent Agent System**, which consists of:

* **Central Repository:** Sends messages to agents and receives processed results.
* **Agents:** Process messages based on assigned tags.
* **Knowledge Source:** Stores decision rules to determine actions based on tags.
* **Message Assembler:** Collects, assembles, and sends messages back to the central repository.

### Data Flow in the System

1. The **Central Repository** sends messages in **JSON format** to agents.
2. **Agents process the messages** using decision rules from the **Knowledge Source**.
3. Processed results are forwarded to the **Message Assembler**.
4. The **Message Assembler** formats and queues messages.
5. Messages are **sent back to the Central Repository** for further processing.

## Working Mechanism

#### **Step 1: Receiving Processed Results**

#### Each agent processes the incoming messages and forwards the **processed results** to the **Message Assembler**.

#### **Step 2: Assembling the Message**

The assemble\_message() function:   
 Extracts the **tag** from the agent.   
 Extracts the **processed result** from the agent's output.   
 Combines both into a JSON message.   
 Adds the message to the **sending queue**.  
 Immediately triggers send\_to\_central\_repo() to dispatch the message.

#### **Step 3: Sending to Central Repository**

The send\_to\_central\_repo() function:  
 Retrieves the next message from the **sending queue**.  
 Sends the message to the **Central Repository**.

## Future Enhancements

In the future, the **Message Assembler** can be improved with the following enhancements:

* **Database Integration:** Store assembled messages in a database before sending.
* **Retry Mechanism:** Implement retries in case of failed transmissions.
* **Batch Processing:** Send multiple messages in a single request to optimize performance.
* **Logging System:** Maintain logs of all sent and received messages.
* **Automated Tag Assignment:** The agent will **automatically assign tags** to the processed result based on the **next required tag**. This means:
* If an agent processes a message, it will determine the **next logical processing step**.
* The **next tag** may belong to the **same agent** or **another agent** in the system.
* This will ensure a **dynamic workflow** without manual intervention.
* **Example:** If **Agent1** processes tag1, and the next step requires tag3, the result will automatically be tagged as tag3 for further processing.

***Priyanka Gupta(21/03/2025)***

# Message disassembler

## Introduction

In a multi-agent intelligent system, messages are received from a central repository in JSON format. These messages contain tags and bodies that guide agents in processing them. However, before an agent can handle a message, it must be disassembled into its individual components so that the relevant agents can process their respective parts.

The Message Disassembler is a critical module responsible for:

1. Receiving JSON messages from the central repository.
2. Extracting tags and message bodies from incoming messages.
3. Sorting and distributing messages to relevant agents based on their assigned tags.
4. Maintaining a queue for efficient processing.

## System Architecture

The **Message Disassembler** is part of the **Intelligent Agent System**, which includes:

* **Central Repository:** Sends messages to the agents in JSON format.
* **Message Disassembler:** Extracts and assigns messages to the correct agents.
* **Agents (Agent1, Agent2, Agent3):** Process messages based on their respective tags.
* **Knowledge Source:** Stores decision rules for processing based on message tags.
* **Message Assembler:** Combines processed results and sends them back to the central repository.

### Data Flow in the System

1. The **Central Repository** sends messages in **JSON format** to the system.
2. The **Message Disassembler** extracts **tags and message bodies**.
3. Messages are **stored in a receiving queue** for processing.
4. The system **assigns messages to the correct agents** based on tags.
5. **Agents process the messages** and send results to the **Message Assembler**.
6. The **Message Assembler** forwards the assembled message back to the **Central Repository**.

## Working Mechanism

#### **Step 1: Receiving Messages**

The **receive\_messages()** function:  
 Accepts a **JSON-formatted input** containing multiple messages.  
 Parses the input and converts it into a **Python list**.  
 Adds each message to a **receiving queue**.  
 Prints the **received messages** for debugging.

#### **Step 2: Distributing Messages to Agents**

The **distribute\_messages()** function:  
 Extracts the **tags and body** from each message.  
 Determines which agent(s) should process each message based on **predefined rules**.  
 Adds messages to the respective **agent-specific queues**.  
 Prints the **distributed messages** for debugging.

## Future Enhancements

In the future, the **Message Assembler** can be improved with the following enhancements:

* **Database Integration:** Store received messages in a database before processing.
* **Retry Mechanism:** Implement retries in case of failed distributions.
* P**riority Handling:** Implement priority-based message routing for urgent tasks.
* **Logging System:** Maintain logs of all incoming and distributed messages.

# **Central Repository Implementation**

## Introduction

This report explains the implementation of an agentic framework for task assignment based on tags. The system processes incoming messages, extracts relevant data, stores instant solutions, and forwards tasks to assigned agents.

## System Workflow

### **Parsing Incoming Message**

* Messages containing task-related data and tags are received.
* The message is parsed using regular expressions to extract:
  + **Tags**: Keywords associated with the task.
  + **Data**: The content or instructions related to the task.

### **Storing Parsed Data in Instant Solution**

* The extracted information is stored in a temporary structure called instant\_solution.
* This storage ensures that messages are not lost before processing.
* If the tag already exists in the instant\_solution list, new data is appended to the existing entry.

### **Forwarding Messages to Agents**

* The system checks for agents mapped to the received tags.
* Using TCP socket connections, messages are forwarded to the appropriate agent ports.
* After successful transmission, the message count is updated.
* Once all agents receive the message, it is removed from instant\_solution.

### **Mock Agent Server for Testing**

* To simulate agent behavior, mock servers are deployed on predefined ports.
* These servers listen for messages and confirm receipt.

## **Implementation Details**

### **Parsing Incoming Data (receive\_data)**

The function extracts **tags** and **data** from the received message format (e.g., XML or JSON-like structure). It then returns a dictionary containing the extracted information.

### **Storing Data in Instant Solution**

* The function maintains an internal list (instant\_solution) to store messages.
* Each entry follows the format:

[tag\_id, count, "data1, data2"]

### **Forwarding Messages to Agents**

* The system identifies agents mapped to the tags.
* Using TCP socket programming, messages are sent to the respective agent ports.
* Successful transmissions decrease the count, and once all agents have received the task, it is removed from the instant solution list.

### **Mock Agent Servers**

* Simulated agent servers are implemented to test message reception.
* Each server listens on a unique port and acknowledges received messages.

### **Main Execution**

* Initializes the MySQL database connection (for tag-agent mapping).
* Starts mock agent servers for testing. Processes a sample input message.
* Calls parsing, storage, and forwarding functions sequentially. Closes database connections after execution.

## **Conclusion**

This agentic framework provides a structured mechanism for **task distribution and automation** using a **tag-based assignment model**. The integration of **instant solutions** ensures smooth message handling, while **TCP sockets** facilitate reliable agent communication. The system is scalable and can be extended with additional functionalities to optimize workflow automation further.

Ayush Dubey 26/03/2025

# Conclusion

The Blackboard-Based Reliable Agentic AI Framework aims to enhance AI-driven decision-making by ensuring transparency, reliability, and efficiency. The Shared Blackboard Layer serves as the backbone of the system, facilitating structured communication and effective agent collaboration. The proposed implementation plan will ensure seamless integration and efficient problem-solving capabilities.