System Architecture Document (SAD)

Real-Time Monitoring and Control System for Industrial Automation

V1.0

1. System Architecture Diagram

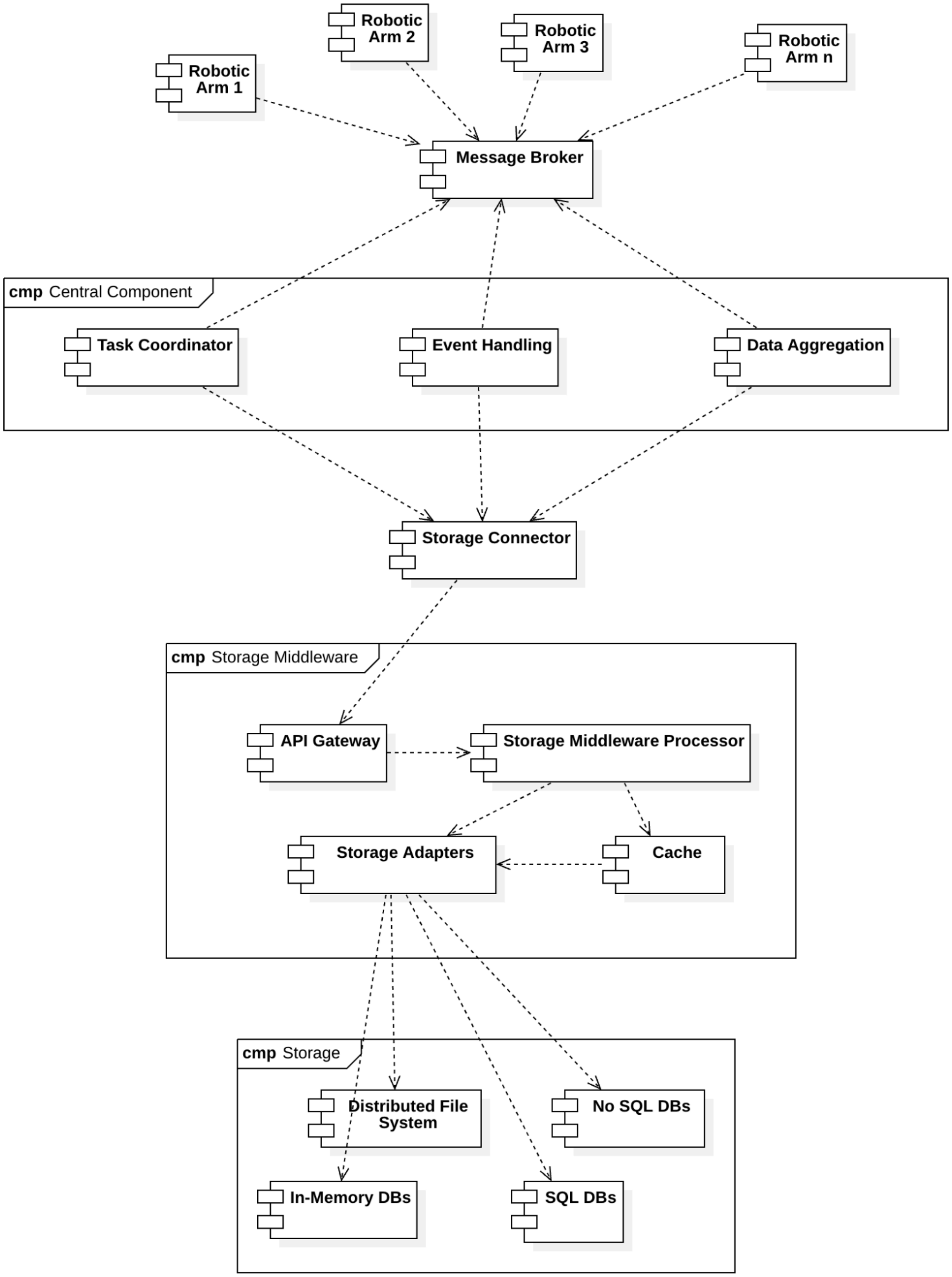

Diagram 1. System Architecture

The **System Architecture Diagram** provides a visual representation of the core components and their interactions within the Real-Time Monitoring and Control System. The diagram includes the following key components:

- **Central Component Microservices**: A set of microservices responsible for managing, coordinating, and monitoring the operations of the system, including the interaction with robotic arms and other microservices. This component acts as the brain of the system, orchestrating workflows, processing data, and ensuring that all services work together seamlessly.
- **Robotic Arms**: Represents a set of simulated microservices that emulate the behavior of physical robotic arms. These microservices are responsible for executing tasks assigned by the Central Component, reporting their status, and coordinating with each other through the Message Broker. The Robotic Arms component is designed to mirror the operations of actual robotic hardware in a controlled and scalable software environment.
- **Message Broker**: A critical component responsible for facilitating communication between the various microservices, including the Central Component microservices and the simulated robotic arms. This layer uses Apache Kafka to manage the asynchronous exchange of messages, ensuring reliable and scalable communication across the system.
- **Storage Connector**: A high-level wrapper library that simplifies communication with the storage middleware API by abstracting complex implementation details. It provides a unified interface, ensuring consistent and reliable communication with storage services while managing tasks like request formatting and handling the API endpoint addresses. The connector standardizes interactions with the storage middleware infrastructure.
- **Storage Middleware**: Serves as a unified interface between the application services and various underlying data storage backends, such as SQL databases, NoSQL databases, and file storage systems. This layer abstracts the complexities of interacting with different storage technologies, providing a consistent API for CRUD operations, transactions, and data queries across the system.
- **Storage**: Represents the underlying data repositories where all persistent data is stored, including configuration data, logs, user information, and operational data. This component interacts directly with the Storage Middleware Layer, which manages access to various types of storage backends such as SQL databases, NoSQL databases, and file systems.

2. Component Descriptions

2.1 Central Component Microservices

The Central Component is the operational hub of the system, providing essential management, coordination, and monitoring functions that keep the entire system running smoothly and efficiently. It plays a crucial role in maintaining the system's reliability, performance, and responsiveness.

Key Features:

- **Task Coordination:** The Central Component assigns and manages tasks across the system, directing commands to robotic arms and other services based on real-time data and predefined workflows.
- **Service Orchestration:** Coordinates the interactions between various microservices, ensuring that tasks are executed in the correct order and that services communicate effectively through the Message Broker.
- **Real-Time Monitoring:** Continuously monitors the status and performance of robotic arms and other microservices, collecting data on system health, task progress, and operational metrics.
- **Fault Tolerance and Recovery:** Detects and handles errors, system failures, and other exceptions, initiating recovery processes or rerouting tasks as necessary to maintain system stability and uptime.
- **Scalability and High Availability:** Deployed on a Kubernetes cluster, the Central Component is designed to scale horizontally, with multiple instances running to ensure high availability and load distribution.

2.2 Robotic Arms Microservices

The Robotic Arms component is a crucial part of the system, enabling the simulation of robotic operations, testing of complex task coordination, and ensuring that the system can adapt to varying operational demands in a scalable and efficient manner.

Key Features:

- **Task Execution:** Each robotic arm microservice receives specific commands from the Central Component and performs the assigned tasks, such as assembly, manipulation, or inspection activities, depending on the system's needs.
- **Status Reporting:** The robotic arms continuously monitor their own operations and send real-time status updates back to the Central Component. This includes task completion, operational metrics, and any error conditions encountered.
- **Inter-Arm Coordination:** When tasks require collaboration between multiple robotic arms, these microservices communicate through the Message Broker to synchronize their actions, ensuring that tasks requiring precise timing and coordination are executed correctly.
- **Scalability**: The Robotic Arms component is designed to scale horizontally, allowing you to deploy and manage multiple robotic arms as microservices within a Kubernetes cluster. This scalability ensures that the system can handle an increasing number of tasks or more complex workflows as needed.

- **Simulation and Testing:** As simulated entities, the robotic arms provide a safe and controlled environment for testing and validating the system's logic, coordination mechanisms, and task management processes without the risks associated with real-world hardware.

2.3 Message Broker

The Message Broker ensures efficient, reliable, and scalable communication across the system, playing a vital role in maintaining the overall system's robustness and responsiveness.
Key Features:

- **Asynchronous Communication:** Enables decoupled communication between services, allowing them to operate independently and handle messages at their own pace.
- **Scalability:** Supports high-throughput message processing, allowing the system to scale seamlessly as the volume of communication grows.
- **Fault Tolerance**: Provides durability and fault tolerance through Kafka's replication mechanisms, ensuring that messages are not lost even if a service or node fails.
- **Inter-Service Coordination**: Facilitates complex workflows by allowing services to publish and subscribe to relevant topics, enabling coordinated actions between microservices, such as task delegation and status updates.
- **Real-Time Data** Streaming: Supports real-time data streaming and monitoring, allowing for immediate processing and analysis of messages.

2.4 Storage Middleware Layer

The Storage Middleware Layer serves as an abstraction layer that simplifies and centralizes data management, enhancing the flexibility, scalability, and reliability of the overall system. The Central Component interacts with the Storage Middleware through the Ocelot API Gateway, which manages and routes requests to the appropriate storage backend, while also integrating a Cache Layer to optimize performance and reduce latency.

Key Features:

- **Unified Data Access:** The API Gateway (Ocelot) provides a single point of access for all data operations, regardless of the underlying storage backend. This simplifies data management for application services by abstracting the complexities of interacting with different storage systems. The inclusion of a Cache Layer further enhances this access by providing faster data retrieval for frequently accessed data, reducing the need to interact with the underlying storage systems for each request.
- **Scalability:** Designed to scale horizontally, the Storage Middleware Layer can handle increasing data loads by distributing requests across multiple storage backends. The API Gateway facilitates this by efficiently routing requests to the appropriate storage service. Additionally, by hiding the Storage Middleware behind the API Gateway, the entire system, including the Kubernetes cluster, can be scaled horizontally. This ensures that both the middleware and the infrastructure supporting it can grow as demand increases.

- **High Availability:** Deployed on a Kubernetes cluster, the Storage Middleware ensures high availability with built-in failover mechanisms and redundancy. The API Gateway adds an additional layer of reliability by managing traffic and rerouting requests in case of backend failures, minimizing downtime and ensuring continuous data access.
- **Modular Architecture:** The Storage Middleware supports a modular architecture with pluggable database connectors and a pluggable Cache Layer. This allows easy integration with new databases, storage systems, or caching technologies without requiring changes to the application logic. The API Gateway abstracts these connections, making the system more adaptable to changes in the storage and caching landscape.
- **Performance Optimization:** The Cache Layer significantly improves the performance of the system by reducing latency and increasing the speed of data retrieval for repeated requests. By storing frequently accessed data in the cache, the system minimizes the need to repeatedly query the underlying storage systems, resulting in faster response times and more efficient data operations.
- **Security and Compliance:** The Storage Middleware, in conjunction with the API Gateway and Cache Layer, implements robust security measures including data encryption, access control, and audit logging. This ensures that all data operations are secure and compliant with regulatory standards, with the API Gateway and Cache Layer enforcing these policies consistently across all data requests.

2.5 Storage Layer

The **Storage** component serves as the foundational layer for data management within the system, providing the necessary infrastructure to store, secure, and retrieve all forms of persistent data.

Key Features:

- **Diverse Storage Backends:** The Storage component can include a variety of storage technologies, such as relational databases (e.g., MySQL, PostgreSQL), NoSQL databases (e.g., MongoDB, Cassandra), and distributed file systems or object storage (e.g., S3-compatible storage).
- **Data Persistence:** Ensures that all critical data is stored reliably and can be retrieved as needed, supporting both short-term operational needs and long-term data retention requirements.
- **Redundancy and Replication:** To enhance data availability and fault tolerance, the Storage component may employ redundancy and replication strategies, ensuring that data is not lost in the event of hardware failures or other disruptions.
- **Secure Data Management:** Implements security measures such as data encryption, access controls, and audit logging to protect sensitive information and ensure compliance with data protection regulations.
- **Scalability:** Designed to scale with the growth of the system, the Storage component can expand to accommodate increasing data volumes, either by scaling vertically (adding more resources to existing databases) or horizontally (adding more database instances or nodes).

3. Technology Stack

3.1 Core Technologies

- **ASP.NET Core:** Utilized for building the Service Layer of the Storage Middleware. This framework provides a robust and scalable environment for handling HTTP requests, managing API endpoints, and ensuring secure communication between services and the storage backend.
- **.NET Core Worker Service**: Employed for developing the Central Component microservices and Robotic Arm Simulators. The Worker Service template is ideal for long-running background tasks, allowing these components to operate continuously without the need for an HTTP interface, while still integrating seamlessly with other parts of the system.
- **Ocelot:** Functions as the API Gateway, specifically responsible for distributing incoming traffic from the Central Components to the Storage Middleware cluster. Ocelot manages and routes these requests efficiently, ensuring that storage operations are handled seamlessly within the Kubernetes cluster, and that the system can handle high loads with improved scalability and resilience.
- **Apache Kafka:** The Message Broker facilitating communication between microservices and robotic arms, providing reliable and scalable messaging.
- **Docker:** Containerization platform used to package and deploy microservices consistently across environments.
- **Kubernetes:** Orchestration tool for managing containerized applications, ensuring high availability, scalability, and automated deployment.

3.3 Security

- **TLS/SSL:** Used to encrypt communication between all system components, including Kafka, microservices, and external interfaces.
- **SASL** (Simple Authentication and Security Layer): Provides authentication for Kafka, ensuring that only authorized services can produce and consume messages.

4. Communication Protocols

4.1 Protocols Overview

- TCP/IP: Used as the underlying transport protocol for all communication within the system, ensuring reliable data transmission in the on-premise environment.
- Kafka Protocol: Custom protocol used by Apache Kafka for all message brokering. It handles asynchronous communication between the Central Component, Robotic Arms, and inter-arm communication.
- gRPC: Used for communication between the Central Component and the Storage Middleware, enabling high-performance and low-latency interactions. gRPC is also used internally within the Central Component microservices when direct communication is required.

- HTTP/HTTPS: Utilized by the ASP.NET Core API Gateway/Service Layer for handling external client requests. HTTPS is enforced for secure communication.

4.2 Component Communication

- Central Component to Robotic Arms:
  - Protocol: Kafka Protocol over TCP
  - Description: The Central Component sends task commands to the Robotic Arms and receives status updates through Kafka topics. This ensures that all messages are reliably queued and can be processed asynchronously, allowing for scalable and decoupled communication.
- Robotic Arms to Central Component:
  - Protocol: Kafka Protocol over TCP
  - Description: Robotic Arms send their status updates, task completion reports, and any fault notifications to the Central Component via Kafka. This communication ensures that the Central Component has a real-time view of the system's status and can coordinate tasks effectively.
- Inter-Robotic Arm Communication:
  - Protocol: Kafka Protocol over TCP
  - Description: When tasks require coordination between multiple Robotic Arms, they communicate through dedicated Kafka topics. This approach ensures that messages between arms are handled asynchronously, facilitating precise coordination even in complex workflows.
- Central Component to Storage Middleware:
  - Protocol: gRPC over TCP
  - Description: All interactions between the Central Component and the Storage Middleware are handled using gRPC, which provides efficient, low-latency communication for CRUD operations, data retrieval, and storage management. This ensures that the Storage Middleware can handle high-throughput data requests effectively.
- Central Component to API Gateway (Ocelot):
  - Protocol: HTTP/HTTPS over TCP
  - Description: Ocelot distributes incoming requests to multiple instances of the Central Component, balancing the load and ensuring high availability across the system.

4.3 Security Measures

- Encryption:
  - All communication between system components, including Kafka messages and gRPC calls, is encrypted using TLS/SSL to prevent unauthorized access and ensure data integrity.
- Authentication:
  - SASL is used for authenticating connections to the Kafka Message Broker, ensuring that only authorized microservices can publish or consume messages.

    - Mutual TLS (mTLS) may be used with gRPC to ensure that both the client (Central Component) and server (Storage Middleware) authenticate each other.

5. Communication Data Flow Between Components

5.1 Central Component to Robotic Arms

- Task Assignment and Execution:
    - Message Sent: The Central Component sends task commands to specific Robotic Arms.
    - When: When tasks are assigned or re-prioritized.
    - Protocol: Kafka Protocol over TCP (direct communication via Kafka topics).
    - Data: Task commands include task ID, task details (e.g., pick up a PCB), and priority level.
    - Response: Robotic Arms send status updates, such as task started, in progress, or completed back to Kafka topics.
- Inter-Robotic Arm Coordination:
    - Message Sent: Coordination messages between Robotic Arms, facilitated by the Central Component.
    - When: When a task requires multiple robotic arms to collaborate (e.g., one arm holds a component while another assembles it).
    - Protocol: Kafka Protocol over TCP.
    - Data: Coordination messages include task ID, current status, and any necessary synchronization data (e.g., "Ready to proceed with assembly").
    - Response: Robotic Arms acknowledge receipt and adjust their operations accordingly.
- Real-Time Data Streaming:
    - Message Sent: Commands to start or stop streaming performance metrics.
    - When: When continuous data streaming is required.
    - Protocol: Kafka Protocol over TCP.
    - Data: Includes streaming commands, topics/queues for data, and specific metrics to be streamed.
    - Response: Robotic Arms begin streaming data like temperature, speed, etc., back to the Message Broker (Kafka).

5.2 Central Component to Storage Middleware

- Data Management and Storage:
    - Message Sent: CRUD operations (Create, Read, Update, Delete) related to configuration data, logs, task queues, and performance metrics.
    - When: On system initialization, task assignment, task completion, and shutdown.
    - Protocol: gRPC over TCP (through Ocelot API Gateway).
    - Data: Includes configuration files, task logs, operational data, and performance metrics.

    - Response: Acknowledgments from the Storage Middleware confirming the successful execution of the operation.
- Event Handling:
    - Message Sent: Event notifications regarding task completions, errors, or hardware failures.
    - When: When events are detected by the Central Component or Robotic Arms.
    - Protocol: gRPC over TCP (through Ocelot API Gateway).
    - Data: Event details, including event ID, type, timestamp, and related data.
    - Response: Confirmation that the event has been logged and appropriate actions have been taken.

5.3 Robotic Arms to Central Component

- Status Reporting:
    - Message Sent: Regular status updates, including task progress and any fault conditions.
    - When: Continuously during task execution or when an error is detected.
    - Protocol: Kafka Protocol over TCP.
    - Data: Task ID, status (e.g., in progress, completed, failed), and operational metrics (e.g., temperature, motor speed).
    - Response: The Central Component logs the status and updates the task queue accordingly.
- Fault Detection and Alerting:
    - Message Sent: Error messages when a robotic arm detects a hardware failure or critical issue.
    - When: When a fault occurs during operation.
    - Protocol: Kafka Protocol over TCP.
    - Data: Error ID, description, timestamp, and suggested corrective actions.
    - Response: The Central Component generates an alert and initiates recovery processes.

5.4 Inter-Robotic Arm Communication

- Coordination Messages:
    - Message Sent: Messages exchanged between Robotic Arms to synchronize actions.
    - When: During tasks that require multiple arms to work in tandem.
    - Protocol: Kafka Protocol over TCP.
    - Data: Task details, synchronization points, and readiness status.
    - Response: Other robotic arms acknowledge receipt and adjust their operations accordingly.

6. Components API Interfaces

1.1. Robotic Arms Microservices API (Worker Services)

Given that these services are .NET Worker Services and communicate primarily through Kafka, here’s how their APIs can be structured:

- Task Execution Service:
  - Message Type: Kafka Message
  - Topic: tasks.execute
    Payload:

    ```
    json
    {
      "taskId": "string",
      "taskDetails": "string",
      "priority": "int"
    }
    ```

  - Response: Status update sent to tasks.status.<taskId> topic.
- Status Reporting Service:
  - Message Type: Kafka Message
  - Topic: status.update.<roboticArmId>
    Payload:

    ```
    json
    {
      "taskId": "string",
      "status": "string",
      "metrics": {
        "temperature": "float",
        "speed": "float"
      }
    }
    ```

  - Response: Acknowledgment received from Central Component via status.ack.<roboticArmId> topic.
- Fault Reporting Service:
  - Message Type: Kafka Message
  - Topic: fault.report.<roboticArmId>
    Payload:

    ```
    json
    {
      "errorId": "string",
      "description": "string",
    ```

```
  "timestamp": "timestamp"
}
```

    - Response: Acknowledgment from Central Component via fault.ack.<roboticArmId> topic.

1.2. Central Component Microservices API (Worker Services)

The Central Component manages task coordination, real-time monitoring, and storage interactions. Here's how the APIs are defined:

- Task Assignment Service:
    - Message Type: Kafka Message
    - Topic: tasks.assign
      Payload:

```
json
{
  "roboticArmId": "string",
  "taskId": "string",
  "taskDetails": "string"
}
```

    - Response: Assignment status sent back via tasks.status.<taskId> topic.
- Data Streaming Control Service:
    - Message Type: Kafka Message
    - Topic: stream.control
      Payload:

```
json
{
  "streamId": "string",
  "metrics": ["temperature", "speed"]
}
```

    - Response: Stream status update sent to stream.status.<streamId> topic.
- Event Handling Service:
    - Message Type: Kafka Message
    - Topic: events.handle
      Payload:

```
json
{
  "eventId": "string",
  "eventDetails": "string"
}
```

    - Response: Action taken status sent to events.status.<eventId> topic.

1.3. Storage Middleware Microservices API (Worker Services)

The Storage Middleware is responsible for managing all data-related operations through gRPC, facilitated by the Ocelot API Gateway:

- CRUD Operations Service:
    - Message Type: gRPC Request (through Ocelot API Gateway)
    - Method: Create, Read, Update, Delete

      Payload:

      ```
      json
      {
        "dataId": "string",
        "dataPayload": "object"
      }
      ```

    - Response: Operation status returned through gRPC.
- Data Query Service:
    - Message Type: gRPC Request (through Ocelot API Gateway)
    - Method: Query

      Payload:

      ```
      json
      {
        "query": "string"
      }
      ```

    - Response: Query results returned through gRPC.
- Event Logging Service:
    - Message Type: gRPC Request (through Ocelot API Gateway)
    - Method: Log

      Payload:

      ```
      json
      {
        "eventId": "string",
        "eventDetails": "string"
      }
      ```

    - Response: Log status returned through gRPC.