International Journal of Business Quantitative Economics and Applied Management Research

Volume-7, Issue-9, 2023 ISSN No: 2349-5677

ENHANCING FLEXIBILITY OF WAREHOUSE MANAGEMENT SYSTEMS THROUGH STATE MACHINE-DRIVEN ARCHITECTURE

Gautham Ram Rajendiran
gautham.rajendiran@icloud.com

Abstract

Warehouse Management Systems (WMS) play a critical role in streamlining supply chain operations by providing various "skills" like container scanning, labeling, picking, and loading. Traditional monolithic WMS architecture faces difficulties to adapt to the diverse workflow variations for different use cases, hence making the systems rigid and hard to maintain. The dynamic configuration of workflows for various use cases is discussed in this paper for flexibility and maintainability of the system through a state machine-driven approach. It thus decouples the configuration of the skills from the underlying code, allowing WMS to handle even very complex scenarios with minimal changes within itself. The proposed system leverages state machines to handle state transitions and UI flows based on current needs of required skills. This paper discusses the architecture and benefits and provides some insight into the implementation details of this state machine-driven WMS exploring the reduced complexity and improved maintainability of the system.

Keywords: Warehouse Management System, State Machine, Software Engineering, Supply Chain, Scalability, Maintainability

I. INTRODUCTION

Warehouse Management Systems have undergone significant changes in the past few decades in order to respond to the always-growing demanding scenarios set by modern supply chain operations [1]. Usually, a WMS manages the activities related to the processing of items in a warehouse such as loading inventory into containers, loading containers into pallets, picking pallets and bins to get ready for loading and loading bins into trucks among other operations and has become a crucial component in modern supply chain systems. Classic monolithic WMS systems tend to develop these activities or "skills" independently as classes or components orchestrated by a high-level manager class according to the specific use case.

While this would work for a few cases, implementing many such use cases in this way is pretty cumbersome and error-prone. For instance, a system operating both with inbound and outbound logistics, where operations such as the picking process should not be needed when setting up long-term storage, exponentially develops the complexity of the codebase. Every time any new workflow is introduced, it requires big changes in the core system; thus, the cost of its maintenance goes up and hence is more faulty.

This paper hereby proposes an approach toward more flexible WMS workflow development by means of dynamic configuration through a state machine [2] by means of a configuration file for different skill sequences in different use cases without any complicated branching logic in code.

The difference between the conventional monolithic architecture and the state machine driven architecture are illustrated through Fig1 and Fig2 shown below.

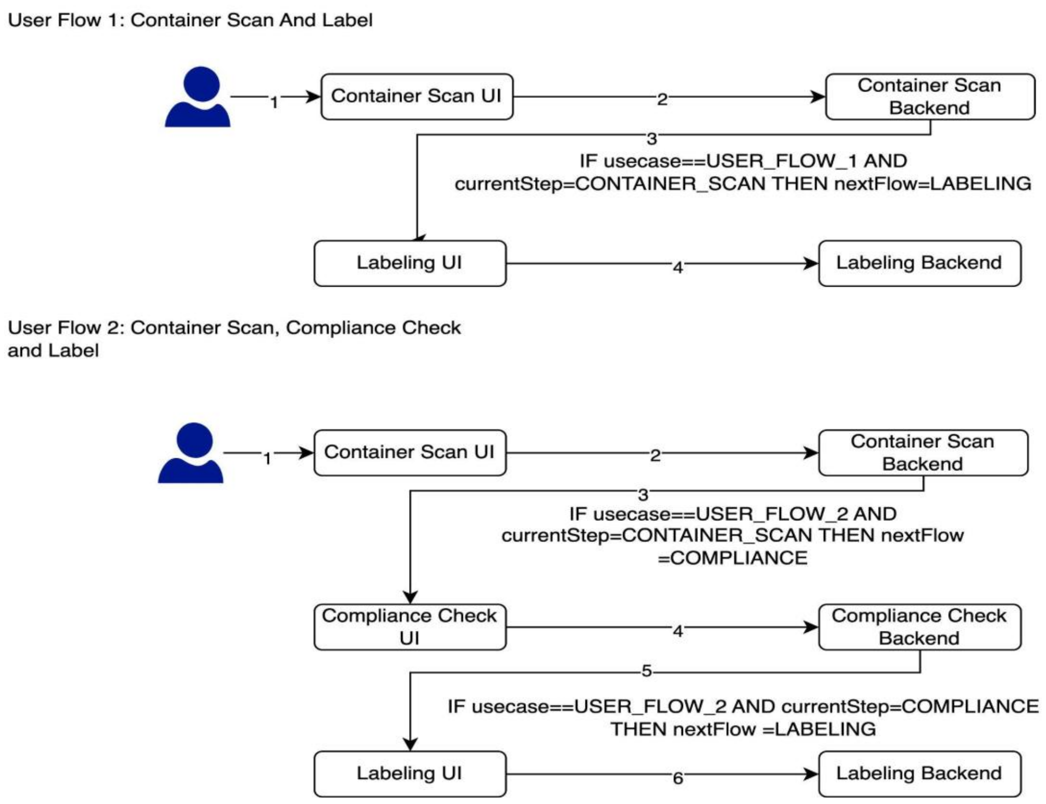
User Flow 1: Container Scan And Label
1
Container Scan UI
2
Container Scan Backend
3
IF usecase==USER_FLOW_1 AND currentStep=CONTAINER_SCAN THEN nextFlow=LABELING
Labeling UI
4
Labeling Backend
User Flow 2: Container Scan, Compliance Check and Label
1
Container Scan UI
2
Container Scan Backend
3
IF usecase==USER_FLOW_2 AND currentStep=CONTAINER_SCAN THEN nextFlow =COMPLIANCE
Compliance Check UI
4
Compliance Check Backend
5
IF usecase==USER_FLOW_2 AND currentStep=COMPLIANCE THEN nextFlow =LABELING
Labeling UI
6
Labeling Backend

Fig 1: WMS flow for Monolithic System Design

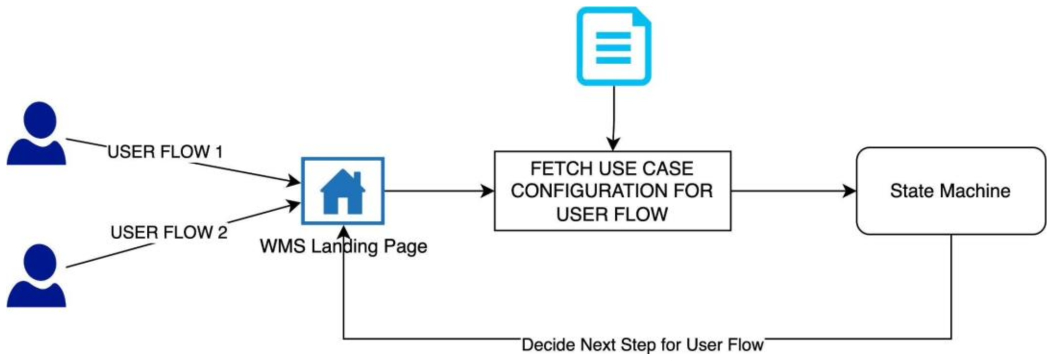
USER FLOW 1
USER FLOW 2
WMS Landing Page
FETCH USE CASE CONFIGURATION FOR USER FLOW
State Machine
Decide Next Step for User Flow

Fig 2: WMS flow Using State Machine Design

II. METHODOLOGY

The proposed architecture consists of the following key components, as depicted in Fig 3:

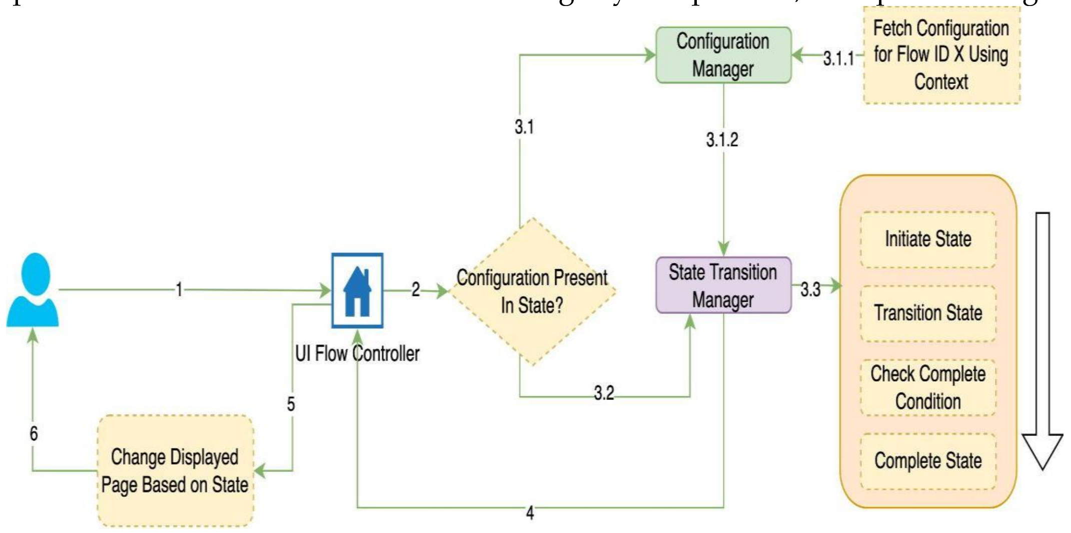

Fig 3: Components

Step 1: User Interaction Initiates Workflow

The user initiates the workflow through a request, such as scanning a container, adding an item to inventory, or beginning the loading process. This action is captured by the UI Flow Controller, which manages the user interface and the associated navigation logic. The UI Flow Controller acts as the primary interaction point between the user and the system, ensuring that the current state of the workflow is correctly represented in the user interface.

Step 2: UI Flow Controller Sends Request for Configuration Validation

The UI Flow Controller forwards the user request to check if a configuration is already present for the current state. This request is routed to the Configuration Manager, which is responsible for managing workflow configurations. The UI Flow Controller also sends relevant contextual information, such as the use case identifier (e.g., "X" for outbound logistics or "Y" for problem-solving), to facilitate configuration retrieval.

Step 3.1: Configuration Manager Fetches the Relevant Workflow Configuration

The Configuration Manager processes the request and uses the provided context and flow ID to fetch the appropriate configuration file. The configuration file contains a detailed list of skills, their order, and the associated context that will drive the state transitions for the use case. For example, a configuration for a standard outbound logistics flow might look like this:

```
{
        "caseIdentifier": "X",
                "skillConfigurationList": [
                        {"skillName": "Container Scan", "context": {...}},
                        {"skillName": "Pallet Scan", "context": {...}},
                        {"skillName": "Labeling", "context": {...}},
                        {"skillName": "Inventory Reservation", "context": {...}},
                        {"skillName": "Picking", "context": {...}},
                        {"skillName": "Loading", "context": {...}}
                ]
}
```

If the configuration is found, it is sent back to the State Transition Manager. If not, the Configuration Manager returns a response indicating that no configuration is available, prompting the system to handle the case accordingly.

Step 3.2: State Transition Manager Initializes and Manages the State Machine
The State Transition Manager initializes the state machine using the retrieved configuration. It sets the initial state and prepares the workflow to progress according to the sequence defined in the configuration. The state machine handles all state transitions based on the user's actions and external inputs, ensuring that each skill is executed in the correct order.

1. Initiate State: The state machine begins by setting the initial skill (e.g., "Container Scan") and sends a response back to the UI Flow Controller with an identifier like CONTAINER_SCAN to notify the UI of the current state.
2. Transition State: As the user completes each skill, the state machine transitions to the next skill. For example, after completing the container scan, the state machine transitions to the pallet scan. The state machine sends the identifier PALLET_SCAN to the UI Flow Controller, along with any additional context required for the UI to display the next skill's page.
3. Check Complete Condition: Each state checks whether the completion criteria for the current skill are satisfied. This involves verifying that the user has completed all necessary actions (e.g., scanning all items) and that there are no unresolved issues.
4. Complete State: Once all conditions are met, the state machine marks the skill as complete and sends a completion message to the UI Flow Controller. The UI Flow Controller updates the displayed page accordingly and prepares for the next state.

Step 4: State Transition Manager Sends State Identifier and Context to UI
After the State Transition Manager transitions to a new state, it sends back a response to the UI Flow Controller containing a unique identifier for the current state (e.g., LABELING, PICKING) along with any relevant contextual information. The identifier indicates to the UI which page or

component should be displayed next. For instance, if the identifier is LABELING, the UI Flow Controller knows to display the labeling page.
The contextual information might include additional details such as the items to be labeled, expected completion time, or instructions for the user. This context allows the UI Flow Controller to render the page dynamically based on the state and to provide the user with the necessary information for completing the current skill.

Step 5: UI Flow Controller Updates the Displayed Page
Upon receiving the state identifier and context, the UI Flow Controller updates the displayed page to match the current state. For example, if the state is CONTAINER_SCAN, the UI will show a container scan page with fields and instructions for scanning items. As the state changes, the UI transitions to the next page (e.g., pallet scan, labeling, or problem solving) according to the state machine's progression.
The UI Flow Controller also listens for user actions (e.g., button clicks, barcode scans) and sends these actions back to the state machine for processing. This tight integration between the UI Flow Controller and the State Transition Manager ensures that the displayed UI always matches the underlying state of the workflow.

Step 6: User Completes Current Skill and Initiates the Next State
The user interacts with the UI components to complete the current skill. For instance, they might scan a container's barcode, enter labeling information, or select items to be picked. Upon completing the required actions, the UI Flow Controller sends a signal back to the State Transition Manager to update the state machine's status.
The state machine processes the event, verifies that all conditions for the current state are met, and then transitions to the next state in the workflow. The State Transition Manager sends back a new state identifier and context, and the UI Flow Controller updates the UI accordingly. This loop continues until the entire workflow is completed.

III. IMPLEMENTATION USING AMAZON WEB SERVICES

The implementation of the proposed architecture integrates four primary components—AWS AppConfig [3] for configuration management, an ECS-Fargate [4] based microservice to handle the state machine logic, a PostgreSQL [5] database to persist and manage state transitions, and a React-Redux [6] front-end to display and update the current state. The interactions between these components are carefully orchestrated to ensure that the Warehouse Management System (WMS) can dynamically adapt to varying workflows based on external configurations, providing a scalable and maintainable architecture. Below, we elaborate on each component's implementation and how they interact within the system.

International Journal of Business Quantitative Economics and Applied Management Research
Volume-7, Issue-9, 2023 ISSN No: 2349-5677

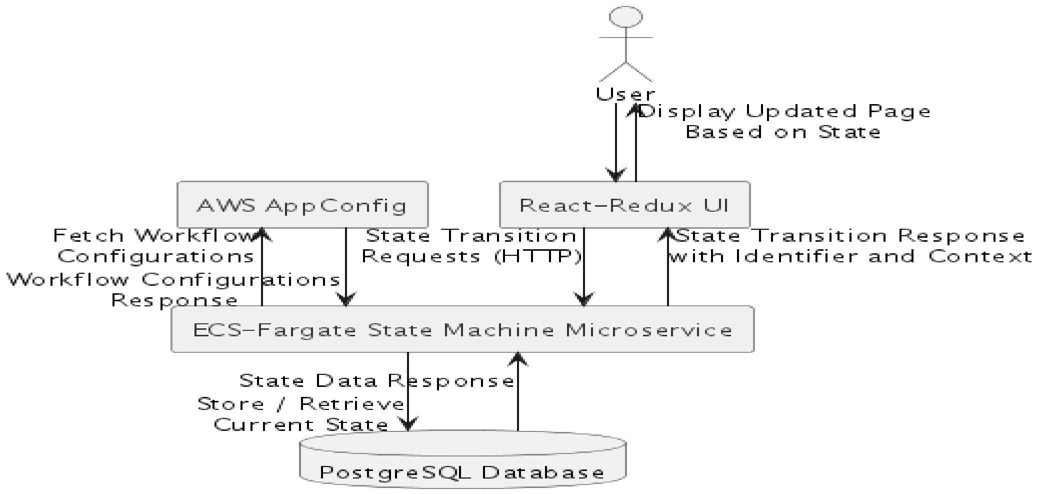
User
Display Updated Page Based on State
AWS AppConfig
React-Redux UI
Fetch Workflow Configurations
Workflow Configurations Response
State Transition Requests (HTTP)
State Transition Response with Identifier and Context
ECS-Fargate State Machine Microservice
State Data Response
Store / Retrieve Current State
PostgreSQL Database

Fig 4: Implementation using AWS

3.1 AWS AppConfig Configuration Management:

AWS AppConfig, a part of the AWS Systems Manager, is utilized to store and manage workflow configurations. Each workflow is represented by a unique configuration file that defines the sequence of skills, conditions, and transitions between states. These configurations are stored in an S3 bucket [8], and AWS AppConfig is used to manage versions of these configurations, allowing for easy updates and rollback in case of errors. Setting up AWS AppConfig begins with creating an application that will host the workflow configurations. This application serves as a container for different environments (e.g., development, staging, production) and configuration profiles. Each configuration profile corresponds to a different workflow and contains the relevant JSON or YAML [7] file that details the sequence of operations. For example, a configuration for an outbound logistics workflow would include the necessary steps like “Container Scan”, “Pallet Scan”, “Labeling”, and so on. AWS AppConfig’s deployment strategies ensure that configuration changes can be applied gradually, making it possible to test new configurations with a subset of the fleet before rolling them out to the entire application.

3.2 ECS-Fargate Microservice for State Machine:

The core logic for managing state transitions is encapsulated in a microservice deployed on AWS ECS using the Fargate launch type. This microservice acts as the state machine orchestrator and is responsible for consuming the workflow configurations from AWS AppConfig, interpreting them, and managing state transitions based on external events or user actions. The state machine microservice is designed to be stateless [10] itself, relying on external systems like PostgreSQL to maintain the state of each workflow. This stateless nature allows the microservice to scale horizontally with ease, handling a high volume of concurrent state transition requests. The microservice exposes a set of RESTful [9] APIs that the React-Redux front-end interacts with to initiate state transitions, query the current state, and update workflow status. When a state transition request is received, the microservice fetches the appropriate workflow configuration from AWS AppConfig using the workflow identifier. It then uses the current state and event information (e.g., "scan_complete") to determine the next state based on the configuration. If the transition is valid, it updates the current state in the PostgreSQL database and returns the new state identifier along with any relevant context to the UI. This context may include additional instructions or metadata needed by the UI to render the new page correctly.

3.3 PostgreSQL Database for State Management:
A PostgreSQL database is used to persist the state of each workflow instance. Each workflow is uniquely identified by a workflow ID, and its current state is stored in a dedicated table that tracks all active workflows in the system. The table schema includes fields like workflow_id, current_state, previous_state, created_at, and updated_at. This setup allows the state machine microservice to query the current state of a workflow at any time, supporting robust state management and recovery mechanisms. When the state machine microservice receives a state transition request, it first checks the PostgreSQL database to retrieve the current state. After processing the transition, it updates the state information in the database to reflect the new state and associated metadata. The use of PostgreSQL ensures that the system can handle complex queries and transactions, making it a reliable choice for managing state information that needs to be queried or updated frequently. Additionally, PostgreSQL's support for ACID transactions [11] guarantees that state transitions are consistent, isolated, and durable, even in the case of failures.

3.4 React-Redux Front-End for State Visualization and Control:
The front-end is built using React for UI rendering and Redux for state management. The UI serves as the primary interface through which users interact with the system, displaying different screens based on the current state of the workflow. The UI Flow Controller in the front-end makes HTTP requests to the ECS-Fargate microservice to trigger state transitions based on user actions, such as completing a container scan or resolving a problem with an item. When the front-end receives a state transition response from the microservice, it updates the Redux store with the new state identifier and any accompanying context information. This allows the UI components to react dynamically, rendering the appropriate pages and forms based on the new state. For example, if the state transition response indicates that the new state is LABELING, the front-end will switch to the labeling page, displaying fields and options that are relevant to this state. The use of React-Redux enables a clear separation between the UI and the application state, making it easier to manage complex UI logic and ensuring that state changes propagate through the application consistently.

3.5 System Interaction and Workflow Execution:
The complete workflow execution involves a series of interactions between these components, ensuring smooth state transitions and dynamic configuration management. When a user initiates an action, such as starting a new workflow or completing a task, the React-Redux UI sends a request to the ECS-Fargate microservice to initiate the state transition. The microservice fetches the workflow configuration from AWS AppConfig and determines the appropriate state

transition based on the configuration rules. It then updates the PostgreSQL database with the new state information and sends a response back to the UI with the updated state and context. The UI updates its Redux store and renders the appropriate page based on the new state, providing a seamless experience to the user. This entire process is repeated for each state transition until the workflow is completed.

The combination of AWS AppConfig, ECS-Fargate, PostgreSQL, and React-Redux offers a highly modular, scalable, and maintainable solution for managing complex workflows in a WMS. Each component is responsible for a specific function within the system, and the interactions between them are well-defined, making the overall architecture robust and flexible. This architecture can be extended to support additional use cases, new skills, or external system integrations without significant changes to the core components, making it a future-proof solution for dynamic warehouse management.

IV. BENEFITS OF THE STATE MACHINE-DRIVEN APPROACH

The state machine-driven approach for Warehouse Management Systems offers dynamic workflow management by enabling easy configuration and reconfiguration of workflows based on business requirements without altering the core codebase. This is achieved through external configuration files that specify the sequence of skills required for each use case, allowing rapid adaptation of workflows in response to changing demands, new product introductions, or shifts in logistics strategies. As a result, developers can reduce the development overhead typically associated with code changes, making it easier to test and implement new workflows.

The proposed architecture also simplifies the codebase by eliminating complex branching logic and consolidating decision-making into configuration files. This shift results in a more maintainable and understandable codebase that reduces the risk of introducing errors during updates. By eliminating the need for intricate conditional statements, the code becomes clearer, easier to troubleshoot, and less prone to bugs. Additionally, the system is easier to document, making onboarding new developers and maintaining the system more straightforward.

The UI consistency is greatly enhanced through the state-driven architecture, ensuring that the interface presented to the user always reflects the current state of the workflow. The State Transition Manager sends a state identifier and relevant context to the UI Flow Controller, which then dynamically renders the appropriate page or component based on the state. This approach guarantees real-time UI updates and provides context-aware elements, reducing the chance of user errors and ensuring a smooth user experience.

Furthermore, the state machine-driven approach provides significant flexibility for workflow management. Administrators can define new workflows or modify existing ones by simply updating configuration files without changing the application logic. This allows for custom workflows to be created for specific warehouses or use cases and enables new skills to be integrated seamlessly into the system. This flexibility extends to managing different requirements across multiple warehouses, providing a scalable solution for large organizations.

The modular nature of the state machine-driven architecture supports better testability. Each skill can be developed, tested, and deployed independently, which streamlines the testing process and enhances the system's overall robustness. Complete workflows can be tested by simulating state transitions and verifying behavior based on configurations. This approach simplifies automated testing, making it easier to validate new workflows and changes without manual intervention.

The proposed architecture also facilitates scalability and performance optimization by allowing skill execution to be distributed based on state transitions. Asynchronous state transitions and horizontal scaling capabilities enable the system to handle high-volume workflows more efficiently. States can be executed on separate services or nodes, optimizing resource utilization and ensuring that the system can scale as needed to meet demand.

The architecture integrates seamlessly with external systems, such as Inventory Management Systems (IMS) and Transport Management Systems (TMS). Clear boundaries between internal WMS logic and external systems enable smooth data exchanges and the triggering of external actions based on state transitions. This integration is facilitated by the State Transition Manager, which communicates state identifiers and context to ensure real-time synchronization across the supply chain.

Robust error handling and state recovery mechanisms are inherent in the state machine-driven approach, ensuring that the system can gracefully manage issues without human intervention. In case of errors, the state machine can revert to a previous state, pause and notify administrators, or automatically resolve known issues using the context provided by the State Transition Manager. This comprehensive error management capability ensures that workflows continue with minimal disruption and that issues are quickly addressed when they occur.

V. CONCLUSION

The state machine-driven architecture presented in this paper demonstrates a robust and scalable approach to handling diverse and complex workflows within Warehouse Management Systems (WMS). By decoupling workflow configurations from the underlying code through the use of AWS AppConfig, the proposed solution enables seamless adjustments and new workflow integrations without altering the core system. The introduction of a state machine hosted as a microservice on AWS ECS-Fargate ensures that state transitions are managed efficiently and consistently, while a PostgreSQL database maintains persistent state information across multiple workflows and sessions, supporting robust state management and error handling.

This architecture's separation of concerns allows each component to perform its role independently, enhancing modularity and reducing overall system complexity. The use of React-Redux on the front-end enables dynamic user interface updates based on real-time state changes, providing an intuitive and responsive user experience. The state-driven nature of the UI, combined with context-sensitive rendering, ensures that users always have the relevant information and options for each stage of the workflow, reducing errors and improving operational efficiency.

Moreover, the implementation is highly scalable and adaptable, making it ideal for organizations with large and distributed warehouse operations. The architecture can handle various use cases, ranging from inbound and outbound logistics to inventory management and problem-solving workflows, by simply updating the configuration files in AWS AppConfig. This eliminates the need for extensive code modifications, thereby reducing the risk of introducing bugs and minimizing downtime during updates.

Overall, the proposed state machine-driven architecture offers a future-proof solution that aligns with modern software engineering principles such as modularity, separation of concerns, and configuration-driven behavior. It allows organizations to quickly adapt to changing business requirements, optimize workflows, and maintain a high level of operational efficiency, making it a valuable framework for the next generation of Warehouse Management Systems.

REFERENCES

1. Jain, V. Karthikeyan, S. B, S. BR, S. K, and B. S, "Demand Forecasting for E-Commerce Platforms," 2020 IEEE International Conference for Innovation in Technology (INOCON), Bangluru, India, 2020, pp. 1-4, doi: 10.1109/INOCON50539.2020.9298395.
2. F. B. Schneider, "The state machine approach: A tutorial," in Fault-tolerant distributed computing, 2005, pp. 18-41.
3. AWS, "What is AWS AppConfig?" Available: https://docs.aws.amazon.com/appconfig/latest/userguide/what-is-appconfig.html.
4. AWS, "AWS Fargate," Available: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/AWS_Fargate.html.
5. PostgreSQL Global Development Group, "PostgreSQL," Available: https://www.postgresql.org/.
6. Redux.js, "Redux Documentation," Available: https://redux.js.org/.
7. M. Eriksson and V. Hallberg, "Comparison between JSON and YAML for data serialization," The School of Computer Science and Engineering Royal Institute of Technology, 2011, pp. 1-25.
8. AWS, "Amazon S3," Available: https://aws.amazon.com/s3/.
9. L. Richardson and S. Ruby, RESTful Web Services. O'Reilly Media, Inc., 2008.

10. A. Shieh, A. C. Myers, and E. G. Sirer, "A stateless approach to connection-oriented protocols," ACM Transactions on Computer Systems (TOCS), vol. 26, no. 3, pp. 1-50, 2008.
11. M. Little, "Transactions and web services," Communications of the ACM, vol. 46, no. 10, pp. 49-54, 2003.