AWS COBOL Card Demo to Java Migration Plan

Oscar Alvarez

2025

Table of Contents

AWS COBOL CARD DEMO TO JAVA MIGRATION PLAN 1

1. Abstract 3
2. Introduction 4
2.1. Project Overview 4
2.2. Background and Motivation for Modernization 7
3. Legacy System Analysis 8
3.1. COBOL 8
3.2. CICS 9
3.3. VSAM 11
3.4. JCL 13
3.5. RACF 14
4. Modern Java-based Technology Stack 16
4.1. Architecture Design 16
4.2. Framework Selection 18
4.3. Integration Approach 20
5. Migration Roadmap 23
5.1. Data Migration Strategy 23
5.2. Business Logic Translation 26
5.3. Integration Points 28
5.4. Testing Approach 29
6. Conclusion 30
6.1. Summary of Migration Plan 30
6.2. Benefits of Modernization 32

1. Abstract

This migration initiative is driven by the need to modernize the *Card Demo* application, a financial transaction management system, and enhance its scalability, maintainability, and operational efficiency. The current COBOL-based system faces challenges such as high maintenance costs, limited developer availability, and difficulty adapting to modern architectures.

Key highlights of the modernization plan include:

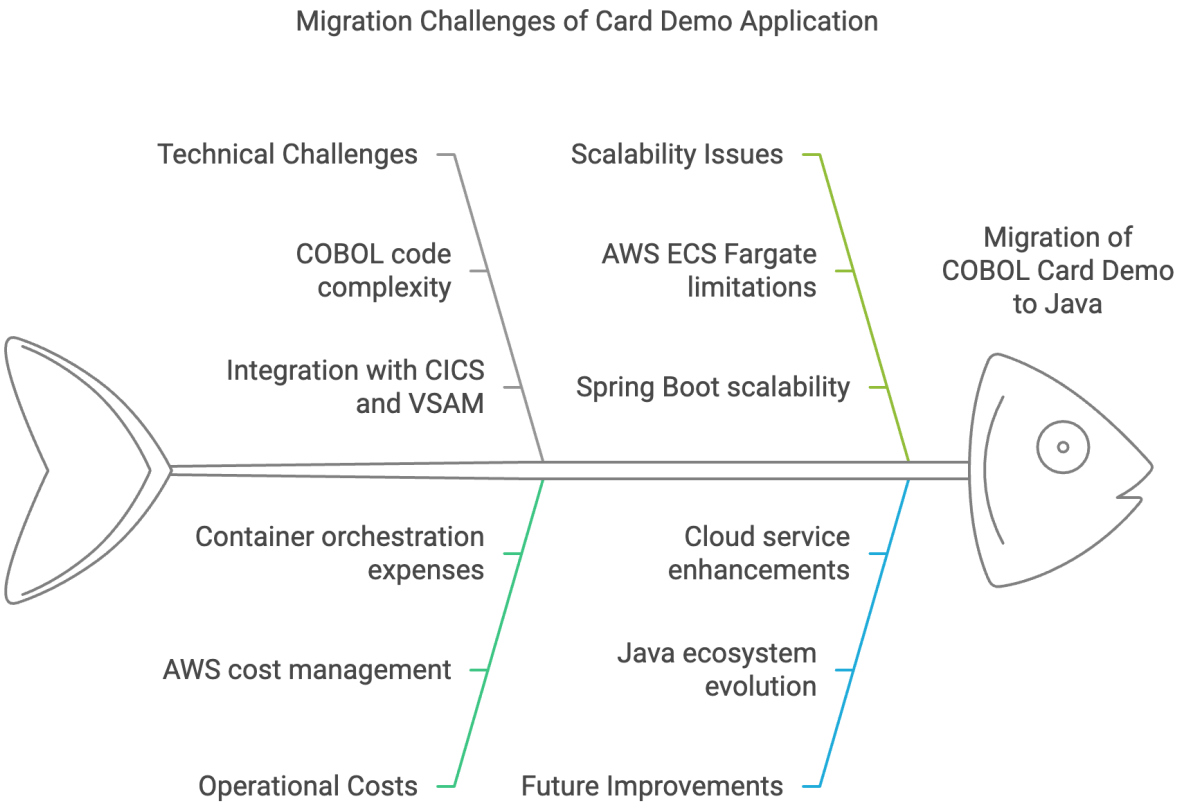
1. **Core Features of Card Demo**: The application supports transaction listing, detailed views, additions, and validation, leveraging COBOL programs and VSAM files.
2. **Rationale for Modernization**: Transitioning to Java addresses COBOL's limitations while utilizing robust frameworks like Spring Boot to support microservices and cloud compatibility. Java's widespread developer base ensures cost-effective development and maintenance.
3. **Target Architecture**:
    - **Frontend**: Modernized with React for dynamic user interfaces.
    - **Backend**: Built with Spring Boot to support robust business logic, RESTful APIs, and integration with AWS services.
    - **Data Layer**: Migrating from VSAM to a relational database (PostgreSQL) for efficient, structured data management.
    - **Deployment**: Containerized with Docker and orchestrated using AWS ECS Fargate for serverless scalability.
4. **Migration Approach**:
    - **Data Migration**: Leveraging tools like AWS Glue and DMS for secure and consistent transformation from VSAM to PostgreSQL.
    - **Business Logic Conversion**: Combining automated COBOL-to-Java translation and manual refactoring to align with modern coding standards.
    - **Integration**: Establishing secure, efficient connections to external systems using AWS services and REST APIs.
5. **Testing and Validation**: Comprehensive testing strategies, including unit, integration, and system tests, ensure functional reliability and performance in production environments.
6. **Strategic Benefits**:
    - Reduced operational costs by eliminating mainframe reliance.

- Improved scalability and system performance through containerized microservices.
- Enhanced security and compliance via AWS Cognito and CloudWatch.
- Accelerated development cycles using modern development tools and CI/CD pipelines.

2. Introduction

2.1. Project Overview

The purpose of this document is to perform an analysis and generate a complete work plan for the migration of the Card Demo application written in COBOL and deployed on mainframe to a modern platform updated in Java with Spring Boot based on containers, using in a first phase using AWS ECS FARGATE for orchestration and towards a cloud infrastructure like AWS with other services. The goal is to allow scalability, maintenance, and future improvements to the application to be made with lower operating costs.



Key Features of CardDemo.

The **CardDemo** application provides a set of functionalities for managing financial transactions. These are:

1. **List Transactions**:
   - Allows users to view a list of transactions stored in the TRANSACT file.
   - Is used to consult previous transactions and navigate through results pages.
2. **View a Specific Transaction**:
   - Provides details of a selected transaction, such as:
     - Transaction ID.
     - Amount.
     - Original date.
     - Processing date.
     - Type and category code.
     - Merchant details (name, city, postal code).
   - Allows validating specific information of an operation.
3. **Add a New Transaction**:

- Allows the user to create a new transaction by entering data such as:
  - Account or card ID.
  - Amount.
  - Description.
  - Dates of origin and processing.
  - Merchant details.
- The application validates that the entered data is correct and stores it in the TRANSACT file.

4. **Data Validation**:

   - Verifies that the entered fields are valid (correct format, mandatory fields not empty, etc.).
   - Performs cross-validation with auxiliary files:
     - **CXACAIX**: Validates the relationship between accounts and cards.
     - **CCXREF**: Checks data related to cards.
5. **Page Navigation**:
   - Allows moving forward and backward in paginated lists of transactions.

User Interaction

- The application uses **CICS (Customer Information Control System)** for managing interactive screens.
- Each interaction with the user is carried out through **CICS maps**, which represent user interfaces.

General Workflow

1. The user accesses an initial screen that displays a list of transactions.
2. From this list, the user can:
   - Select a transaction to view it in detail.
   - Add a new transaction.
   - Navigate between pages of transactions.
3. If adding a transaction is selected:
   - A screen is presented to enter the data.

- The data is validated.
- If it is correct, it is stored in the TRANSACT file.
- If there are errors, a message is displayed on the screen.

4. If a specific transaction is selected:
   - A screen is displayed with the complete details of the selected transaction.

VSAM Files Used

The application interacts with several files to store and retrieve data:

1. TRANSACT:
    - Main file that stores financial transactions.
2. CXACAIX:
    - Auxiliary index to validate relationships between account IDs and card numbers.
3. CCXREF:
    - Auxiliary file to validate cards and their associated accounts.
4. ACCTDAT:
    - Additional file that can store information related to accounts.

Technical Functionality

Each functionality is implemented as an independent COBOL program within the system:

1. COTRN00C (List Transactions):
    - Lists the transactions stored in the TRANSACT file.
    - Allows for page navigation.
2. COTRN01C (Visualize Transaction):
    - Displays the details of a specific transaction selected by the user.
3. COTRN02C (Add Transaction):
    - Validates and adds a new transaction to the TRANSACT file.

2.2. Background and Motivation for Modernization

While it is true that applications written in COBOL enjoy an initial advantage due to their extremely high performance and mathematical precision with fixed-point decimal values, currently Java would allow to solve this precisely with BigDecimal data types, also making some adjustments with the JIT compiler, performance can be optimized. Another major motivation for migrating to Java is indeed the human resources; it's no secret there are many more developers in Java than in COBOL which would allow for cost optimization in development, deployments and maintenance than with systems built entirely in COBOL.

3. Legacy System Analysis

3.1. COBOL

3.1.1. Purpose and Role

COBOL (Common Business-Oriented Language) is a programming language specifically designed for business applications and financial systems. Its purpose in the Card Demo application is to handle the business logic for the opening of VSAM files, displaying screens, capturing user data and processing the information.

3.1.2. Key Characteristics and Constraints

 Key Characteristics:

- Language oriented to human reading, with a simple and descriptive syntax.
- Optimized for batch processing and financial transactions.
- High precision in numeric calculations through decimal arithmetic.
- Tight integration with mainframe systems and legacy databases.

 Limitations:

- Difficulty adapting to modern architectures and distributed environments.
- Lack of support for modern concepts such as object-oriented programming and microservices.
- Lack of developers with experience in COBOL due to its age.

3.1.3. Migration Challenges

Migrating COBOL applications to modern technologies faces several challenges:

- **Dependency on Legacy Systems**: The CardDemo application is integrated with legacy databases such as VSAM.
- **Code Volume**: The CardDemo application has quite a few scripts and maps responsible both for business logic, as well as user interface handling and data captures. Big issue for decoupling.
- **Accuracy and Performance**: Preserving the accuracy in financial calculations and the performance of batch processing can be tricky in modern languages like Java.

- **Costs and Time**: Migration can be costly and time-consuming due to the need for exhaustive testing and functionality adjustment.

3.1.4. Modern Equivalent: Java

Java is a modern alternative for replacing COBOL applications due to its flexibility, portability and ability to handle modern enterprise architectures. It is suitable for distributed systems, web applications and cloud environments.

3.1.4.1. Justification for Java

 Portability: Java is platform-independent, which allows applications to run in multiple environments.

 Modern Ecosystem: It offers a wide range of frameworks and tools, such as Spring Boot, to accelerate development.

 Precision: With the BigDecimal class, Java ensures precision in financial calculations.

 Scalability: It is compatible with modern architectures like microservices, which facilitates scalability.

 Community and Support: The large community of developers and ongoing support make Java a robust option to replace COBOL.

3.2. CICS

3.2.1. Purpose and Role

CICS (Customer Information Control System) is a transactional application server used mainly in mainframe systems. Its purpose is to manage large volumes of real-time transactions efficiently, which makes it essential for critical applications in banking, insurance, retail, and government sectors. CICS allows the simultaneous execution of multiple transactions with high availability and reliability.

3.2.2. Key Characteristics and Constraints

 Key Features:

- Efficient handling of concurrent transactions.
- High availability and fault recovery.
- Tight integration with legacy databases (VSAM).

- Support for multiple programming languages, including COBOL, PL/I, and assembler.
- Robust security through detailed access controls.

 Constraints:

- Dependence on mainframe platforms, which hinders portability.
- Less intuitive interfaces and tools compared to modern technologies.
- High operation and maintenance costs in legacy environments.

3.2.3. Migration Challenges

 Complexity of Architecture: The CICS applications being highly integrated with other mainframe systems and legacy databases, which makes the separation of components difficult.

 Transaction Rewriting: Transactions programmed in COBOL or other languages must be rewritten to adapt to modern frameworks.

 Performance: Ensuring comparable performance on distributed systems can be a challenge, especially for high loads.

 Security: Migrating robust CICS security configurations to a modern system requires careful planning.

 Training Costs: Development teams need training in modern technologies such as Spring Boot to ensure a successful transition.

3.2.4. Modern Equivalent: Spring Boot

Spring Boot is a modern alternative to CICS for the development of transactional and enterprise applications. This framework simplifies the creation of production-ready Java applications, compatible with modern architectures such as microservices and cloud environments.

3.2.4.1. Justification for Spring Boot

 Flexibility: Allows the development of transactional and scalable applications using microservices or modern monolithic architectures.

 Support for Transactions: With **Spring Transaction Management**, distributed transactions can be efficiently implemented.

 Scalability and Cloud Deployment: Compatible with cloud platforms like AWS.

 Integration with Databases: Provides tools like Spring Data JPA to interact with modern and legacy databases.

 Productivity: Simplifies configuration and development with features such as auto-configuration and pre-integrated libraries.

 Community and Ecosystem: Wide community support and a large number of available extensions to integrate customized solutions.

 Cost Reduction: Allows eliminating the reliance on mainframes and reducing long-term operational costs.

3.3. VSAM

3.3.1. Purpose and Role

VSAM (Virtual Storage Access Method) is a data storage system primarily used in mainframe environments. Its purpose is to manage and provide efficient access to sequential and key-organized data files, commonly used in transactional and financial applications. It is an essential technology in legacy systems for storing large volumes of structured data.

3.3.2. Key Characteristics and Constraints

 Key Features:

- Support for different types of files, such as KSDS (Key-Sequenced Data Set), ESDS (Entry-Sequenced Data Set) and RRDS (Relative Record Data Set).
- Optimization for reading and writing operations in large data volumes.
- Direct integration with COBOL and CICS applications.
- Support for indexing in primary keys for quick access to records.

 Limitations:

- Dependence on mainframe environments, which hinders portability.
- Lack of flexibility for complex queries, compared to relational databases.
- Limitations in scalability for distributed architectures.
- Costly maintenance and dependence on specialized personnel.

3.3.3. Migration Challenges

 Data Conversion: Migrating the data stored in VSAM structures (like KSDS) to relational databases requires careful mapping in order to preserve integrity and relationships.

 Business Logic Transformation: VSAM-based logic, including direct manipulation of files, must be rewritten to to use SQL operations.

 Compatibility with Existing Applications: Many applications are specifically designed to interact with VSAM, which makes the transition difficult without affecting functionality.

 Size and Complexity: Large data volumes and the customized structures in VSAM complicate the migration process.

 Exhaustive Testing: It is necessary to carry out tests to ensure that operations in relational databases produce results equivalent to those of VSAM.

3.3.4. Modern Equivalent: Relational Database (e.g., PostgreSQL)

Modern relational databases, such as **PostgreSQL**, are ideal alternatives to replace VSAM systems. These databases allow to store, manage and query structured data efficiently and offer advanced scalability, portability and analysis capabilities.

3.3.4.1. Justification for Relational Database

 Advanced Query Capability: Relational databases support SQL, which allows complex queries and data analysis that are difficult or impossible in VSAM.

 Horizontal and Vertical Scalability: Databases like PostgreSQL can efficiently scale on modern distributed architectures.

 Data Integrity: They provide native support for primary key, foreign keys and constraints, ensuring referential integrity.

 Flexibility: They are compatible with a wide range of programming languages and modern frameworks.

 Tool Ecosystem: Tools such as ORMs (Object-Relational Mapping), ETL and data analysis seamlessly integrate relational databases into business workflows.

 Cloud Support: Databases like PostgreSQL are fully integrated into cloud services such as AWS RDS.

3.4. JCL

3.4.1. Purpose and Role

JCL (Job Control Language) is a scripting language used in mainframe systems to define and control the execution of tasks (jobs). Its main purpose is to manage the execution of programs, assign system resources (such as files and memory), and monitor workflow in batch and transactional environments. It is widely used in critical operations, like massive data processing and scheduled jobs.

3.4.2. Key Characteristics and Constraints

 Key Features:

- Specialized language for handling batch jobs and repetitive tasks in mainframes.
- Total control over the allocation of resources such as datasets, memory, and processors.
- Support for complex workflows through the definition of dependencies between tasks.
- Native integration with other mainframe components, such as CICS and VSAM.

 Constraints:

- Steep learning curve due to its cryptic and system-oriented syntax.
- Reliance exclusively on mainframe environments, making it difficult to port to modern platforms.
- Lack of flexibility to adapt to modern architectures or integrate with current development tools.

3.4.3. Migration Challenges

 Script Rewriting: Migrating JCL scripts to modern tools requires rewriting job definitions and resource assignments in a compatible format.

 Workflow Complexity: Workflows in JCL can be highly complex, making their direct translation to modern tools difficult.

 Legacy Resource Dependency: JCL is deeply integrated with legacy systems and files, such as VSAM and CICS, requiring a clear strategy to replace these dependencies.

 Error Management: Modern applications must replicate the error control and recovery mechanisms that JCL offers.

 Training Costs: Teams must be trained to use modern build automation and workflow tools.

3.4.4. Modern Equivalent: Build automation tools (e.g., Maven, Gradle)

Build automation tools like **Maven** and **Gradle** are modern equivalents that replace JCL functionality in contemporary development environments. These tools allow defining, managing, and automating build workflows, tests, and deployment in distributed and cloud-based applications.

3.4.4.1. Justification for Build Automation Tools

- **Standardization**: Maven and Gradle are widely used in the industry, allowing standardization of build and deployment processes.
- **Flexibility**: They offer compatibility with multiple programming languages and platforms, in addition to integration with version control systems and CI/CD tools (like Jenkins or GitLab CI).
- **Portability**: They allow workflows to be run in any environment (local, server, or cloud), eliminating dependence on mainframes.
- **Time Optimization**: They automate repetitive tasks such as compilation, testing, packaging, and deployment.
- **Ease of Configuration**: They provide simple declarative configurations through files such as pom.xml (Maven) or build.gradle (Gradle).
- **Plugin Ecosystem**: Extensive support for plugins that expand their functionality (for example, for testing, reporting generation, and dependency management).
- **Scalability**: They can handle small or large projects with thousands of modules, while maintaining high efficiency.
- **Cost Reduction**: They decrease the need for mainframe experts and reduce time spent on manual tasks.

3.5. RACF

3.5.1. Purpose and Role

RACF (Resource Access Control Facility) is an access control system integrated in IBM z/OS mainframes. Its purpose is to protect the resources of the mainframe system (such

as datasets, programs, and commands) through the management of users, permissions, and roles. RACF ensures that only authorized people and processes can access sensitive resources, maintaining security and regulatory compliance.

3.5.2. Key Characteristics and Constraints

 Key Features:

- Centralized management of users and permissions in mainframe systems.
- Role and resource-specific user authentication and authorization.
- Detailed audits to record access and activities, essential for compliance with regulations such as SOX or GDPR.
- Native integration with z/OS system components and applications such as CICS and DB2.
- Support for advanced security policies, including encrypted passwords and dynamic access rules.

 Constraints:

- Exclusive to mainframe systems, making it difficult to integrate with distributed systems or modern cloud environments.
- Complex configuration and maintenance that require specialized personnel.
- Rigidity in adapting to microservice architectures and hybrid environments.

3.5.3. Migration Challenges

 Policy Reconfiguration: RACF access policies should be mapped to modern identity management systems, maintaining security and granularity of permissions.

 Legacy Resources Compatibility: Many resources protected by RACF are deeply integrated into mainframe systems.

 Operational Complexity: Migrating users, roles, and permissions from a centralized system like RACF to a distributed IAM system can be laborious.

 Regulatory Compliance: It is necessary to ensure that modern systems retain the required controls and audits to comply with legal regulations.

 Training and Support: Teams need training in the new IAM solution to manage roles, users, and policies efficiently.

3.5.4. Modern Equivalent: Identity and Access Management (IAM) solution (AWS IAM)

AWS IAM (Identity and Access Management) is a modern solution for managing identities, permissions, and access in distributed and cloud-based environments. It's a suitable alternative to RACF that allows for the security management of resources both in the cloud and in hybrid applications.

3.5.4.1. Justification for IAM solution

 Scalability: AWS IAM is designed to handle environments with thousands of users and distributed resources, which allows its use in modern and global architectures.

 Portability: It can protect resources in multiple services and applications, eliminating exclusive dependence on mainframes.

 Detailed Access Management: Offers granular control through JSON policies to define specific permissions.

 Integration with Modern Services: AWS IAM integrates with cloud services such as S3, DynamoDB, Lambda and others, as well as being compatible with hybrid applications.

 Audits and Security: Includes advanced tools such as AWS CloudTrail and AWS Config to monitor and audit access.

 Multi-user and Multi-role Support: Allows managing identities for users, services and devices, as well as supporting federated authentication (SSO) and MFA.

 Flexibility and Adaptation: It is compatible with modern architectures such as microservices and serverless applications.

 Cost Reduction: Replacing RACF with an IAM solution reduces the need to maintain exclusive mainframes for security.

4. Modern Java-based Technology Stack

4.1. Architecture Design

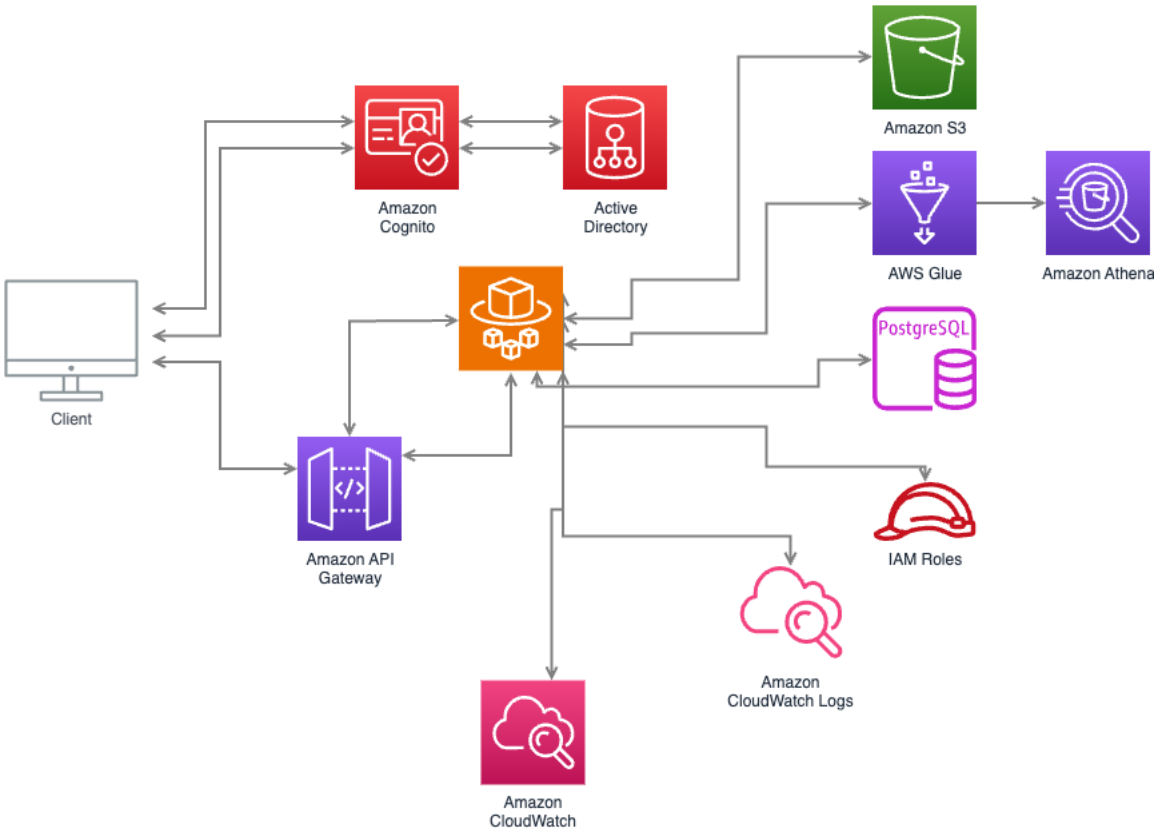
The architectural design of a Java-based technology stack focuses on leveraging the capabilities of modern frameworks, automation tools, and scalable design principles to build resilient, secure, and efficient systems. This approach is ideal for migrations from legacy systems like COBOL and CICS to modern environments.

4.1.1. High-level System Architecture

The high-level architecture for a modern technology stack based on Java includes:

- Frontend:
  - A modern web framework like React to provide responsive and dynamic user interfaces.
  - Communication with the backend via REST API.
- Backend:
  - A framework like Spring to build robust enterprise applications.
  - Implementation of microservices for greater modularity and scalability.
  - Integration with messaging systems, for example, RabbitMQ.
- Database:
  - Use of modern relational databases like PostgreSQL for structured data storage.
  - Supplementing with NoSQL databases like Redis for unstructured data or cache storage.
- Infrastructure:
  - Containers (Docker) and orchestrators (ECS FARGATE) for deployment and scaling.
  - Integration with cloud services like AWS.
- Security:
  - Use of modern IAM tools like AWS IAM for user and permission management.
  - Implementation of authentication and authorization with Spring Security and protocols like OAuth2.
- Automation and CI/CD:
  - Jenkins, GitLab CI for continuous integration and deployment.
  - Dependency and build management with Maven or Gradle.

4.1.2. Component Breakdown and Interactions



4.2. Framework Selection

4.2.1. Justification for Spring Boot

Spring Boot is selected as the main framework due to its native compatibility with AWS tools, allowing for the efficient construction and deployment of modern business applications. Key components of the solution include:

- **Ease of Configuration**:
  Spring Boot simplifies configuration through an auto configuration approach, allowing for easy integration with services such as **AWS RDS PostgreSQL** and **AWS Cognito**.

- **Cloud Compatibility**:
  It offers seamless integration with **AWS ECS Fargate** for server-less containers, reducing operational complexity, and enabling scalable, reliable deployment.

- **Identity and Access Management**:
  Integration with **AWS Cognito** allows for efficient user management,

authentication and authorization, leveraging features such as Single Sign-On (SSO) and MFA.

- **Scalable Database**:
  Support for **AWS RDS PostgreSQL** through **Spring Data JPA** ensures robust, scalable data management, whilst simplifying reading and writing operations.

- **Serverless Architecture with Fargate**:
  The combination of **Spring Boot** with **ECS Fargate** facilitates the running of containers without the need to manage servers, improving operational efficiency and reducing costs.

- **Spring Ecosystem**:
  Tools like **Spring Cloud AWS** optimize integration with AWS services, allowing you to take advantage of features like queues (SQS), storage (S3) and monitoring (CloudWatch).

- **Transaction Support**:
  With **Spring Transaction Management**, data consistency is ensured in critical operations on RDS PostgreSQL.

- **Communities and Resources**:
  An active community and detailed documentation make Spring Boot a backed and reliable solution, aligned with AWS services.

4.2.2. Additional Libraries and Frameworks

To implement the solution based on **Spring Boot** and **REST API**, with deployment in **AWS ECS Fargate** and managed containers in **AWS ECR**, the following libraries and frameworks are selected:

- **REST API Management**:
  - **Spring Web**: Main framework for creating RESTful endpoints with support for JSON and HTTP communication.
  - **Jackson**: Library for efficient serialization and deserialization between Java objects and JSON.

- **Database Access**:

- **Spring Data JPA**: Simplifies the interaction with **AWS RDS PostgreSQL** through a declarative approach for queries and transactions.

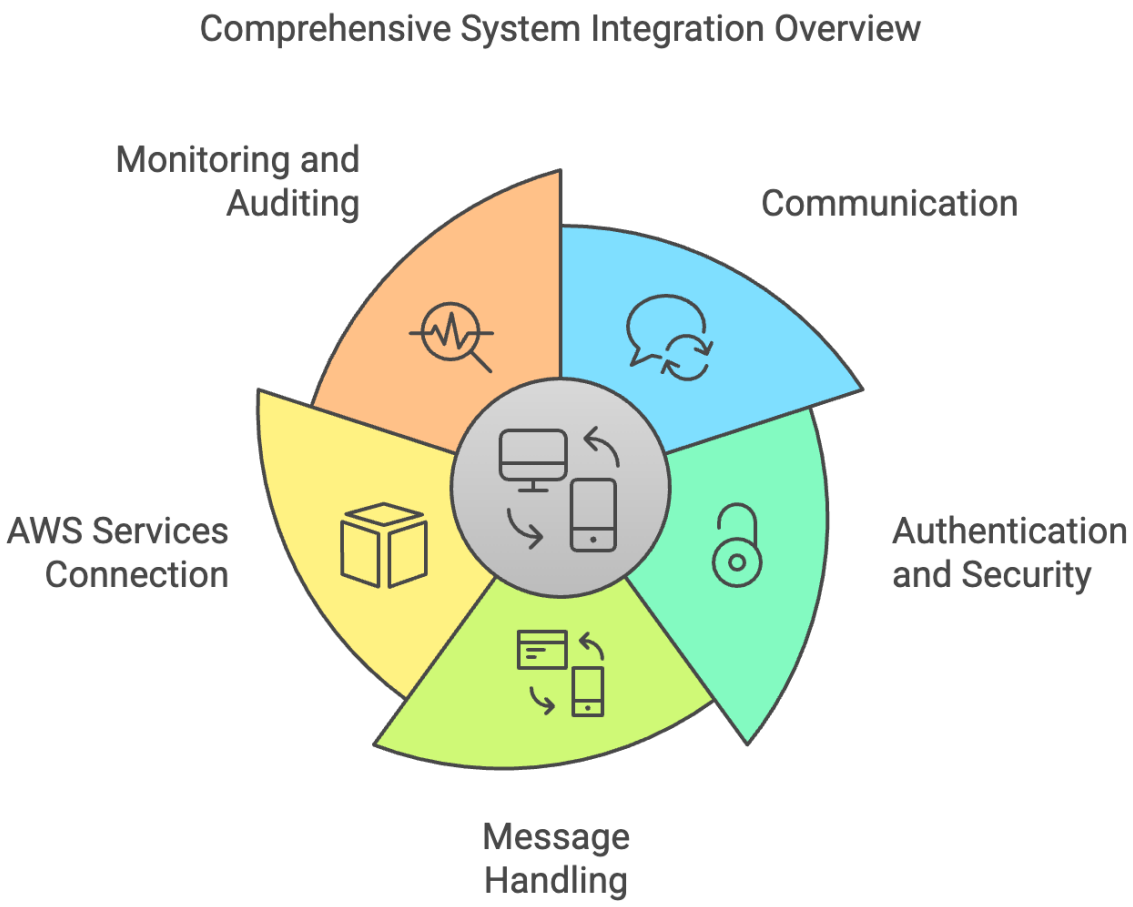
- **HikariCP**: High-performance connection pool that optimizes connections to the database.
- **API Documentation**:
  - **Springdoc OpenAPI**: Automatic generation of documentation in OpenAPI/Swagger format, facilitating the development and integration of APIs.
- **Security**:
  - **Spring Security**: To manage authentication and authorization at the REST endpoints.
  - **AWS Cognito SDK**: Direct integration with **AWS Cognito** for authentication, Single Sign-On (SSO) and MFA.
- **Containerization and Deployment**:
  - **Docker**: For packaging the application into containers.
  - **AWS SDK for Java**: To interact with AWS services, such as **ECR**, to publish and manage container images.
  - **Spring Cloud AWS**: To facilitate the configuration and connection with AWS services (such as ECR, RDS and CloudWatch).
- **Monitoring and Logs**:
  - **Spring Boot Actuator**: Exposure of metrics and real-time monitoring.
  - **ELK Stack**: To centralize logs and make their analysis easy.
  - **Prometheus and Grafana**: Optional integration for advanced metrics.
- **Testing**:
  - **JUnit 5**: Main framework for unit tests.
- **Testcontainers**: For integration testing using locally managed Docker containers.

4.3. Integration Approach

4.3.1. External System Integration Strategy

The strategy for integration with external systems is based on the implementation of **REST API** and specific AWS tools to ensure efficient, secure, and scalable communication. The main considerations include:

- Communication through REST API:
  - Use of Spring Web to create RESTful endpoints that facilitate interaction with external systems.
  - Data serialization in JSON format using Jackson to ensure interoperability.
- Authentication and Security:
  - Integration with AWS Cognito for user authentication and request authorization.
  - Implementation of OAuth2 and JWT (JSON Web Tokens) to protect REST endpoints.
- Message Handling:
  - Use of Amazon SQS or SNS for asynchronous messaging and communication between systems.
  - Optionally, integration with Apache Kafka for real-time data streams.
- Connection with AWS Services:
  - Client configuration using the AWS SDK for Java to interact with services such as S3, RDS, and ECR.
  - Use of Spring Cloud AWS to facilitate connection and configuration management with native AWS services.
- Monitoring and Auditing:
  - Implementation of AWS CloudWatch to monitor the activity of integrated systems and generate alerts.
  - Traceability of requests through Spring Boot Actuator and distributed tracing with AWS X-Ray.



4.3.2. Data Access Layer Design

The data access layer design uses best practices in abstraction and decoupling to ensure efficient, secure, and scalable access to **AWS RDS PostgreSQL**.

- **Main Framework**:
  - Use of **Spring Data JPA** to simplify interactions with the database and leverage features such as declarative queries and automatic transaction management.
- **Efficient Connections**:
  - Implementation of **HikariCP** as a connection pool to optimize resource use and enhance performance.

  - Dynamic connection configurations via **Spring Cloud AWS** for integration with **AWS RDS**.

- **Data Modeling Strategy**:
  - Mapping of entities through **Hibernate** to reflect relational structures in the database.
- **Transaction Management**:
  - Use of **Spring Transaction Management** to ensure data consistency in complex operations.
  - Configuration of distributed transactions in cases requiring interaction with multiple systems.
- **Testing and Validation**:
- Use of **Testcontainers** for integration tests with PostgreSQL instances in containers.
- Validation of queries using **JUnit 5** to ensure their correct operation.

5. Migration Roadmap

5.1. Data Migration Strategy

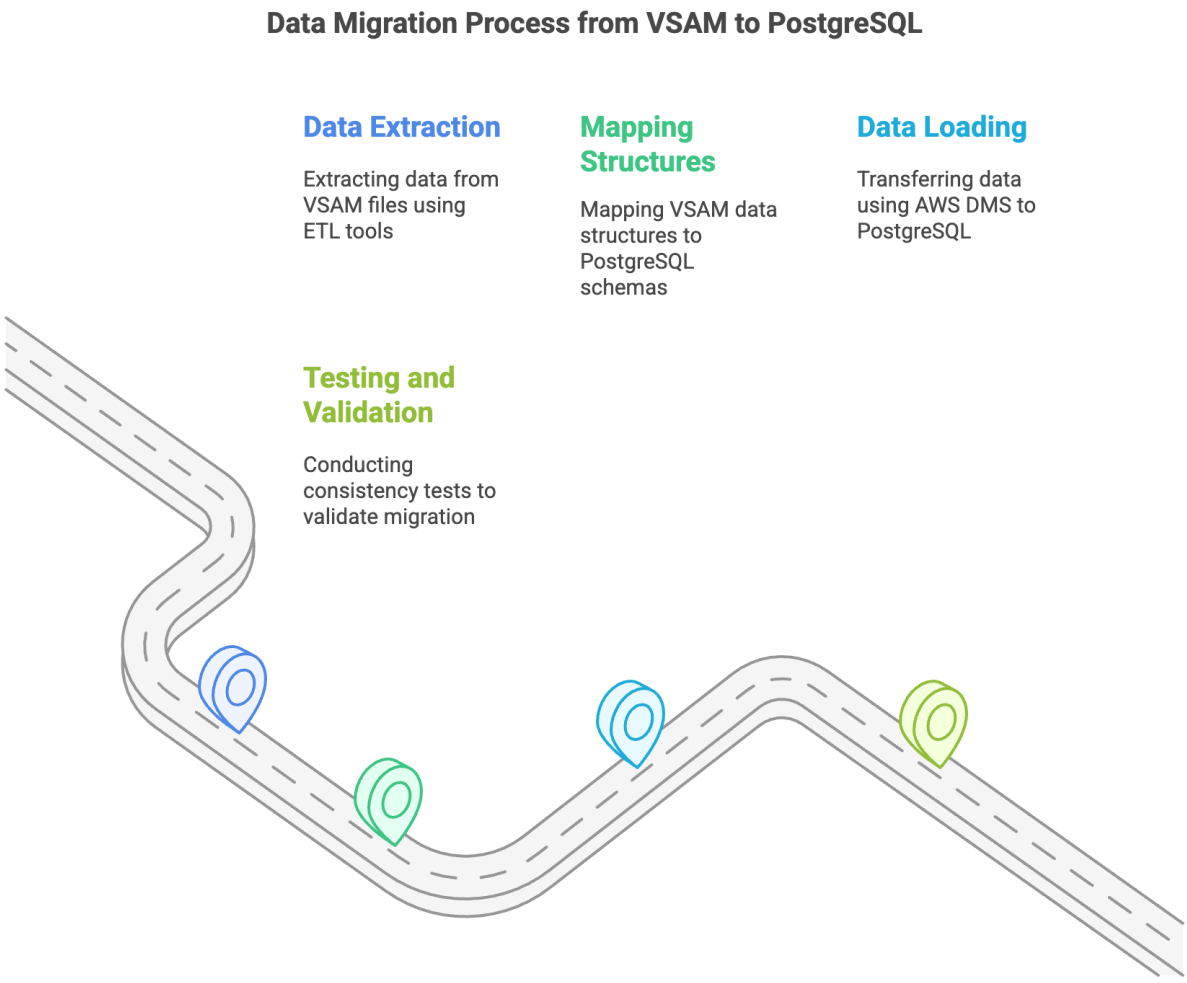
5.1.1. VSAM to Relational Database Migration Approach

The migration of data from VSAM to a relational database like **AWS RDS PostgreSQL** will be carried out through a structured approach that ensures the integrity and availability of the data. Key steps include:

- **Data Extraction**:
  - Use of ETL tools such as **AWS Glue** or custom scripts to extract data from VSAM files.
  - Implementation of native connectors to access VSAM datasets on the mainframe.
- **Mapping Structures**:
  - Identification of data structures in VSAM (KSDS, ESDS, RRDS) and their mapping to relational schemas in PostgreSQL.

  - Design of relational tables with primary and foreign keys to maintain relationships between the data.

- Data Loading:
  - Use of **AWS Database Migration Service (DMS)** to transfer data from VSAM to the relational environment.
  - Incremental verification to ensure that the data are migrated correctly without interrupting the legacy system.
- Testing and Validation:
  - Conducting consistency tests to ensure that the migrated data matches the original data.
  - Validating referential integrity in the relational database.



4.1.2 Data Transformation and Cleansing

The transformation and cleaning of data will be carried out to ensure that the migrated data is accurate, complete, and useful in the new environment:

- Transformation:
  - Converting legacy data types (packed decimal, zoned decimal) to standard relational formats.
  - Normalizing data to avoid redundancy and ensure an efficient structure.
- Cleaning:

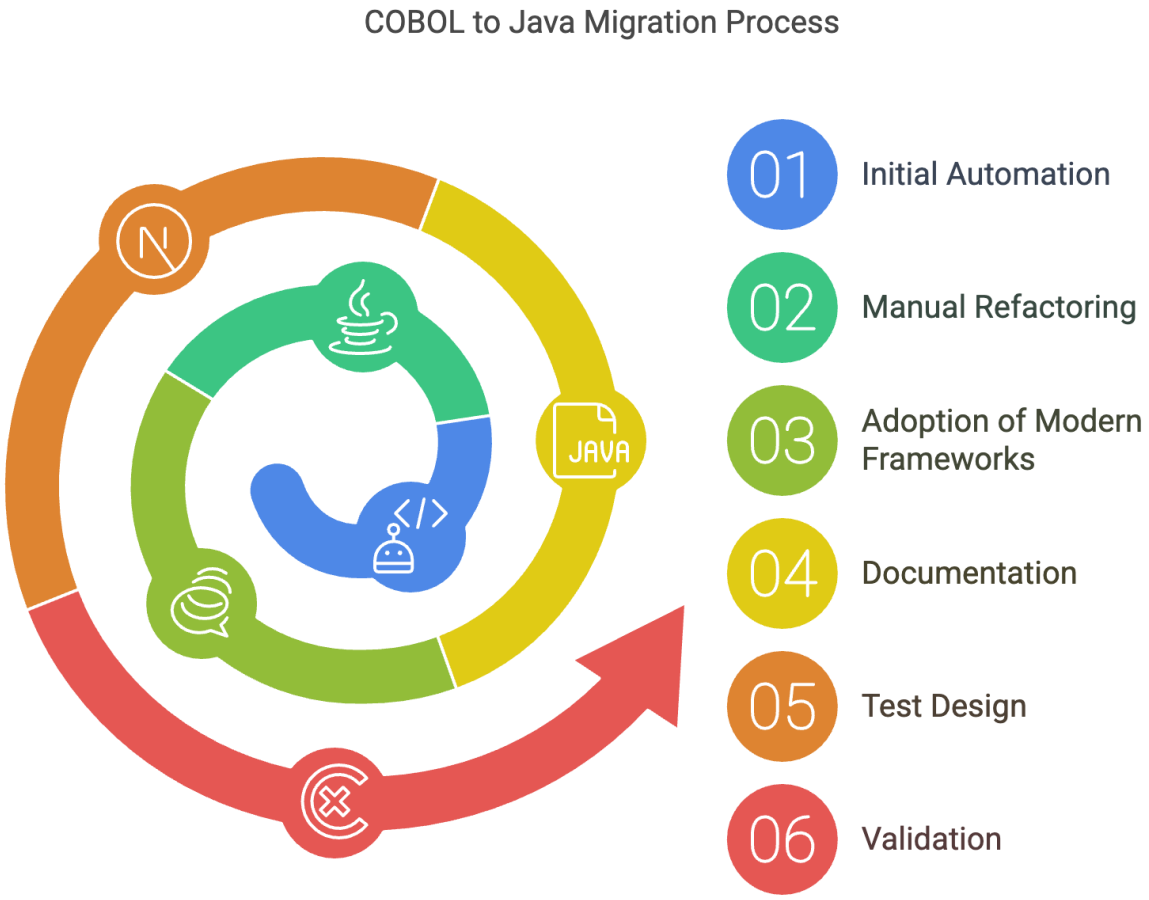
  - Removing duplicate records, obsolete data or inconsistencies detected during the migration process.
  - Using tools like **AWS Glue DataBrew** to identify and correct data quality issues.

5.2. Business Logic Translation

5.2.1. COBOL to Java Code Conversion Strategy

The conversion of business logic written in COBOL to Java will be carried out using a combination of automated and manual approaches:

- **Initial Automation**:
  - Using conversion tools like **Micro Focus Enterprise Analyzer** to automatically translate COBOL code to an initial foundation in Java.
- **Manual Refactoring**:
- Refactoring the converted code to align it with Java's best practices and Spring Boot capabilities.
- Modularizing the business logic into reusable classes and methods.
- **Adoption of Modern Frameworks**:
  - Use of **Spring Boot** to implement the translated business functionalities.
  - Integration of design patterns such as Dependency Injection to improve flexibility and maintenance.
- **Documentation**:
  - Creation of detailed documentation to map COBOL functions to their Java equivalents, facilitating the understanding and maintenance of the system.



5.2.2. Unit Testing of Migrated Business Logic

Unit tests are crucial to ensure that the migrated business logic functions as expected:

- Test Design:
  - Definition of test cases based on the original COBOL scenarios.
  - Use of tools such as **JUnit 5** and **Mockito** to create automated unit tests.
- Validation:
  - Comparison of the Java test results with the results produced by the original COBOL system.
  - Execution of regression tests to ensure that errors are not introduced into the existing functionality.

- **Test Coverage**:
  - Ensuring a test coverage of at least 80% for the migrated methods.
- Use of tools like **JaCoCo** to measure and improve coverage.

5.3. Integration Points

5.3.1. Identification of Integration Points with External Systems

Identifying integration points with external systems is critical to ensuring that the new architecture can interact without interruption with the necessary systems. The main areas of integration include:

- **Authentication and Authorization**:
  - Integration with **AWS Cognito** for user management, Single Sign-On (SSO), and multi-factor authentication (MFA).
- **Messaging Systems**:
  - Asynchronous communication with **Amazon SQS** or **SNS** to handle workflows and events.
- **Storage**:
  - Interaction with **Amazon S3** for file management and storage of unstructured data.
- **Database**:
  - Access to **AWS RDS PostgreSQL** for CRUD operations, set up through **Spring Data JPA**.
- **Monitoring Services**:
  - Integration with **AWS CloudWatch** to log metrics, logs, and system events.

5.3.2. Development and Testing of Integration Logic

The development of integration logic will be based on principles of decoupling and iterative testing to ensure that systems interact effectively:

- **Development**:
  - Use of **Spring Cloud AWS** to simplify connections with AWS services.
- Design of RESTful interfaces for communication with external systems, using **Spring Web**.

- **Integration Testing**:
  - Simulation of external systems using tools like **WireMock** to test real scenarios.
  - Validation of API REST responses and asynchronous events using **Postman** or **Newman**.
- **Test Automation**:
  - Implementation of automated tests with **JUnit 5** and **Testcontainers** to recreate integration environments.

5.4. Testing Approach

5.4.1. Unit Testing Strategy

The unit testing approach will ensure that each component functions correctly in isolation:

- **Test Coverage**:
  - A minimum coverage of 80% of the code is required using tools like **JaCoCo**.
- **Tools**:
  - Use of **JUnit 5** for unit testing.
  - Simulation of external dependencies with **Mockito** to ensure component isolation.
- **Automated Testing**:
  - Automation of unit testing as part of the CI/CD pipeline, using **GitHub Actions** or **Jenkins**.

5.4.2. Integration Testing Strategy

Integration testing will validate the interaction between components and external systems:

- **Approach**:
  - Test API REST with real data using tools like **Postman** or automated test scripts.
- Validate database transactions and message operations in **SQS/SNS**.

- **Simulation of External Systems**:
  - Use of **Testcontainers** to recreate databases and messaging systems.
  - Simulation of external services using **WireMock** or isolated environments.

5.4.3. System Testing Strategy

System testing will verify that all components work together as a cohesive system in a simulated production environment:

- **Full Functionality Testing**:
  - Validate that the application meets the business and functional requirements.
- **Performance Testing**:
  - Use of tools such as **JMeter** to evaluate performance under different loads.
- **Security Testing**:
  - Validate data protection and authentication through **AWS Cognito** and security settings in the APIs.
- **Regression Testing**:
  - Ensure that new implementations do not affect existing functionalities.
- **Testing Environment**:
  - Use of controlled environments in **AWS ECS Fargate** to replicate production conditions.

6. Conclusion

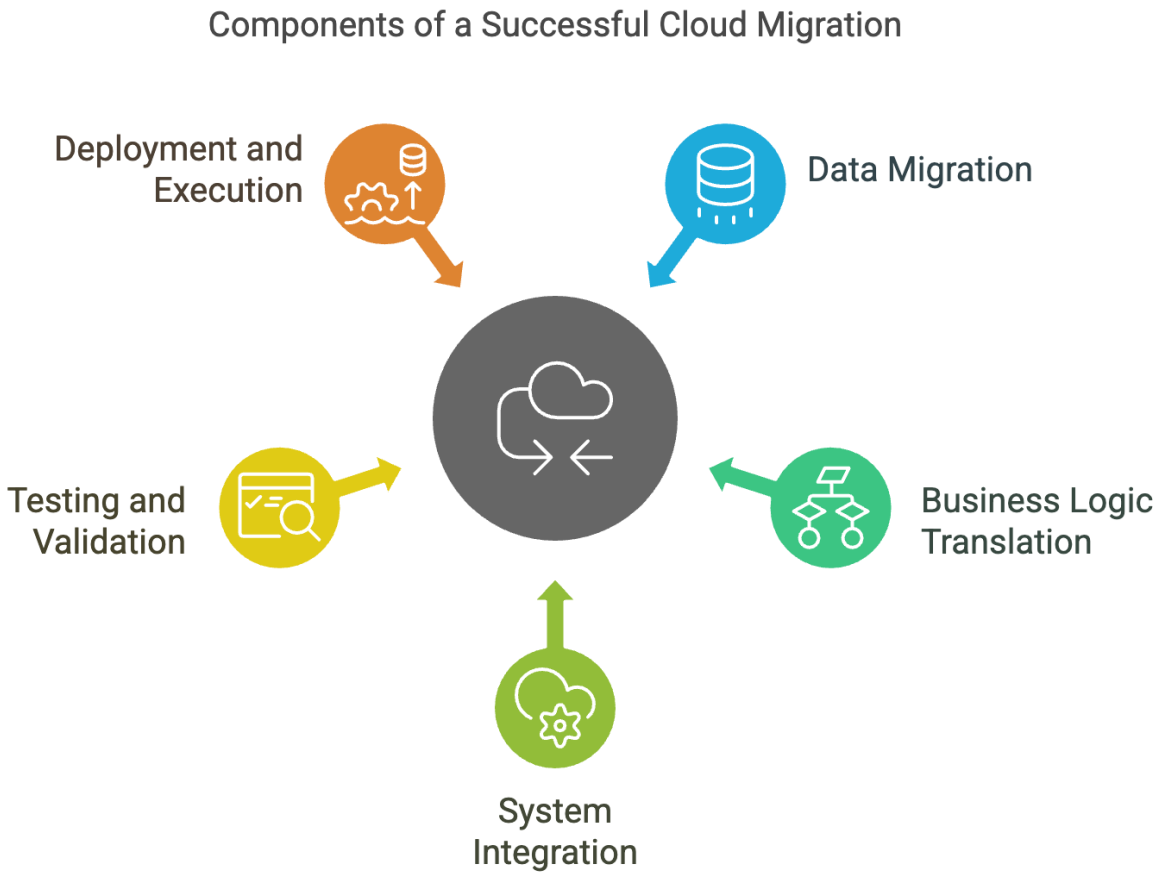
6.1. Summary of Migration Plan

The migration plan presented aims to modernize a legacy system based on technologies such as COBOL, VSAM and CICS, towards a modern architecture based on **Spring Boot** and **AWS services** (Cognito, ECS Fargate, RDS PostgreSQL and ECR). The main steps include:

1. **Data Migration**:
   - Transform the data stored in VSAM into a relational database in **AWS RDS PostgreSQL**, ensuring integrity and consistency through data migration and cleaning tools.

2. Business Logic Translation:
    - Convert the logic written in COBOL to Java through a hybrid approach that combines automated tools and manual refactoring.
    - Modularize the business logic to improve its maintainability and scalability.
3. Integration with External Systems:
    - Identify and develop integration points with critical services such as authentication via AWS Cognito, messaging with SQS/SNS, and storage in S3.
4. Testing and Validation:
    - Implement a comprehensive testing strategy that includes unit, integration, and system tests, ensuring a reliable and uninterrupted deployment.
5. Deployment and Execution:

- Use AWS ECS Fargate to manage serverless containers and AWS ECR to store Docker images, simplifying infrastructure maintenance and allowing efficient scalability.



6.2. Benefits of Modernization

Modernizing the system to a **Java**-based architecture and **AWS** cloud services provides numerous strategic, operational, and economic benefits:

- **Scalability and Performance**:
  - The container and microservices-based architecture allows for horizontal scaling as needed, optimizing resources and costs.
- **Reduction of Operational Costs**:
  - By eliminating reliance on mainframes and adopting cloud services like AWS, the cost of infrastructure and maintenance is significantly reduced.
- **Agility and Development Speed**:
  - Using modern frameworks like Spring Boot, in conjunction with CI/CD tools, accelerates the development and deployment of new features.

- Security Enhancement:
  - With **AWS Cognito** and modern authentication protocols like OAuth2 and JWT, secure and flexible identity management is guaranteed.
- Ease of Maintenance:
  - Migrating the code to Java and modularizing business logic improves readability, reusability, and upgradability.
- Integration with New Technologies:
- The compatibility with AWS services and industry standards allows for easy integration of future technologies and expansion of the system's functionality.
- Regulatory Compliance:
  - The ability to audit and track activities in AWS CloudWatch ensures compliance with regulations such as GDPR, PCI-DSS, and SOX