Microservice Architecture Document

Banking System Deployment Approaches

Document Version: 1.0

Date: April 2, 2025

Status: Draft

Table of Contents

Introduction

This document outlines the architectural approaches for deploying a banking system based on microservices. It presents two primary deployment strategies: a standard microservices approach with a distributed database and a cell-based architecture. Each approach is analyzed for its advantages, disadvantages, and suitability for the banking domain.

Executive Summary

The banking system will be implemented using three core microservices to handle different aspects of account management and transactions. Two architectural approaches have been evaluated:

1. **Standard Microservices with Distributed Database:** A traditional microservices approach with a horizontally sharded database, providing centralized data management with scaling capabilities.

2. **Cell-Based Architecture:** An approach where independent, self-contained cells each manage a subset of customer accounts, providing enhanced fault isolation and scalability.

Each approach offers distinct advantages in terms of data consistency, scalability, fault tolerance, and operational complexity. The choice between these approaches should be based on specific business requirements, expected growth patterns, and operational constraints.

Microservice Architecture Overview

Service Decomposition

The banking system has been decomposed into three microservices:

1. **Deposit/Transaction Microservice:** Handles all transaction-related operations
2. **Account Command Microservice:** Manages account lifecycle and configuration
3. **Account Inquiry Microservice:** Provides read-only access to account information

Microservice Responsibilities

1. Deposit/Transaction Microservice

This service handles all financial transactions and is responsible for:

- **Credit Operations:** Processing deposits and incoming transfers
- **Debit Operations:** Processing withdrawals and outgoing payments
- **Fund Transfers:** Managing transfers between accounts
- **Earmarking:** Handling fund reservations and blocks
- **Fee/Interest Calculations:** Computing and applying service fees and interest

2. Account Command Microservice

This service manages the lifecycle and configuration of accounts:

- **Account Opening:** Creating new customer accounts with proper validation
- **Account Closure:** Processing account termination requests
- **Status/Signal Updates:** Managing account statuses (active, dormant, frozen, etc.)
- **Account Updates:** Modifying account parameters and configurations

3. Account Inquiry Microservice

This service provides read-only operations for account information:

- **Account Summary:** Retrieving account balances and basic information
- **Transaction History:** Providing detailed transaction records
- **Earmark Inquiry:** Checking reserved funds and blocks
- **General Inquiries:** Processing all other information requests

Deployment Architecture Options

Standard Microservices with Distributed Database

Architecture Description

This approach follows a traditional microservices pattern where services are deployed independently but share a distributed database system. The database is horizontally sharded based on account numbers to provide scalability.

Technical Components

Component	Implementation Details
Application Layer	Microservices deployed on OpenShift containers
Database Layer	Horizontally sharded database partitioned by account numbers
Transaction Management	Database-level ACID transactions
Scaling Strategy	Vertical scaling of database shards, horizontal scaling of services
Communication	REST/gRPC for synchronous calls, message queues for asynchronous operations

Architectural Diagram

[Placeholder for Standard Microservices Architecture Diagram]

Advantages

- **Data Consistency:** Centralized database makes it easier to maintain data integrity across services
- **Simplified Deployment:** Easier management of a single distributed database compared to multiple isolated databases
- **Efficient Resource Utilization:** Shared database resources reduce infrastructure costs
- **Comprehensive Observability:** Unified monitoring and logging across all services
- **Simplified Transaction Management:** Native database transactions for operations affecting multiple entities

Challenges

- **Potential Single Point of Failure:** Issues with the database can impact all microservices
- **Performance Bottlenecks:** Heavy loads on specific database shards can create system-wide bottlenecks
- **Sharding Complexity:** Managing even data distribution and rebalancing can be technically challenging
- **Limited Account Isolation:** No logical separation between accounts makes blast radius management difficult
- **Scaling Limitations:** Database scaling might become a challenge under high loads

Cell-Based Architecture

Architecture Description

In this approach, the system is divided into multiple independent "cells," each containing its own instances of all microservices and a dedicated database. Each cell manages a subset of customer accounts, providing strong isolation between different customer groups.

Technical Components

Component	Implementation Details
Cell Structure	Self-contained units with dedicated microservices and database
Database Layer	Isolated per-cell databases, each handling a customer subset
Transaction Management	Local ACID transactions within cells, Saga pattern for cross-cell operations
Scaling Strategy	Horizontal cell duplication, adding more cells for more customers
Communication	Direct service calls within cells, API gateway for routing between cells
Failure Domain	Limited to individual cells

Architectural Diagrams

[Placeholder for Cell-Based Architecture Diagram 1]

[Placeholder for Cell-Based Architecture Diagram 2]

Advantages

- **Strong Fault Isolation:** A failure in one cell affects only a limited subset of customers (e.g., with 20 cells, a complete cell failure impacts ≤5% of customers)
- **Horizontal Scalability:** Easy to scale by adding more cells as customer base grows
- **Reduced Blast Radius:** Problems and maintenance activities can be contained to specific cells

- **Performance Predictability:** Dedicated resources per cell provide more consistent performance
- **Simplified Testing and Deployment:** Changes can be rolled out progressively across cells

Challenges

- **Cross-Cell Data Consistency:** Maintaining data consistency across cells requires complex synchronization mechanisms
- **Operational Complexity:** Managing multiple instances of services and databases increases operational overhead
- **Routing Mechanism:** Implementing and maintaining the customer-to-cell routing system adds complexity
- **Higher Infrastructure Costs:** Resource duplication across cells increases overall resource requirements
- **Cross-Cell Transactions:** Operations spanning multiple cells require additional coordination mechanisms

Architecture Comparison

Architectural Aspect	Standard Microservices + Distributed DB	Cell-Based Architecture
System Organization	Single logical system with distributed components	Multiple independent cells, each a complete system
Scalability Model	Horizontal scaling by adding more database shards	Horizontal scaling by adding more complete cells
Data Organization	Sharded database based on account numbering	Each cell contains a complete subset of accounts
Fault Isolation	Failures may impact multiple services due to shared resources	Strong fault isolation—failures contained within cells
Disaster Recovery	Requires comprehensive DR for the entire system	Can implement cell-by-cell recovery strategies
Operational Complexity	Requires robust distributed transaction management	Requires careful cell orchestration and routing
Deployment Flexibility	Centralized database with scaling strategies	Multiple instances of services and databases
Consistency Implementation	Distributed transactions (e.g., two-phase commit)	Local transactions within cells, Saga pattern across cells
Maintenance Operations	System-wide maintenance impacts all customers	Maintenance can be performed cell by cell
Resource Efficiency	More efficient resource utilization	Some resource duplication across cells
Implementation Complexity	Lower initial implementation complexity	Higher initial implementation complexity
Long-term Maintenance	Potentially challenging as system grows	More manageable with clear cell boundaries

Considerations and Recommendations

When selecting between these architectural approaches, consider the following factors:

Business Considerations

- **Customer Base Size:** For larger customer bases, the cell-based approach offers better scalability
- **Regulatory Requirements:** Data isolation requirements may favor the cell-based approach
- **Service Level Agreements:** High availability needs may favor cell-based architecture's fault isolation
- **Growth Projections:** Expected rapid growth favors the more horizontally scalable cell-based approach

Technical Considerations

- **Development Maturity:** Less experienced teams may prefer starting with the standard approach
- **Operational Capabilities:** Cell-based architecture requires more sophisticated operational practices
- **Performance Requirements:** Analyze transaction patterns to determine the better-performing architecture
- **Migration Strategy:** Consider how an existing system would migrate to either architecture

Phased Approach Option

A hybrid approach could begin with the standard microservices architecture and evolve toward a cell-based model as the system matures and customer base grows.

Appendix: Glossary of Terms

- **ACID Transactions:** Atomic, Consistent, Isolated, Durable database transactions
- **Blast Radius:** The scope of impact when a failure occurs in the system
- **Cell:** A self-contained unit containing all microservices and a dedicated database
- **Earmark:** A reservation or block on funds within an account
- **Horizontal Scaling:** Adding more instances of a component (scaling out)
- **Microservice:** A small, independent service focused on a specific business capability
- **Saga Pattern:** A sequence of local transactions coordinated to maintain consistency
- **Sharding:** Partitioning a database across multiple instances based on a key
- **Vertical Scaling:** Adding more resources to a single component (scaling up)