Microservice deployment approach

High level plan is to have three microservices

1. Deposit/Transaction MS
2. Account Command MS
3. Account Inquiry MS
4. Depsoit microservice will have
   1. Credit
   2. Debit
   3. Fund Transfer
   4. Earmark
   5. Fee/ Interest
5. Account Command MS will have
   1. Account open
   2. Account close
   3. Account Status/Signal update
   4. Account update
6. Account Inquiry MS will have
   1. Account Summary
   2. Transaction History Inquiry
   3. Earmark Inquiry
   4. All other Inquiry

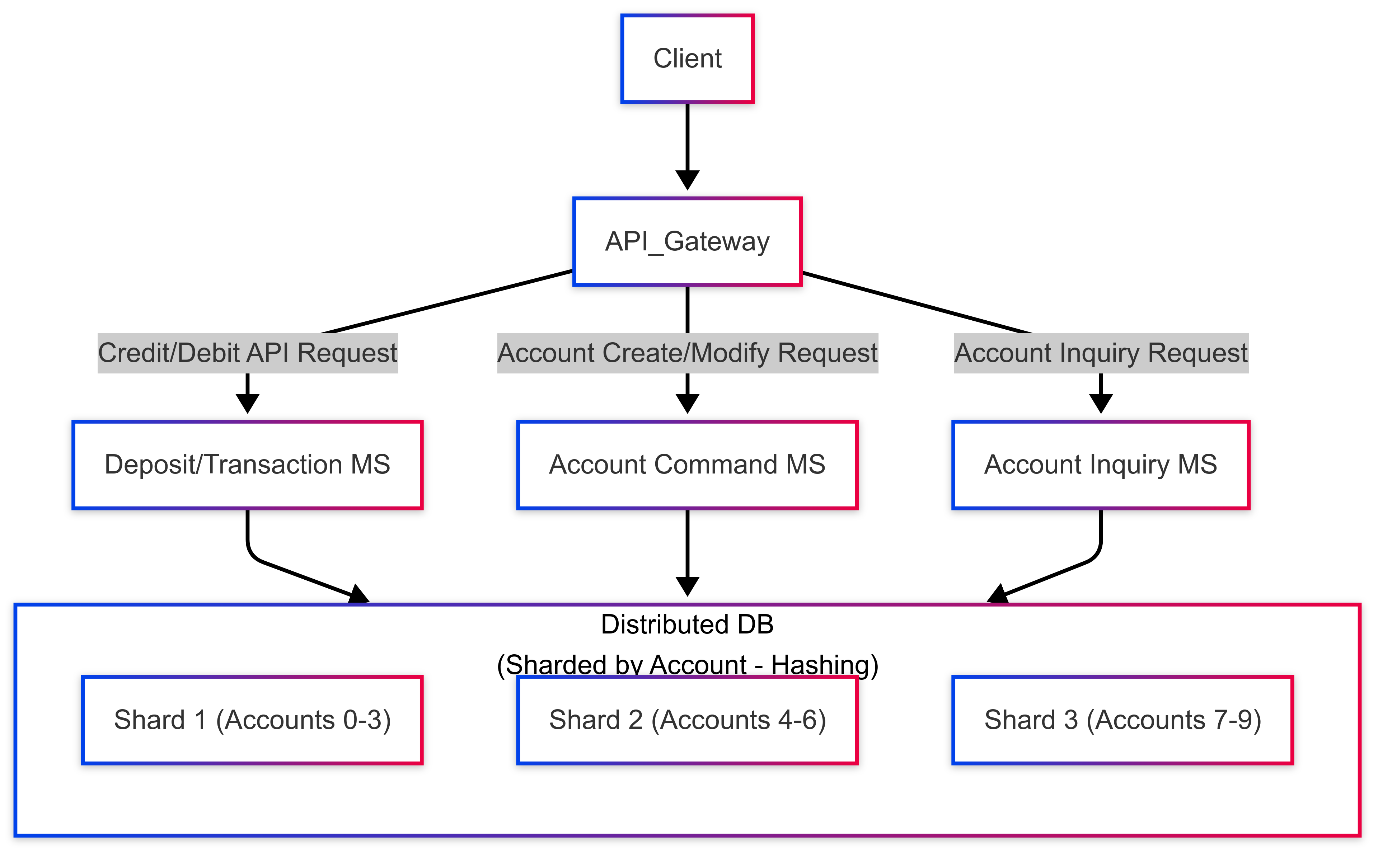
There are two architecture approach primarily discussed for deployment of app and for these microservices.

1. **Standard Microservices + Distributed DB**
2. Cell-Based Deployment Approach

Approach 1: **Standard Microservices + Distributed DB**

**Architecture:**

| **Aspect** | **Implementation** |
| --- | --- |
| Application | Microservices deployed on OpenShift |
| Database | Horizontally sharded DB by account |
| Transaction Handling | Database-level ACID transactions |
| Scalability | Vertical scaling of DB shards |

****

**Pros:**

* **Centralized Data:** Data consistency is easier to manage as all microservices share a single database.
* **Simplified Deployment:** Easier to deploy and manage a single, distributed database compared to multiple cell-based databases.
* **Scalability:** DB sharding allows for horizontal scaling.
* Unified observability across services
* Lower initial infrastructure costs

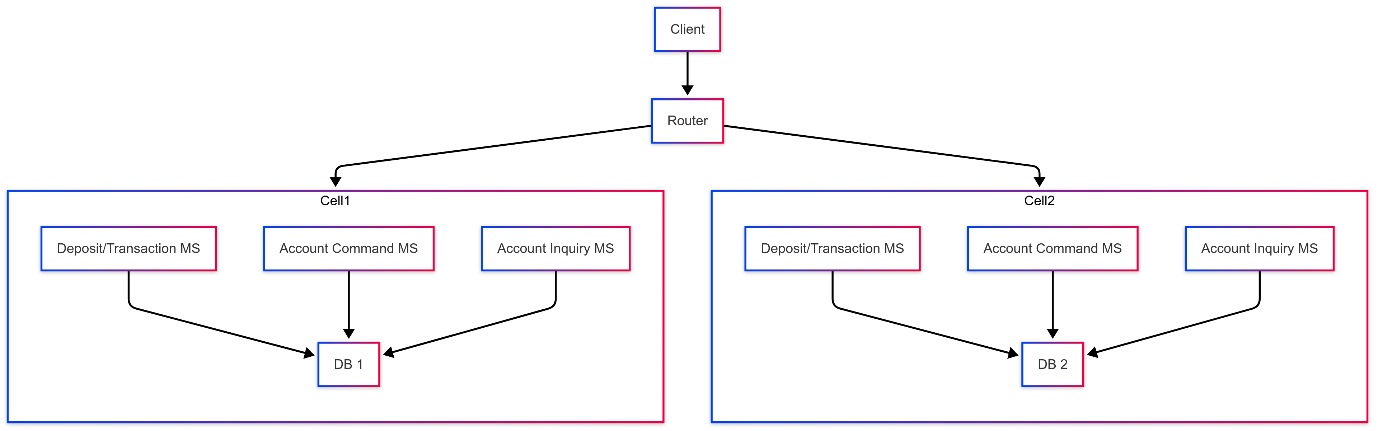
**Cons:**

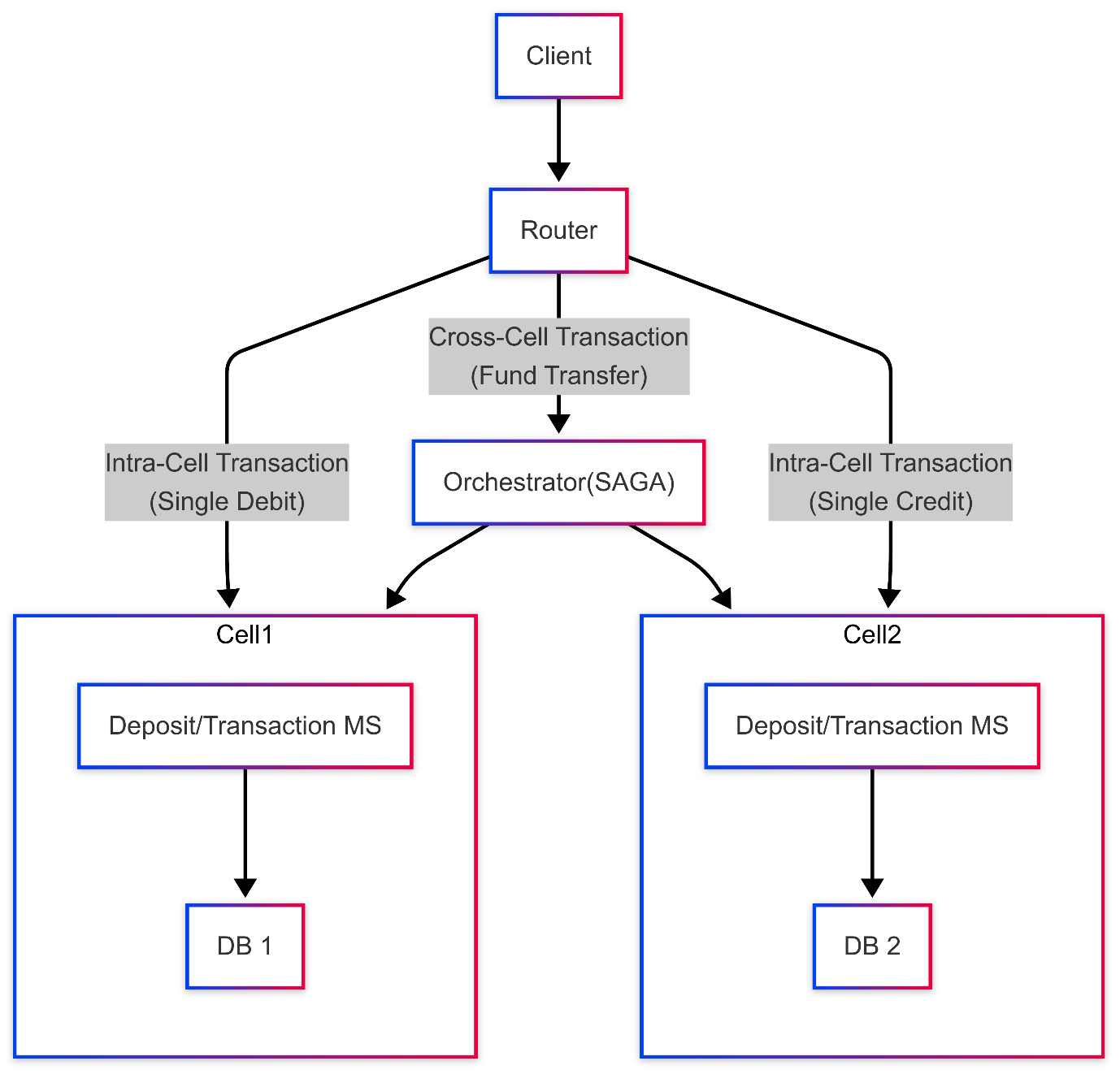
* **Single Point of Failure:** While sharded, issues with the central database can impact all microservices.
* **Potential Bottlenecks:** Heavy load on specific shards or the database connection pool can create bottlenecks.
* **Complexity of Sharding:** Managing sharding and ensuring even data distribution can be complex.
* **No logical separation of account is possible**

Approach 2: Cell-Based Architecture

**Architecture:**

| **Aspect** | **Implementation** |
| --- | --- |
| Database | Isolated per-cell databases |
| Transaction Handling | Saga pattern for cross-cell transactions |
| Scalability | Horizontal cell duplication |
| Failure Domain | Limited to individual cell |





**Pros:**

* **Isolation:** Cells are isolated, so a failure in one cell does not affect others, improving fault tolerance. Like for example, if there are 20 cells then even 100% failure in one cell affects ≤5% customers.
* **Scalability and Resilience:** Easier to scale by adding more cells.
* **Reduced Blast Radius:** Problems within one cell are contained.

Cons:

* **Data Consistency:** Maintaining data consistency across cells can be complex, requiring robust synchronization mechanisms.
* **Increased Complexity:** Managing multiple cells and databases adds operational complexity.
* **Routing Complexity:** Implementing and maintaining the routing mechanism is crucial.
* **Resource Duplication:** Each cell duplicates resources, leading to higher infrastructure costs.

**Comparison of the Two Approaches**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Standard Microservices + Distributed DB (Sharded) Approach** | **Cell-Based Approach** |
| **Architecture** | Single logical system with a distributed, sharded database | Independent, self-contained cells, each with its own microservices and database |
| **Scalability** | Horizontal scaling by adding more shards | Horizontal scaling by adding more cells |
| **Data Partitioning** | Sharded database based on account number | Each cell contains a subset of accounts |
| **Fault Isolation** | Failures may impact multiple services due to shared database | Strong fault isolation—failure in one cell does not affect others |
| **Operational Complexity** | Requires strong distributed transaction management | Requires careful cell orchestration, data partitioning, and routing |
| **Deployment Flexibility** | Easier to manage a centralized DB with scaling strategies | More effort required for maintaining multiple instances of microservices and DBs |
| **Consistency Model** | Distributed transactions need to be handled (e.g., two-phase commit, eventual consistency) | Each cell is self-contained, reducing consistency concerns but requiring cross-cell synchronization (e.g., SAGA) |
| **Maintenance Overhead** | Requires distributed DB maintenance (backups, schema updates, etc.) | Requires maintaining multiple DBs, leading to higher ops effort |