Technical Architecture and Solution for a Scalable Real-Time Transaction System

1. Objective

Objective

Develop a real-time balance calculation system in Java, deploy it on a Kubernetes (K8s) cluster on a cloud platform (AWS/GCP/Ali) and ensure it meets high availability and resilience requirements.

Requirements

1. Core Functionality:

Implement a service that can process financial transactions and update account balances in real-time.

Each transaction should include a unique transaction ID, source account number, destination account number, amount, and timestamp.

`The service should handle concurrent transactions and update balances accordingly.`

2. High Availability and Resilience:

Deploy the service on a Kubernetes cluster (AWS EKS, GCP GKE, Alibaba ACK). Use Kubernetes features like Deployment, Service, and Horizontal Pod Autoscaler (HPA) to ensure high availability and scalability.

Implement a retry mechanism for failed transactions.

Use a managed database service (e.g., AWS RDS, GCP Cloud SQL, Alibaba RDS) to store account and transaction data.

`Ensure data consistency and integrity using database transactions and locks where necessary.`

3. Performance:

Optimize the service to handle high-frequency transactions.

`Implement caching using a distributed caching service (e.g., AWS ElastiCache, GCP Memorystore, Alibaba Cloud ApsaraDB for Redis).`

4. Testing:

Write unit tests using JUnit.

Write integration tests to ensure the service works correctly with the database and cache.

Perform resilience testing to ensure the service can recover from failures (e.g., pod restarts, node failures).

`Measure performance using a load-testing tool (e.g., Apache JMeter).`

5. Mocking Data:

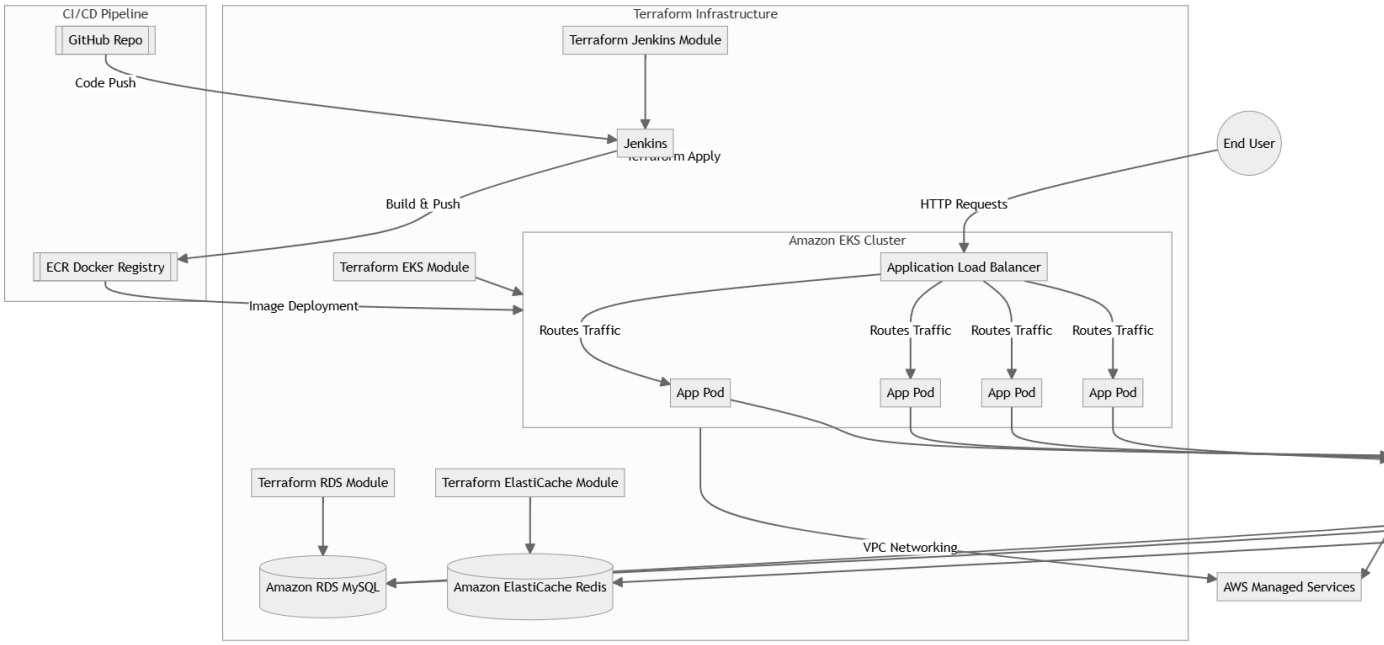
Use a mock data generator to simulate a large number of transactions and account balances for testing purposes.

2. Design of technical architecture

2.1 Summary

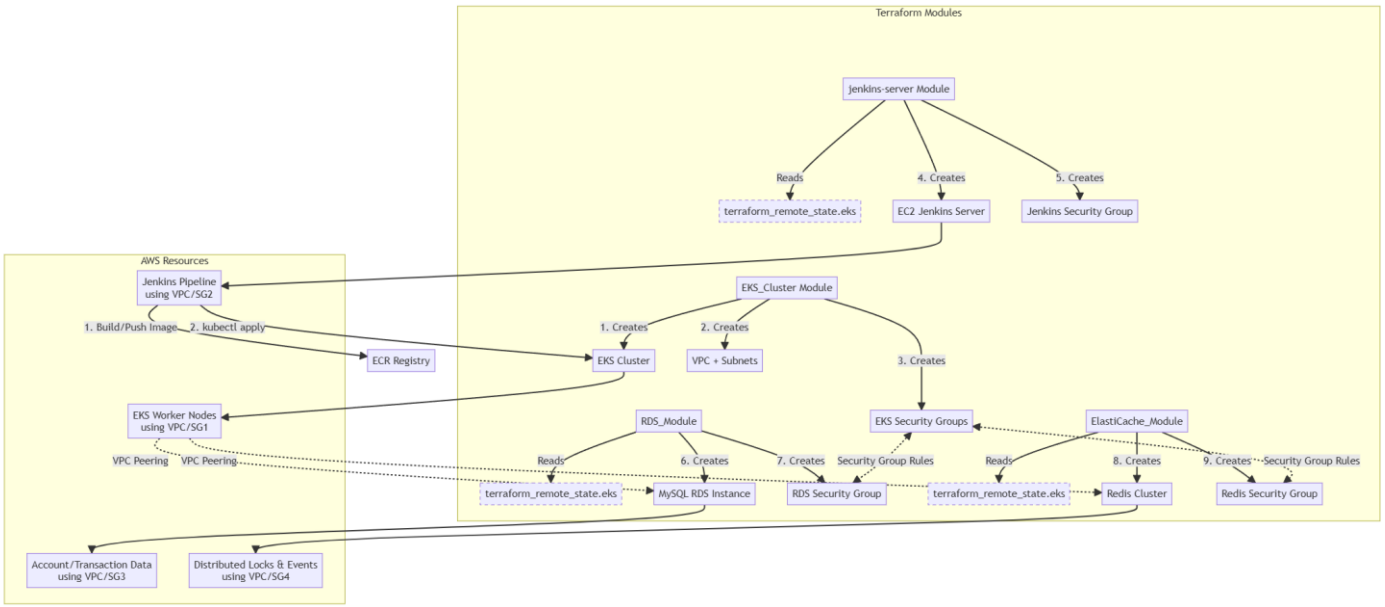
The combination of **Amazon EKS**, **Elastic Load Balancer**, **Amazon RDS** with Multi-AZ, **ElastiCache**, and **Terraform** provides a highly available and scalable architecture for the project. The use of managed services simplifies infrastructure management and guarantees that high availability is maintained without complex configurations.

2.2 Overall architecture diagram

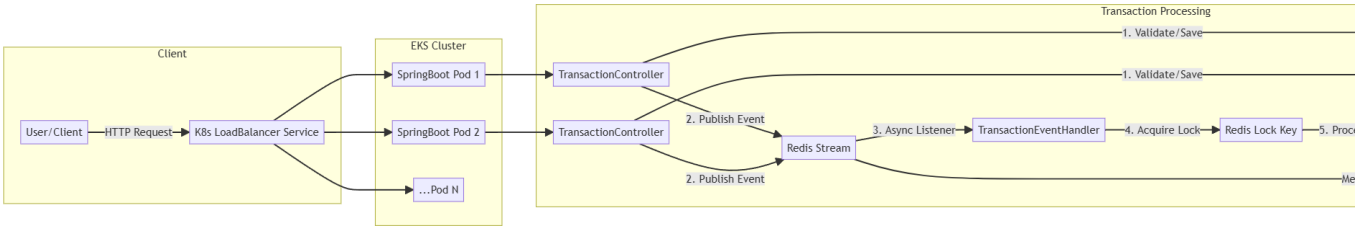


2.3 Flow charts

2.3.1 AWS Cluster Management Flowchart



2.3.2 Business Request Flow Chart



3. Functions

1. Account Management

- **Create Account**: Provides an endpoint to create a new account by accepting a Account object, performing validation, and saving the account data in the database.
- **Delete Account**: Provides an endpoint to delete an account by its ID.

2. Transaction Management

- **Process Transaction**: Provides an endpoint to accept and enqueue a transaction. The transaction is saved in the database immediately, and then processed asynchronously. If any errors occur, appropriate error messages are returned to the user.
- **Get Transaction Status**: Provides an endpoint to get the status of a transaction by its ID. The status is fetched from the database.

3. Transaction Processing

- **Asynchronous Processing**: Transactions are processed asynchronously. The processTransactionAsync() method handles the business logic after the transaction is saved, including acquiring a distributed lock (using Redis) to ensure only one instance processes the transaction at a time.
- **Funds Transfer**: The core logic of processing the transaction involves transferring funds between source and destination accounts. The balances of both accounts are updated, and the transaction status is changed accordingly.
- **Scheduled Task for Pending Transactions**: A scheduled task is set up to periodically process pending transactions that are either in "PENDING" or "IN_PROGRESS" status. Redis locks are used to ensure that only one instance processes transactions at a time.

4. Infrastructure Setup using Terraform

- **AWS EC2 Instance for Jenkins**: Terraform is used to provision an EC2 instance in AWS, where Jenkins is installed and used to automate the CI/CD pipeline.
- **AWS EKS Cluster**: Terraform is used to provision an AWS EKS (Elastic Kubernetes Service) cluster for running Kubernetes workloads.
- **AWS RDS Instance**: Terraform provisions an AWS RDS instance for database services.
- **AWS ElastiCache for Redis**: Terraform provisions an AWS ElastiCache Redis cluster for distributed locking.
- **AWS ECR (Elastic Container Registry)**: Terraform provisions an AWS ECR repository to store Docker images used in the application.

5. CI/CD Pipeline with Jenkins

- **Build and Test**: The Jenkins pipeline builds the project using Maven and runs tests.
- **Docker Image Build**: The Jenkins pipeline builds Docker images of the application and pushes them to the AWS ECR repository.
- **Kubernetes Deployment**: The pipeline deploys the application to AWS EKS using Kubernetes manifests. It updates the image with the latest commit hash as a tag.

6. Kubernetes Deployment Configuration

- **Spring Boot Application Deployment**: The application is packaged as a Docker container and deployed to AWS EKS with multiple replicas (4 in this case). The deployment is configured to expose port 8080, and a service of type LoadBalancer is created to expose the application externally on port 80.
- **Automatic Image Update**: When a new Docker image is pushed to ECR, the pipeline updates the running deployment in Kubernetes with the new image, ensuring that the application is always running the latest version.

4. Database database table design

Accounts:

```
CREATE TABLE `accounts` (
  `account_id` varchar(50) NOT NULL COMMENT 'Unique identifier
for the account',
  `balance` decimal(19,4) NOT NULL COMMENT 'Current balance of
the account',
  `account_type` varchar(20) NOT NULL COMMENT 'Type of account
(e.g., savings, checking)',
  `currency` varchar(10) NOT NULL COMMENT 'Currency code (e.g.,
USD, EUR)',
  `created_at` timestamp NULL DEFAULT CURRENT_TIMESTAMP COMMENT
'Account creation timestamp',
  `updated_at` timestamp NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP COMMENT 'Last update timestamp',
  PRIMARY KEY (`account_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 ;
```

Transactions:

```
CREATE TABLE `transactions` (
  `transaction_id` varchar(50) NOT NULL COMMENT 'Unique
identifier for the transaction',
  `source_account_id` varchar(50) NOT NULL COMMENT 'Source
account ID',
  `destination_account_id` varchar(50) NOT NULL COMMENT
'Destination account ID',
  `amount` decimal(19,4) NOT NULL COMMENT 'Transaction amount',
  `timestamp` datetime NOT NULL COMMENT 'Timestamp of the
transaction',
  `created_at` timestamp NULL DEFAULT CURRENT_TIMESTAMP COMMENT
'Transaction record creation timestamp',
  `updated_at` timestamp NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP COMMENT 'Last update timestamp',
  `status` varchar(255) NOT NULL,
  PRIMARY KEY (`transaction_id`),
  KEY `source_account_id` (`source_account_id`),
  KEY `destination_account_id` (`destination_account_id`))
ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 ;
```

5. Cache design

Serial number	Cache Key	Cache Value	Describe
1	TRANSACTION_LOCK_	/	redis lock for transaction
2	PENDING_TRANSACTION_TASK_LOCK	/	redis lock for transaction retry schedule

6. Configurations

below configurations is is application.properties

```
# Core pool size (initial number of threads in the pool)
transaction.threadPool.corePoolSize=10

# Max pool size (maximum number of threads in the pool)
transaction.threadPool.maxPoolSize=100

# Queue capacity (maximum number of tasks in the queue before
blocking or rejecting)
transaction.threadPool.queueCapacity=500

# Keep-alive seconds (time to wait before closing idle threads)
transaction.threadPool.keepAliveSeconds=60

# transaction retry job time
transaction.scheduled.fixedRate=60000
```

Simple TPS calculation:
Before the CPU scheduling of the system reaches a bottleneck, the performance allowed by the system is almost related to the size of the thread pool, so the expected TPS that can be achieved by adjusting the thread pool size can be estimated by testing the TPS when a small number of threads are used

7. Related services

Related services and systems that need to be relied upon in the technical solution:

Serial number	Service Name	Describe	Remark

5	**AWS ECR (Elastic Container Registry)**	Container image repository for storing and managing Docker images.	Secure storage and seamless integration with EKS for deployment.
6	**S3 (Simple Storage Service)**	Object storage service used to store Terraform remote configuration files.	Ensures version-controlled state management for Terraform deployments.
7	**Terraform**	Infrastructure as Code (IaC) tool for provisioning and managing AWS resources.	Enables automated and version-controlled infrastructure management.
8	**Jenkins**	CI/CD automation tool for building, testing, and deploying applications.	Automates the entire process of building Docker images, pushing them to ECR, and deploying to Kubernetes (EKS).

8. Interface design

1. Create Account API

Request Path
POST /api/v1/accounts
Request Parameters
The request body uses the AccountRequest DTO, containing the following fields:

Parameter	Type	Required	Description
accountId	String	Yes	The account ID
balance	BigDecimal	Yes	The account balance (must be positive)

Example Request

```
{
  "accountId": "12345",
  "balance": 1000.50
}
```

Response Parameters

The response body returns the created account object:

Parameter	Type	Description
accountId	String	The account ID
balance	BigDecimal	The account balance
accountType	String	The type of the account
currency	String	The currency of the account

Example Response

```
{"message": "Account deleted successfully."}
```

---

3. Get Account API

Request Path

GET /api/v1/accounts/{accountId}

Path Parameter

Parameter	Type	Required	Description
accountId	String	Yes	The ID of the account to retrieve

Example Request

GET /api/v1/accounts/12345

Response Parameters

The response body returns the account details:

Parameter	Type	Description
accountId	String	The account ID
balance	BigDecimal	The account balance
accountType	String	The type of the account
currency	String	The currency of the account

Example Response

```
{"accountId": "12345","balance": 1000.50,"accountType":
"SAVINGS","currency": "USD"}
```

---

4. Process Transaction API

Request Path
POST /api/v1/transactions/process
Request Parameters
The request body uses the TransactionRequest DTO, containing the following fields:

Parameter	Type	Required	Description
transactionId	String	Yes	The transaction ID
sourceAccountId	String	Yes	The ID of the source account
destinationAccountId	String	Yes	The ID of the destination account
amount	BigDecimal	Yes	The transaction amount (must be greater than 0)
timestamp	LocalDateTime	No	The timestamp of the transaction

Example Request

```
{"message": "Transaction accepted"}
```

Example Response (Error)

```
{"message": "Error processing transaction: Insufficient funds"}
```

---

5. Get Transaction Status API

Request Path

GET /api/v1/transactions/{transactionId}/status

Path Parameter

Parameter	Type	Required	Description
transactionId	String	Yes	The ID of the transaction

Example Request

GET /api/v1/transactions/txn12345/status

Response Parameters

The response body returns the current status of the transaction:

Parameter	Type	Description
status	String	The status of the transaction

Example Response

10. Compatibility

Because the system is only the first version, there is currently no consideration for compatibility with the old version system. At the same time, this is a pure backend system, so there is no need to consider compatibility with the frontend (I have integrated a simple page for testing, but this is not a system function).

11. Monitoring Alarm and Report

1. AWS CloudWatch Monitoring for EKS

- **Container Insights**:Container Insights in AWS CloudWatch can automatically collect metrics and logs from EKS clusters, such as CPU and memory usage, disk I/O, network traffic, and Kubernetes-level metrics (pods, nodes, etc.).

2. CloudWatch Alarms

System set up CloudWatch Alarms for metrics that exceed defined thresholds. this is my set value:

- **High CPU usage :** > **80%** for **5 minutes**
- **High memory usage :** > **60%** for **5 minutes**
- **Pod crashes or restarts :** More than 1 **pod restarts** in **10 minutes**
- **Node unavailability :** If any node is **NotReady** for **more than 5 minutes**
- **API request latency :** API server request duration > **2 seconds** for **more than 1%** of requests over **5 minutes**

3. CloudWatch Logs

- **EKS Logs Integration**: AWS EKS automatically integrates with CloudWatch Logs to collect logs from Kubernetes clusters

12. Waiting confirmation or improvement items

12.1 AWS SQS instead of Spring Event (hi -recommend)（not done for billing cost）:

- **Use Amazon SQS (Fully managed message queuing service) to decouple transaction processing and improve reliability.**
- **Replace the current ApplicationEventPublisher with SQS for asynchronous task delivery.**
- **Reduced CPU/Memory Usage in system worknode on EKS**

12.2 WAF (Web Application Firewall,not done for billing cost):

- **AWS WAF:** Use AWS WAF to protect application from common web exploits such as SQL

injection, cross-site scripting, etc. It allows project to set rules and filters for malicious traffic.

- **Custom Rules:** Create custom rules to block malicious IP addresses, or patterns in requests indicative of attacks.

12.3 API Rate Limiting & Throttling（not done for billing cost）：

- **Rate Limiting:** Protect API endpoints from brute force attacks by limiting the number of requests a user can make per minute/hour. Use tools AWS API Gateway, or implement rate-limiting directly in service.

12.4 HTTPS Security（not done for billing cost）：

- **Set up a custom domain in Route 53**
- **Request an SSL certificate for domain in ACM and update to ELB**
- **HTTP Strict Transport Security (HSTS):** Force browsers to connect over HTTPS only, preventing man-in-the-middle attacks.

13. Reference documents

1. https://registry.terraform.io/modules/terraform-aws-modules/ i always use it to find Terraform module example

14. Work schedule

/

15. Q&A：

1. How the system ensure high availability and scalability

project leverages AWS's managed services and cloud-native features to ensure both high availability and scalability.

- **Amazon Elastic Kubernetes Service (EKS)**:
  - The project is deployed on **Amazon EKS**, which automatically scales the number of pods based on traffic load. With Kubernetes, the deployment is resilient to node failures, as EKS can automatically replace any failed pods. By setting the number of replicas to 4 in the Kubernetes deployment (replicas: 4), the project ensures that it can handle increased traffic by distributing the load across multiple pods.
  - **Horizontal Pod Autoscaling (HPA)** is configured to scale the number of pods in response to traffic fluctuations. This enables the system to scale up during high demand and scale down when traffic is low, optimizing resource usage and ensuring cost-efficiency.

- **Amazon Elastic Load Balancer (ELB)**:
  - The project uses the **LoadBalancer** service type in Kubernetes to expose the application. This setup ensures that incoming requests are evenly distributed across the available pods. The AWS **Elastic Load Balancer (ELB)** automatically detects healthy pods and routes traffic to them, ensuring high availability even if one or more pods become unhealthy.
  - The LoadBalancer ensures that if a pod or container fails, traffic will be automatically redirected to other healthy instances, minimizing downtime and guaranteeing continuous service.
- **Amazon ElastiCache (Redis)**:
  - For distributed locking and caching, **Amazon ElastiCache** (Redis) is employed. Redis is a highly available, managed in-memory data store that supports automatic failover and replication. This guarantees that the project can continue processing even if one Redis node fails.
  - Redis's elasticity and high availability play a critical role in the transaction lock mechanism, ensuring that only one instance processes a transaction at any given time.
- **Amazon RDS (Relational Database Service)**:
  - The project relies on **Amazon RDS** for transaction data storage. RDS provides high availability through **Multi-AZ deployments**, which provisions a synchronous standby replica in a different Availability Zone (AZ). This ensures automatic failover in case of database instance failure.
  - The RDS service guarantees reliable database connections with automatic failover, ensuring that transactions are not lost and database queries are routed to the available instance.
- **Terraform for Infrastructure Management**:
  - **Terraform** is used to manage and provision AWS resources, such as EKS clusters, RDS instances, and ElastiCache. Through Terraform, the infrastructure can be dynamically scaled based on demand, with scaling policies and availability zones configured to support high availability.
  - By using **Infrastructure as Code (IaC)**, the project can easily manage, version, and automate changes to the infrastructure across multiple environments, improving the overall maintainability and consistency of the deployment.

2. What's the retry mechanism for failed transactions

- **Distributed Lock Release**: If a transaction fails, the system ensures that the lock is always released using releaseLock(). This allows other instances to retry the transaction if the first attempt fails.
- **Transaction Status Handling**: If a transaction fails, it is marked with a FAILED status in the database. Then the scheduled task processPendingTransactions() is designed to pick up transactions that are still marked as PENDING or IN_PROGRESS, and retry processing them.
- **Graceful Failure Handling**: In the processTransactionAsync() method, if an error occurs (e.g., insufficient funds), the system logs the failure, ensures the lock is released, and marks the transaction as FAILED. This enables external systems or monitoring tools to retry failed transactions if necessary.

3. How ensure data consistency

- **Transactional Integrity**: The use of @Transactional annotations ensures that database changes for transactions are handled atomically. This ensures that a transaction is either completely processed or not processed at all (in case of failures).
- **Locking Mechanism**: Redis locks ensure that only one instance processes a particular transaction at a time. This prevents race conditions where multiple instances might attempt to modify the same

data concurrently.

- **Immediate Database Persistence**: When a transaction is enqueued, it is immediately saved in the database to ensure that transaction data is persisted before any asynchronous processing begins. This guarantees that transaction data is not lost even in the event of a failure.
- **Select for update use**: The **@Lock(LockModeType.PESSIMISTIC_WRITE)** annotation enforces database-level row locking to guarantee atomic updates to account balances.
- **Event-Driven Architecture**: The use of **ApplicationEventPublisher** and **@TransactionalEventListener** ensures that after a transaction is saved in the database, the transaction processing happens asynchronously, but the transaction's data integrity remains intact through event-based communication.

16. Afterall

While this system provides basic account management and transaction functionalities, the core focus has always been to ensure seamless Kubernetes (K8S) cluster management, security group creation, and CI/CD pipelines. The design emphasizes system robustness, transactional reliability, and high availability, aligning with the objectives of delivering a stable, scalable infrastructure rather than building an overly complex account and transaction management system.

In this project, my primary goal was to ensure that the underlying architecture could support continuous growth and adaptability, with the transaction system serving as a crucial but foundational component. The functionalities related to accounts and transactions are minimal, designed to fulfill basic operational needs while allowing the infrastructure, system security, and automation pipelines to take the forefront. By focusing on Kubernetes, Terraform, and AWS services, I’ve prioritized the system's resilience, scalability, and fault tolerance to deliver a strong, future-proof base for ongoing enhancements.