Virtual Card Financial System Architecture

Comprehensive Technical Documentation

Architecture Design Tool: FigJam (Figma)

Table of Contents

Executive Summary

This document provides a comprehensive technical specification for a mobile-based virtual card financial system designed to enable users to create, manage, and use virtual payment cards for secure digital transactions. The system is built on a **microservices architecture** utilizing independent, loosely coupled modules that communicate through **REST APIs**, ensuring flexibility, scalability, and maintainability.

The architecture prioritizes security, performance, and user experience through strategic technology choices including Node.js with Express.js for backend services, PostgreSQL for data persistence, Redis for high-performance caching, and React Native or Flutter for the mobile client application. This document details how each component operates, communicates with other modules, and processes user requests in a secure and efficient manner.

System Overview

Purpose and Scope

The Virtual Card Financial System is a comprehensive mobile application platform that enables users to request, manage, and utilize virtual payment cards for online and offline transactions. The system handles the complete lifecycle of virtual card management, from user authentication and Know Your Customer (KYC) verification to payment processing and transaction monitoring.

Core Functionality

The system delivers the following primary capabilities:

User Authentication and Authorization - The system provides secure user login mechanisms with JWT-based token authentication, session management through Redis caching, and multi-factor authentication support for sensitive operations. Users authenticate through the mobile application, which communicates with the authentication module via the API Gateway.

User Profile Management - The User Management module maintains comprehensive user account information, including personal details, account status, verification results, and permission settings. This module serves as the central repository for user data accessed by other modules during various operations.

Virtual Card Provisioning - Through integration with third-party card provider APIs, the system enables users to request and receive virtual payment cards. The Card Provision module validates user eligibility through KYC status before provisioning cards and securely stores card details in the database.

Know Your Customer (KYC) Verification - The KYC module implements regulatory compliance requirements by validating user identity through third-party KYC verification providers. This module ensures that only verified users can access card provisioning and payment processing features.

Fraud Detection - The Fraud Detection module analyzes transaction patterns in real-time to identify and prevent suspicious activities. Using machine learning algorithms and velocity checks, the module can block or flag transactions that exhibit fraudulent characteristics.

Payment Processing - The Payment Processing module handles transaction execution through third-party payment gateways. It orchestrates the complete

payment flow, including card validation, fraud checking, gateway communication, and transaction recording.

Real-Time Notifications - The Notification module delivers push notifications to users through Firebase Cloud Messaging (FCM), alerting them of transaction status, fraud alerts, and account activities. Notifications are triggered by events from Payment Processing and Fraud Detection modules.

High-Performance Caching - Redis caching layer stores session tokens, frequently accessed user data, and card information, reducing database load by 60-80% and ensuring response times under 200 milliseconds for 95th percentile latency.

Use Case: Mobile Application

The primary use case for this system is a **mobile application** available on iOS and Android platforms. Users interact with the system exclusively through this mobile client, which provides a user-friendly interface for:

- **User Registration and Authentication** - Creating accounts and logging in securely
- **KYC Verification** - Submitting identity documents and personal information for verification
- **Card Management** - Requesting virtual cards, viewing card details.
- **Payment Execution** - Using virtual cards to make payements
- **Real-Time Notifications** - Receiving instant alerts about transactions and account events

The mobile application communicates exclusively with the backend system through the API Gateway using HTTPS, ensuring all data transmission is encrypted and secure.

Architecture Diagram

Visual Representation

The system architecture has been designed using **FigJam on Figma**, a collaborative design tool that enables visualization of complex system interactions and data flows. The architecture diagram illustrates the complete request-response flow from the mobile client through various system modules, external integrations, and data persistence layers.

Architecture Diagram:

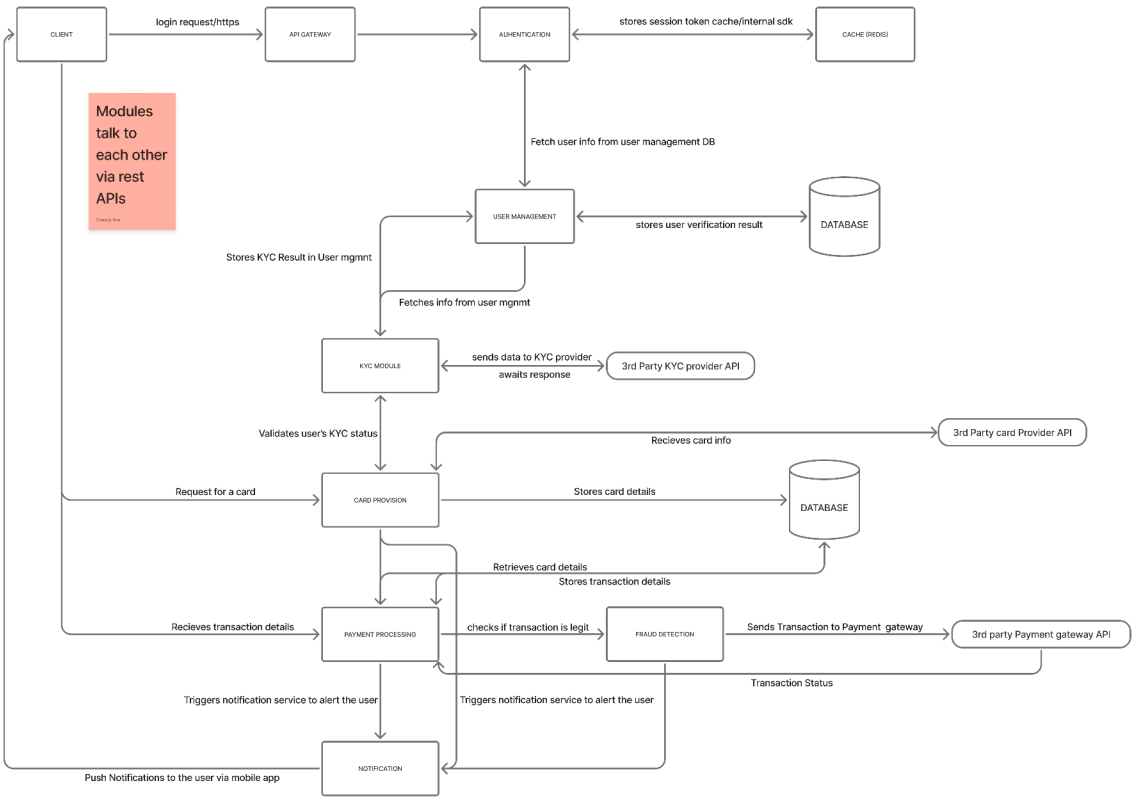

Diagram Components

The architecture diagram depicts the following key elements:

Client Layer - The mobile application serves as the user interface and primary entry point for all user interactions.

API Gateway - Acts as the single entry point for all client requests, handling routing, validation, rate limiting, and API versioning.

Core Modules - Eight independent microservices that handle specific business functions and communicate via REST APIs.

Data Layer - PostgreSQL database serving as the persistent data store for all system information.

Caching Layer - Redis in-memory data store providing high-speed access to frequently used data and session tokens.

External Integrations - Third-party APIs for KYC verification, card provisioning, and payment processing.

Interactive Diagram

The complete interactive version of this architecture diagram is available on Figma using FigJam. This version allows stakeholders to explore the diagram in detail, understand component relationships, and collaborate on architectural discussions.

Figma Link:
https://www.figma.com/board/KGbFoVuhbpP2nzOj2q5E2g/Untitled?node-id=0-1&t=YfBJ0QG3ZULOTfq5-1

System Components and Modules

1. Client Layer: Mobile Application

The mobile application is the primary interface through which users interact with the virtual card financial system. It is built using cross-platform technologies to ensure consistent user experience across iOS and Android platforms.

Responsibilities:

- Presenting a user-friendly interface for all system features
- Collecting user input for authentication, card requests, and payment transactions
- Communicating with the backend through the API Gateway using HTTPS
- Storing authentication tokens securely on the device
- Displaying real-time notifications received from the Notification module
- Managing local caching of non-sensitive user data for offline functionality

Technology Choices:

- **React Native** or **Flutter** (recommended for performance-critical applications)
- Both frameworks enable code sharing between iOS and Android platforms, reducing development time and maintenance costs
- Native performance and UI components ensure optimal user experience

2. API Gateway

The API Gateway serves as the single entry point for all client requests, implementing critical cross-cutting concerns before requests reach individual microservices.

Responsibilities:

- **Request Routing** - Directing incoming requests to appropriate microservices based on URL paths and HTTP methods
- **Authentication Validation** - Verifying JWT tokens and session validity before allowing requests to proceed

- **Request Validation** - Validating request format, required fields, and data types
- **Rate Limiting** - Preventing abuse by limiting request frequency per user or IP address
- **API Versioning** - Managing multiple API versions to support client updates without service disruption
- **Load Balancing** - Distributing traffic across multiple instances of each microservice
- **Response Aggregation** - Combining responses from multiple services when necessary

Technology Choices:

- **Kong** (open-source API gateway with extensive plugin ecosystem)
- **AWS API Gateway** (managed service with built-in AWS integration)
- **NGINX** (lightweight reverse proxy with powerful routing capabilities)

3. Authentication Module

The Authentication Module manages user identity verification and session management, serving as the security gatekeeper for the entire system.

Responsibilities:

- **User Login** - Validating user credentials (email and password) against stored records
- **Token Generation** - Creating JWT (JSON Web Tokens) for authenticated users
- **Token Validation** - Verifying JWT tokens on subsequent requests
- **Session Management** - Storing and retrieving session information from Redis cache
- **Token Refresh** - Issuing new access tokens using refresh tokens
- **Multi-Factor Authentication** - Supporting additional authentication factors for sensitive operations
- **User Information Retrieval** - Fetching user details from the User Management module

Key Features:

- JWT tokens with configurable expiration (typically 15-30 minutes for access tokens)
- Refresh tokens stored in Redis with longer expiration (7-30 days)

- Secure password hashing using bcrypt or Argon2
- Rate limiting on login attempts to prevent brute force attacks
- Audit logging of authentication events

4. User Management Module

The User Management Module maintains comprehensive user profile information and serves as the central repository for user data accessed by other modules.

Responsibilities:

- **User Profile Storage** - Maintaining user personal information, contact details, and account preferences
- **User Verification** - Recording and retrieving user verification status from KYC processes
- **User Data Retrieval** - Providing user information to other modules during various operation
- **Data Persistence** - Storing all user information in the PostgreSQL database

5. KYC Module

The KYC (Know Your Customer) Module implements regulatory compliance requirements by validating user identity through third-party verification providers.

Responsibilities:

- **User Data Collection** - Gathering necessary user information for identity verification
- **Third-Party Integration** - Communicating with external KYC provider APIs
- **Verification Processing** - Submitting user data to KYC providers and awaiting responses
- **Result Storage** - Recording KYC verification results in the User Management module
- **Status Validation** - Checking user KYC status before allowing sensitive operations

KYC Verification Process:

12 User submits identity documents and personal information through the mobile app
13 KYC Module collects and validates the submitted data
14 User data is sent to the third-party KYC provider API

15 KYC provider performs identity verification using various methods (document scanning, facial recognition, etc.)
16 Verification result is returned to the KYC Module
17 Result is stored in the User Management module
18 User is notified of verification outcome

Third-Party KYC Providers:

- Onfido - AI-powered identity verification
- IDology - Comprehensive identity verification solutions
- Jumio - Real-time identity verification and compliance
- Trulioo - Global identity verification platform

6. Card Provision Module

The Card Provision Module handles virtual card creation requests and integrates with third-party card provider APIs to issue virtual payment cards.

Responsibilities:

- **Card Request Processing** - Handling user requests for virtual card creation
- **KYC Status Validation** - Ensuring users have completed KYC verification before provisioning
- **Third-Party Integration** - Communicating with card provider APIs for card issuance
- **Card Details Storage** - Securely storing card information in the database
- **Card Information Retrieval** - Providing card details to Payment Processing module

Card Provisioning Flow:

19 User requests virtual card through mobile application
20 Card Provision Module validates user's KYC status
21 If KYC approved, module sends card creation request to third-party provider
22 Card provider generates virtual card details (card number, CVV, expiry date)
23 Card details are returned to Card Provision Module
24 Card information is encrypted and stored in database
25 Card details are returned to user through mobile app

Third-Party Card Providers:

- Marqeta - Virtual card issuance platform

- Stripe - Payment processing with virtual cards
- Checkout.com - Global payment solutions
- Adyen - Comprehensive payment platform

7. Payment Processing Module

The Payment Processing Module orchestrates the complete payment transaction flow, coordinating with fraud detection, external payment gateways, and notification systems.

Responsibilities:

- **Transaction Initiation** - Receiving payment requests from the mobile client
- **Card Validation** - Retrieving and validating card details from the database
- **Fraud Checking** - Submitting transactions to Fraud Detection module for analysis
- **Gateway Communication** - Sending approved transactions to third-party payment gateways
- **Transaction Recording** - Storing transaction details and status in the database
- **Notification Triggering** - Initiating notifications for transaction outcomes

Transaction Processing Flow:

26 User initiates payment through mobile application
27 Payment Processing Module receives transaction request
28 Module retrieves card details from database
29 Transaction data is sent to Fraud Detection module
30 Fraud Detection returns risk assessment
31 If transaction is legitimate, it is sent to payment gateway
32 Payment gateway processes transaction and returns status
33 Transaction details are stored in database
34 Notification module is triggered to alert user
35 Transaction result is returned to mobile application

Payment Gateway Integrations:

- Stripe - Comprehensive payment processing
- Square - Payment processing and point-of-sale
- PayPal - Global payment solutions
- 2Checkout - Multi-currency payment processing

8. Fraud Detection Module

The Fraud Detection Module analyzes transaction patterns in real-time to identify and prevent fraudulent activities using machine learning algorithms and rule-based checks.

Responsibilities:

- **Transaction Analysis** - Examining transaction characteristics for suspicious patterns
- **Risk Assessment** - Assigning risk scores to transactions based on multiple factors
- **Anomaly Detection** - Identifying unusual transaction patterns using machine learning
- **Velocity Checking** - Monitoring transaction frequency and amounts
- **Geolocation Verification** - Comparing transaction location with user's known locations
- **Device Fingerprinting** - Tracking devices used for transactions
- **Real-Time Response** - Returning fraud assessment results immediately
- **Transaction Blocking** - Blocking or flagging suspicious transactions

Fraud Detection Factors:

- Transaction amount compared to user's historical patterns
- Transaction location compared to user's known locations
- Transaction frequency (velocity checks)
- Device fingerprint consistency
- Time of day analysis
- Merchant category analysis
- User account age and history
- Multiple failed transaction attempts

Machine Learning Models: The module employs various machine learning approaches:

- **Supervised Learning** - Training models on historical fraud/legitimate transaction data
- **Unsupervised Learning** - Detecting anomalies through clustering and isolation forests
- **Ensemble Methods** - Combining multiple models for improved accuracy

9. Notification Module

The Notification Module delivers real-time push notifications to users, alerting them of transaction status, fraud alerts, and account activities.

Responsibilities:

- **Notification Generation** - Creating notification messages based on system events
- **Push Notification Delivery** - Sending notifications to mobile applications via FCM
- **Multi-Channel Support** - Supporting email, SMS, and push notifications

Notification Events:

- Transaction successful
- Transaction failed
- Fraud alert detected
- KYC verification approved/rejected
- Card provisioning successful
- Account security alerts

Technology:

- **Firebase Cloud Messaging (FCM)** - Google's push notification service supporting iOS and Android
- **SendGrid** - Email notification service
- **Twilio** - SMS notification service

10. Caching Layer (Redis)

The Redis caching layer provides ultra-fast access to frequently used data, reducing database load and improving system responsiveness.

Responsibilities:

- **Session Token Storage** - Caching JWT tokens for fast authentication validation
- **User Data Caching** - Storing frequently accessed user profiles and settings
- **Card Information Caching** - Caching card details for quick retrieval during transactions
- **Cache Invalidation** - Removing stale data based on TTL and event-based triggers

- **Distributed Caching** - Supporting multiple Redis instances for high availability
- **Pub/Sub Messaging** - Enabling real-time event distribution between modules

Caching Strategy:

- **Session Tokens** - TTL of 30 minutes for access tokens, 7 days for refresh tokens
- **User Profiles** - TTL of 1 hour, invalidated on profile updates
- **Card Information** - TTL of 15 minutes, invalidated on card status changes
- **KYC Status** - TTL of 24 hours, invalidated on verification completion

Performance Impact:

- Reduces database queries by 60-80% for read-heavy operations
- Improves API response times from 500ms to under 100ms
- Enables support for 100,000+ concurrent users
- Reduces database server load significantly

11. Database (PostgreSQL)

PostgreSQL serves as the persistent data store for all system information, ensuring data integrity, consistency, and reliability.

Responsibilities:

- **User Data Storage** - Maintaining user profiles, credentials, and account information
- **Card Information Storage** - Storing encrypted card details securely
- **Transaction Recording** - Maintaining complete transaction history
- **KYC Status Storage** - Recording verification results and compliance data
- **Audit Logging** - Tracking all system activities for compliance and security
- **Data Integrity** - Enforcing constraints and relationships between data entities

Key Tables:

- users - User account information
- cards - Virtual card details
- transactions - Transaction history
- kyc_verifications - KYC verification records
- audit_logs - System activity logs
- sessions - Session management data

Database Features:

- ACID compliance for financial transaction integrity
- Row-level security for data isolation
- Full-text search for transaction history
- JSON support for flexible data structures
- Replication for high availability
- Point-in-time recovery for disaster recovery

Technology Stack

Backend Services

Node.js with Express.js

Node.js is the recommended runtime environment for implementing microservices in this architecture. Express.js provides a lightweight and flexible web application framework built on top of Node.js.

Rationale:

- **Non-Blocking I/O** - Node.js's event-driven, non-blocking I/O model is ideal for handling multiple concurrent requests from mobile clients
- **JavaScript Ecosystem** - Extensive npm package ecosystem provides libraries for authentication, validation, API communication, and more
- **Development Speed** - Express.js enables rapid development and deployment of REST APIs
- **JSON Processing** - Native JSON support in JavaScript simplifies API communication
- **Scalability** - Node.js can handle thousands of concurrent connections with minimal resource overhead
- **Microservices Friendly** - Lightweight nature makes it ideal for containerized microservices

Alternative: Python with FastAPI

For modules requiring advanced data processing or machine learning integration (particularly the Fraud Detection module), Python with FastAPI offers advantages:

- Excellent data processing libraries (Pandas, NumPy)
- Mature machine learning frameworks (TensorFlow, scikit-learn)
- Strong statistical analysis capabilities
- Rapid API development with automatic documentation

API Gateway

Kong or AWS API Gateway

The API Gateway is a critical component that handles request routing, authentication, rate limiting, and API management.

Kong (Open-Source)

- Extensible plugin architecture
- Built-in authentication and rate limiting
- High performance and scalability
- Community support and active development
- Can be self-hosted for maximum control

AWS API Gateway (Managed Service)

- Fully managed service with no operational overhead
- Native integration with AWS services
- Built-in authentication and authorization
- CloudWatch integration for monitoring
- Pay-per-use pricing model

Database

PostgreSQL

PostgreSQL is the recommended relational database for this system due to its robustness, feature richness, and suitability for financial applications.

Rationale:

- **ACID Compliance** - Ensures data consistency and reliability critical for financial transactions
- **Data Integrity** - Enforces constraints and relationships between data entities
- **Advanced Features** - JSON support, full-text search, and custom data types
- **Replication** - Built-in replication for high availability and disaster recovery
- **Security** - Row-level security, encryption at rest, and comprehensive audit logging
- **Performance** - Efficient indexing and query optimization for fast data retrieval
- **Maturity** - Stable, well-documented, and widely used in production systems

Alternative: MongoDB

For non-transactional data like user profiles and settings, MongoDB offers:

- Flexible schema for evolving requirements
- Horizontal scalability through sharding
- Document-oriented storage matching API responses

- However, MongoDB lacks ACID guarantees needed for financial transactions

Recommendation: Use PostgreSQL as the primary database for all transactional data, with optional MongoDB for non-critical data like user preferences and analytics.

Caching Layer

Redis

Redis is the recommended in-memory data store for session management, user data caching, and real-time event distribution.

Rationale:

- **Performance** - In-memory storage provides microsecond-level latency
- **Data Structures** - Supports strings, lists, sets, hashes, and sorted sets for flexible caching
- **TTL Support** - Automatic data expiration based on time-to-live settings
- **Pub/Sub** - Built-in publish/subscribe for real-time event distribution
- **Persistence** - Optional persistence to disk for data durability
- **Replication** - Master-slave replication for high availability
- **Cluster Mode** - Horizontal scaling across multiple Redis instances

Use Cases:

- Session token storage (access tokens, refresh tokens)
- User profile caching
- Card information caching
- Rate limiting counters
- Real-time event distribution
- Leaderboards and sorted data

Mobile Application

React Native (Recommended for JavaScript Teams)

React Native enables development of native iOS and Android applications using JavaScript and React.

Advantages:

- **Code Sharing** - Single codebase for iOS and Android reduces development time
- **Native Performance** - Compiles to native code for optimal performance
- **Rich Ecosystem** - Extensive library ecosystem for UI components, navigation, and utilities
- **Developer Experience** - Hot reload for rapid development iteration
- **Community** - Large and active community with abundant resources

Flutter (Recommended for Performance-Critical Applications)

Flutter is Google's UI framework for building native applications for mobile, web, and desktop.

Advantages:

- **Performance** - Compiles to native code with minimal overhead
- **UI Consistency** - Material Design components ensure consistent UI across platforms
- **Development Speed** - Hot reload and hot restart enable rapid iteration
- **Rich Widgets** - Comprehensive widget library for building complex UIs
- **Dart Language** - Modern, type-safe language with excellent tooling

Recommendation: Choose React Native if your team has JavaScript expertise; choose Flutter for performance-critical applications or if your team prefers Dart.

Authentication and Security

JWT (JSON Web Tokens)

JWT provides stateless authentication suitable for distributed microservices architectures.

Implementation:

- Access tokens with short expiration (15-30 minutes)
- Refresh tokens with longer expiration (7-30 days) stored in Redis
- Tokens signed with RS256 (RSA Signature with SHA-256)
- Token validation on every request through API Gateway

OAuth 2.0

OAuth 2.0 enables secure authorization for third-party integrations and social login.

Use Cases:

- Third-party application access to user data
- Social login (Google, Apple, Facebook)
- Delegated authorization for partner integrations

Password Security:

- Bcrypt or Argon2 for password hashing
- Minimum 12-character passwords with complexity requirements
- Regular password expiration and change requirements
- Secure password reset flow with email verification

Notification Service

Firebase Cloud Messaging (FCM)

Firebase Cloud Messaging provides reliable push notification delivery to iOS and Android applications.

Rationale:

- **Cross-Platform** - Single service for iOS and Android notifications
- **Reliability** - High delivery rate with built-in retry logic
- **Free Tier** - No cost for reasonable notification volumes
- **Integration** - Easy integration with mobile applications
- **Analytics** - Built-in analytics for notification delivery and user engagement
- **Segmentation** - Ability to target notifications to specific user groups

Alternative: OneSignal

OneSignal offers enhanced analytics and segmentation features:

- Advanced user segmentation
- A/B testing for notifications
- Detailed analytics and reporting
- Email and SMS support in addition to push notifications

Inter-Module Communication

Communication Protocol

All modules communicate through **REST APIs** using the HTTP/HTTPS protocol. This approach provides several advantages for a distributed microservices architecture:

Loose Coupling - Modules operate independently without direct dependencies on internal implementations of other modules. Services can be updated or replaced without affecting dependent modules as long as the API contract remains stable.

Technology Agnostic - Each module can be implemented using different programming languages and frameworks. For example, the Fraud Detection module could be implemented in Python for machine learning capabilities while other modules use Node.js.

Standard Protocols - HTTP/HTTPS are universally supported across all platforms and languages, enabling seamless integration with third-party services and future extensions.

Easy Testing - Each API endpoint can be tested independently using standard HTTP testing tools, enabling comprehensive test coverage and rapid development cycles.

Scalability - REST APIs enable horizontal scaling where multiple instances of a service can handle requests independently, with load balancers distributing traffic.

Data Exchange Format

JSON (JavaScript Object Notation) is the standard format for all data exchange between modules and with external APIs.

Advantages:

- **Lightweight** - Minimal overhead compared to XML or other formats
- **Human-Readable** - Easy to understand and debug
- **Native Support** - JavaScript and most modern languages have native JSON support
- **Flexible** - Supports complex nested structures and arrays
- **Standard** - Widely adopted across REST APIs and web services

Example Request:

```
{
  "email": "user@example.com",
  "password": "securePassword123",
  "device_id": "device_uuid_123"
}
```

Example Response:

```
{
  "success": true,
  "token": "eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9 ... ",
  "user": {
    "id": "user_uuid_123",
    "email": "user@example.com",
    "first_name": "John",
    "last_name": "Doe",
    "kyc_status": "approved"
  }
}
```

Communication Patterns

The system employs three primary communication patterns depending on the nature of the operation:

Synchronous REST Calls - Used for operations requiring immediate responses and real-time interaction. Examples include user authentication, payment processing, and card provisioning. The calling module waits for a response before proceeding.

Asynchronous Processing - Used for operations that don't require immediate responses. Examples include fraud detection analysis and notification delivery. The calling module submits a request and continues processing without waiting for completion.

Event-Driven Architecture - Used for notifications and real-time updates. Modules publish events that other modules subscribe to, enabling loose coupling and

scalability. For example, when a transaction completes, the Payment Processing module publishes an event that the Notification module subscribes to.

API Versioning

To support client updates and backward compatibility, the API implements versioning through URL paths:

```
/api/v1/auth/login
```

```
/api/v2/auth/login
```

This approach enables:

- Supporting multiple client versions simultaneously
- Gradual migration of clients to new API versions
- Deprecation of old API versions with advance notice
- Testing of new API versions before full deployment

Request-Response Architecture

User Login Flow

The user login process demonstrates the complete request-response flow through the system:

Step 1: Client Initiates Login Request The mobile application collects user credentials (email and password) and sends a POST request to the API Gateway:

```
POST /api/v1/auth/login
```

```
Host: api.virtualcards.com
Content-Type: application/json
Body: {
  "email": "user@example.com",
  "password": "securePassword123"
```

```
}
```

Step 2: API Gateway Routes Request The API Gateway receives the login request and routes it to the Authentication Module after basic validation.

Step 3: Authentication Module Checks Cache The Authentication Module first checks Redis cache for an existing valid session token for the user. If found, it returns the cached token without further processing.

Step 4: Authentication Module Queries User Management If no cached token exists, the Authentication Module calls the User Management module to retrieve user credentials:

```
GET /api/v1/users/verify
```

```
Headers: { Authorization: Bearer internal_service_token }
```

```
Query: { email: "user@example.com" }
```

Step 5: User Management Queries Database The User Management module queries the PostgreSQL database to retrieve the user record:

```sql
SELECT * FROM users WHERE email = 'user@example.com';
```

Step 6: Database Returns User Data The database returns the user record including the password hash:

```
{
```

```
  "user_id": "uuid_123",
  "email": "user@example.com",
  "password_hash": "$2b$12$ ... ",
  "first_name": "John",
  "kyc_status": "approved"
```

```
}
```

Step 7: User Management Returns Verification Result The User Management module returns the user data to the Authentication Module:

```
{
  "user_id": "uuid_123",
  "email": "user@example.com",
  "first_name": "John",
  "kyc_status": "approved",
  "verified": true
}
```

Step 8: Authentication Module Validates Password The Authentication Module compares the provided password with the stored hash using bcrypt:

```
const passwordMatch = await bcrypt.compare(providedPassword, storedHash);
```

Step 9: Authentication Module Stores Token in Cache Upon successful password validation, the Authentication Module generates a JWT token and stores it in Redis with a 30-minute TTL:

```
SET session:user_uuid_123 "eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9..." EX
1800
```

Step 10: Authentication Module Returns Token to API Gateway The Authentication Module returns the JWT token and user information to the API Gateway:

```
{
  "success": true,
  "token": "eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9 ... ",
  "user": {
    "id": "uuid_123",
    "email": "user@example.com",
    "first_name": "John",
    "kyc_status": "approved"
  }
```

```
}
```

Step 11: API Gateway Returns Response to Client The API Gateway returns the authentication response to the mobile client:

```
{
  "success": true,
  "token": "eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9 ... ",
  "user": {
    "id": "uuid_123",
    "email": "user@example.com",
    "first_name": "John",
    "kyc_status": "approved"
  }
}
```

Step 12: Mobile Client Stores Token The mobile application stores the JWT token securely (using Keychain on iOS or Keystore on Android) for use in subsequent requests.

Virtual Card Request Flow

The virtual card request process demonstrates interaction with external APIs and KYC validation:

Step 1: User Requests Card The mobile application sends a card request to the API Gateway:

```
POST /api/v1/cards/request
```

```
Headers: { Authorization: Bearer <jwt_token> }
Body: {
  "card_type": "virtual_visa",
  "currency": "USD"
}
```

Step 2: API Gateway Validates Token The API Gateway validates the JWT token by checking Redis cache and verifying the token signature.

Step 3: API Gateway Routes to Card Provision Module The validated request is routed to the Card Provision Module.

Step 4: Card Provision Module Checks KYC Status The Card Provision Module calls the KYC Module to verify the user's KYC status:

```
GET /api/v1/kyc/status/user_uuid_123
```

```
Headers: { Authorization: Bearer internal_service_token }
```

Step 5: KYC Module Queries User Management The KYC Module retrieves the user's KYC status from the User Management module:

```
GET /api/v1/users/user_uuid_123/kyc-status
```

Step 6: KYC Module Returns Status The KYC Module returns the user's KYC verification status:

```
{
```

```
  "kyc_status": "approved",
  "verified_date": "2025-10-15",
  "verification_provider": "onfido"
```

```
}
```

Step 7: Card Provision Module Calls Card Provider API If KYC is approved, the Card Provision Module sends a card creation request to the third-party card provider:

```
POST https://api.cardprovider.com/v1/cards
```

```
Headers: {
  "Authorization": "Bearer provider_api_key",
  "Content-Type": "application/json"
}
Body: {
```

```
  "user_id": "user_uuid_123",
  "card_type": "virtual_visa",
  "currency": "USD",
  "amount_limit": 10000
}
```

Step 8: Card Provider Returns Card Details The card provider generates and returns virtual card details:

```
{
  "card_id": "card_uuid_456",
  "card_number": "4532123456789012",
  "cvv": "123",
  "expiry_date": "10/27",
  "status": "active"
}
```

Step 9: Card Provision Module Encrypts and Stores Card Details The Card Provision Module encrypts the card details using AES-256 encryption and stores them in the database:

```
INSERT INTO cards (card_id, user_id, card_number_encrypted,
cvv_encrypted, expiry_date, status)
VALUES ('card_uuid_456', 'user_uuid_123', 'encrypted_card_number',
'encrypted_cvv', '10/27', 'active');
```

Step 10: Card Provision Module Returns Card to Client The Card Provision Module returns the card details to the mobile client (with sensitive fields masked):

```
{
  "success": true,
  "card": {
    "card_id": "card_uuid_456",
    "card_number": "****9012",
    "cvv": "***",
    "expiry_date": "10/27",
    "status": "active"
```

```
  }
}
```

Payment Processing Flow

The payment processing flow demonstrates real-time fraud detection and external gateway integration:

Step 1: User Initiates Payment The mobile application sends a payment request:

```
POST /api/v1/payments/process
```

```
Headers: { Authorization: Bearer <jwt_token> }
Body: {
  "card_id": "card_uuid_456",
  "amount": 99.99,
  "currency": "USD",
  "merchant": "example-merchant.com",
  "merchant_category": "electronics"
}
```

Step 2: API Gateway Validates Request The API Gateway validates the JWT token and request format.

Step 3: Payment Processing Module Retrieves Card Details The Payment Processing Module retrieves the card details from the database:

```
SELECT card_number, cvv, expiry_date FROM cards WHERE card_id =
'card_uuid_456' AND user_id = 'user_uuid_123';
```

Step 4: Payment Processing Module Calls Fraud Detection The Payment Processing Module sends the transaction to the Fraud Detection module for analysis:

```
POST /api/v1/fraud/check
```

```
Headers: { Authorization: Bearer internal_service_token }
Body: {
```

```
  "user_id": "user_uuid_123",
  "amount": 99.99,
  "merchant": "example-merchant.com",
  "merchant_category": "electronics",
  "device_id": "device_uuid_123",
  "ip_address": "192.168.1.1",
  "timestamp": "2025-10-20T14:30:00Z"
}
```

Step 5: Fraud Detection Analyzes Transaction The Fraud Detection module analyzes the transaction using machine learning models and rule-based checks:

- Compares amount to user's historical patterns
- Checks transaction location against known locations
- Validates device fingerprint
- Checks for velocity violations
- Analyzes merchant category

Step 6: Fraud Detection Returns Risk Assessment The Fraud Detection module returns a risk assessment:

```
{
  "risk_score": 0.15,
  "risk_level": "low",
  "approved": true,
  "checks": {
    "amount_check": "pass",
    "location_check": "pass",
    "device_check": "pass",
    "velocity_check": "pass"
  }
}
```

Step 7: Payment Processing Sends to Payment Gateway If the transaction is approved, the Payment Processing Module sends it to the payment gateway:

```
POST https://api.paymentgateway.com/v1/transactions
Headers: {
  "Authorization": "Bearer gateway_api_key",
```

```
  "Content-Type": "application/json"
}
Body: {
  "card_number": "4532123456789012",
  "cvv": "123",
  "expiry_date": "10/27",
  "amount": 99.99,
  "currency": "USD",
  "merchant": "example-merchant.com",
  "idempotency_key": "transaction_uuid_789"
}
```

Step 8: Payment Gateway Processes Transaction The payment gateway processes the transaction through the card networks and returns the result:

```
{
  "transaction_id": "txn_uuid_789",
  "status": "success",
  "amount": 99.99,
  "currency": "USD",
  "timestamp": "2025-10-20T14:30:05Z"
}
```

Step 9: Payment Processing Stores Transaction Details The Payment Processing Module stores the transaction in the database:

```
INSERT INTO transactions (transaction_id, user_id, card_id, amount,
currency, merchant, status, created_at)
VALUES ('txn_uuid_789', 'user_uuid_123', 'card_uuid_456', 99.99, 'USD',
'example-merchant.com', 'success', NOW());
```

Step 10: Payment Processing Triggers Notification The Payment Processing Module calls the Notification Module to alert the user:

```
POST /api/v1/notifications/send
Headers: { Authorization: Bearer internal_service_token }
Body: {
```

```
  "user_id": "user_uuid_123",
  "type": "transaction_success",
  "title": "Payment Successful",
  "message": "Your payment of $99.99 to example-merchant.com was
successful",
  "data": {
    "transaction_id": "txn_uuid_789",
    "amount": 99.99,
    "merchant": "example-merchant.com"
  }
```

```
}
```

Step 11: Notification Module Sends Push Notification The Notification Module sends a push notification to the user's mobile device via Firebase Cloud Messaging:

```
POST https://fcm.googleapis.com/fcm/send
```

```
Headers: {
  "Authorization": "key=server_api_key",
  "Content-Type": "application/json"
}
Body: {
  "to": "device_fcm_token_123",
  "notification": {
    "title": "Payment Successful",
    "body": "Your payment of $99.99 was successful"
  },
  "data": {
    "transaction_id": "txn_uuid_789",
    "amount": "99.99"
  }
```

```
}
```

Step 12: Payment Processing Returns Result to Client The Payment Processing Module returns the transaction result to the mobile client:

```
{
```

```
  "success": true,
  "transaction": {
    "transaction_id": "txn_uuid_789",
    "status": "success",
    "amount": 99.99,
    "currency": "USD",
```

```
    "merchant": "example-merchant.com",
    "timestamp": "2025-10-20T14:30:05Z"
  }
}
```

Step 13: Mobile Client Displays Confirmation The mobile application displays a transaction confirmation to the user.

Data Flow Patterns

Request-Response Cycle

The fundamental data flow pattern in this architecture follows a request-response cycle:

36 **Client Initiates Request** - Mobile application sends HTTP request to API Gateway with user action
37 **API Gateway Processes** - Validates request, checks authentication, routes to appropriate module
38 **Module Processing** - Microservice processes request, may call other modules or external APIs
39 **Data Persistence** - Information is stored in database or cache as needed
40 **Response Generation** - Module generates response with results or error information
41 **Client Receives Response** - Mobile application receives response and updates UI

Synchronous vs. Asynchronous Processing

Synchronous Processing is used for operations requiring immediate responses:

- User authentication
- Card provisioning
- Payment processing
- User profile updates

In synchronous processing, the calling module waits for a response before proceeding, ensuring immediate feedback to the user.

Asynchronous Processing is used for operations that don't require immediate responses:

- Fraud detection analysis (can run in parallel)
- Notification delivery
- Audit logging
- Data cleanup and maintenance

In asynchronous processing, the calling module submits a request and continues processing without waiting for completion, improving system responsiveness.

Caching Strategy

The system implements a multi-level caching strategy to optimize performance:

Session Token Caching - JWT tokens are cached in Redis with 30-minute TTL, enabling fast authentication validation without database queries.

User Profile Caching - User profile information is cached with 1-hour TTL, reducing database load for frequently accessed user data.

Card Information Caching - Card details are cached with 15-minute TTL, enabling fast card retrieval during payment processing.

Cache Invalidation - Cached data is invalidated when:

- TTL expires
- User updates their profile
- Card status changes
- KYC status changes

Error Handling and Retry Logic

The system implements comprehensive error handling:

Transient Errors - Temporary failures (network timeouts, service unavailable) are retried with exponential backoff:

- First retry after 1 second

- Second retry after 2 seconds
- Third retry after 4 seconds
- Maximum 3 retries before failure

Permanent Errors - Non-recoverable errors (invalid input, authentication failure) are returned immediately to the client.

Circuit Breaker Pattern - If a service consistently fails, the circuit breaker opens to prevent cascading failures:

- Open state: Requests fail immediately without calling the service
- Half-open state: Limited requests are allowed to test service recovery
- Closed state: Normal operation resumes

Security Considerations

Authentication and Authorization

JWT Token Security - Tokens are signed using RS256 (RSA Signature with SHA-256) with private keys stored securely in environment variables or key management services.

Token Expiration - Access tokens expire after 15-30 minutes, requiring users to refresh tokens using refresh tokens stored in Redis.

Multi-Factor Authentication - For sensitive operations (card provisioning, payment above threshold), users must complete additional authentication factors.

Role-Based Access Control - Users have roles (customer, merchant, admin) with specific permissions enforced by the API Gateway.

Data Protection

End-to-End Encryption - All communication between client and server uses TLS/SSL encryption (HTTPS).

Database Encryption - Sensitive data (card details, PINs) is encrypted at rest using AES-256 encryption.

PCI-DSS Compliance - Card data handling follows Payment Card Industry Data Security Standard requirements:

- Encrypted storage of card details
- Restricted access to card data
- Regular security audits
- Secure deletion of card data

API Security

Rate Limiting - API Gateway implements rate limiting to prevent abuse:

- 100 requests per minute per user
- 1000 requests per minute per IP address
- Stricter limits for sensitive endpoints (login, payment)

Input Validation - All user input is validated and sanitized to prevent injection attacks:

- Email format validation
- Password complexity requirements
- Amount validation (positive numbers, reasonable limits)
- SQL injection prevention through parameterized queries

CORS Policies - Cross-Origin Resource Sharing policies restrict API access to authorized domains.

Third-Party Integration Security

API Key Management - API keys for third-party services are stored in secure vaults and rotated regularly.

Webhook Signature Verification - Webhooks from third-party services are verified using cryptographic signatures.

IP Whitelisting - Where applicable, third-party API calls are restricted to whitelisted IP addresses.

Scalability and Performance

Horizontal Scaling

Microservices can be scaled independently based on load:

Payment Processing Module - During peak transaction times, multiple instances handle payment requests with load balancers distributing traffic.

Fraud Detection Module - During high-transaction-volume periods, additional instances analyze transactions in parallel.

Notification Module - Multiple instances deliver notifications concurrently to thousands of users.

Database Optimization

Read Replicas - Read-only database replicas distribute query load, enabling high-concurrency read operations.

Connection Pooling - Database connection pools reuse connections, reducing overhead of establishing new connections.

Indexing - Strategic indexes on frequently queried columns (user_id, card_id, transaction_id) enable fast data retrieval.

Query Optimization - Queries are optimized to minimize execution time and database resource usage.

Conclusion

The Virtual Card Financial System architecture provides a comprehensive, scalable, and secure foundation for delivering virtual payment card services to mobile users. The microservices architecture with REST API communication enables independent development, deployment, and scaling of components while maintaining system cohesion.

The carefully selected technology stack—Node.js with Express.js for backend services, PostgreSQL for data persistence, Redis for high-performance caching, and

React Native or Flutter for mobile clients—balances performance, developer productivity, and operational reliability. The architecture incorporates multiple layers of security including JWT authentication, encryption at rest and in transit, and real-time fraud detection to protect sensitive financial data.

The request-response architecture demonstrates how data flows through the system from mobile client through API Gateway, microservices, databases, and external integrations. Clear separation of concerns, comprehensive error handling, and strategic caching ensure optimal performance and user experience.

This architecture is designed to support 100,000+ concurrent users, process 10,000+ transactions per second, and maintain 99.9% uptime while meeting the stringent security and compliance requirements of modern financial systems.

References

42 Node.js Official Documentation - https://nodejs.org/en/docs/

43 Express.js Documentation - https://expressjs.com/

44 PostgreSQL Official Documentation - https://www.postgresql.org/docs/

45 Redis Official Documentation - https://redis.io/documentation

46 Firebase Cloud Messaging Documentation - https://firebase.google.com/docs/cloud-messaging

47 JWT (JSON Web Tokens) - https://jwt.io/

48 OAuth 2.0 Authorization Framework - https://tools.ietf.org/html/rfc6749

49 React Native Documentation - https://reactnative.dev/docs/getting-started

50 Flutter Documentation - https://flutter.dev/docs

51 Kong API Gateway Documentation - https://docs.konghq.com/

52 Kubernetes Documentation - https://kubernetes.io/docs/

53 Docker Documentation - https://docs.docker.com/

54 OWASP API Security - https://owasp.org/www-project-api-security/

55 PCI DSS Compliance Guide - https://www.pcisecuritystandards.org/

56 Microservices Architecture - https://microservices.io/