3. System Design

3.1 Architectural Patterns

FalsoPay implements a combination of architectural patterns to create a robust, scalable, and maintainable payment system. Below are the key architectural patterns used and the rationale behind each choice:

Microservices Architecture

FalsoPay is built using a microservices architecture, which divides the application into loosely coupled, independently deployable services.

Why this pattern?

- **Scalability**: Each service can be scaled independently based on demand
- **Resilience**: Failure in one service doesn't bring down the entire system
- **Technology Flexibility**: Different services can use different technologies based on specific requirements
- **Independent Development**: Teams can work on different services simultaneously without tight coordination
- **Easier Maintenance**: Smaller, focused codebases are easier to understand and maintain

API Gateway Pattern

An API Gateway serves as the single entry point for all client requests, routing them to appropriate microservices.

Why this pattern?

- **Simplified Client Interface**: Clients only need to know about a single endpoint
- **Security**: Centralized authentication and authorization
- **Cross-cutting Concerns**: Handles common functionality like logging, rate limiting, and monitoring
- **Protocol Translation**: Can translate between web protocols and internal protocols
- **Request Aggregation**: Can combine results from multiple services for a single client request

Event-Driven Architecture

FalsoPay uses event-driven architecture for handling transactions and notifications.

Why this pattern?

- **Decoupling**: Services communicate through events without direct dependencies
- **Asynchronous Processing**: Enables handling of high-volume transactions without blocking
- **Real-time Updates**: Facilitates immediate notifications for account activities
- **Audit Trail**: Events provide a natural audit log for all system activities
- **Scalability**: Easily scales to handle varying loads of transactions

CQRS (Command Query Responsibility Segregation)

The system separates read operations (queries) from write operations (commands).

Why this pattern?

- **Performance Optimization**: Read and write operations can be optimized separately
- **Scalability**: Read-heavy operations can be scaled independently from write operations
- **Security**: Easier to implement different security models for reads vs. writes
- **Simplified Models**: Simpler domain models for specific operations

3.2 System Architecture

The FalsoPay system consists of the following major components:

1. **User Management Service**
    - Handles user registration, authentication, and profile management
    - Manages user roles and permissions
2. **Payment Processing Service**
    - Processes money transfers between accounts
    - Handles different payment methods
    - Implements transaction security measures
3. **Account Management Service**
    - Manages bank account linking
    - Handles balance inquiries
    - Processes transaction limits
4. **Notification Service**
    - Sends transaction confirmations
    - Delivers security alerts
    - Manages notification preferences
5. **Transaction History Service**
    - Stores and retrieves transaction records
    - Provides filtering and search capabilities
    - Generates transaction reports
6. **Security Service**
    - Implements PIN verification
    - Manages transaction blocking/unblocking
    - Detects suspicious activities
7. **Customer Support Service**
    - Handles support ticket creation and management
    - Provides system status information
    - Manages user reports
8. **API Gateway**
    - Routes client requests to appropriate services
    - Handles authentication and authorization
    - Implements rate limiting and monitoring
9. **Event Bus**

- Facilitates asynchronous communication between services
- Ensures reliable event delivery
- Supports event sourcing for audit trails

Integration Points

FalsoPay integrates with several external systems:

1. **Banking APIs**
   - For bank account verification
   - For processing transfers to/from bank accounts
2. **Payment Networks**
   - For processing card payments
   - For interoperability with other payment systems
3. **Regulatory Compliance Systems**
   - For KYC (Know Your Customer) verification
   - For AML (Anti-Money Laundering) checks

Deployment Architecture

FalsoPay uses a cloud-native deployment approach:

1. **Containerization**: All services are containerized using Docker
2. **Orchestration**: Kubernetes manages container deployment and scaling
3. **Infrastructure as Code**: Terraform scripts define and provision infrastructure
4. **CI/CD Pipeline**: Automated testing and deployment processes
5. **Multi-region Deployment**: For high availability and disaster recovery

Security Architecture

Security is implemented at multiple levels:

1. **Network Level**: Firewalls, VPNs, and network segmentation
2. **Application Level**: Input validation, output encoding, and secure coding practices
3. **Data Level**: Encryption at rest and in transit
4. **Identity Level**: Multi-factor authentication and role-based access control
5. **Monitoring Level**: Intrusion detection and security information and event management

3.3 Class Classification

FalsoPay's classes are categorized into the following classifications:

1. **Entity Classes**
   - User
   - Transaction
   - BankAccount
   - Card
   - SupportTicket
   - Notification
2. **Boundary Classes**
   - UserInterface
   - APIGateway
   - PaymentProcessor
   - NotificationSender
   - ReportGenerator
3. **Control Classes**
   - AuthenticationController
   - TransactionController
   - AccountController
   - NotificationController
   - SecurityController
4. **Service Classes**
   - UserService
   - TransactionService
   - AccountService
   - NotificationService
   - SecurityService
5. **Repository Classes**
   - UserRepository
   - TransactionRepository
   - AccountRepository
   - NotificationRepository
   - SecurityRepository

3.4 Interaction Patterns

The system employs several key interaction patterns:

1. **Request-Response**: Used for synchronous operations like balance inquiries
2. **Publish-Subscribe**: Used for notifications and event propagation
3. **Command-Query**: Implementing CQRS for optimized data operations
4. **Observer Pattern**: For real-time updates of transaction status
5. **Chain of Responsibility**: For processing transaction validations

3.5 Design Patterns

FalsoPay implements several design patterns:

1. **Factory Method**: For creating different types of payment methods
2. **Singleton**: For global services like authentication and logging
3. **Strategy**: For implementing different transaction processing algorithms
4. **Observer**: For notification handling
5. **Decorator**: For adding security layers to transactions

6. **Adapter**: For integrating with different banking APIs
7. **Facade**: For simplifying complex subsystems
8. **Command**: For encapsulating transaction operations