**InterPay Payment System - Architecture Documentation**

**📋 Table of Contents**

1. [System Overview](https://claude.ai/chat/4350c4c2-95b2-4d29-a5c8-548b75c2269c#system-overview)
2. [Database Architecture](https://claude.ai/chat/4350c4c2-95b2-4d29-a5c8-548b75c2269c#database-architecture)
3. [Data Flow Patterns](https://claude.ai/chat/4350c4c2-95b2-4d29-a5c8-548b75c2269c#data-flow-patterns)
4. [Consistency Strategies](https://claude.ai/chat/4350c4c2-95b2-4d29-a5c8-548b75c2269c#consistency-strategies)
5. [Scaling & Performance](https://claude.ai/chat/4350c4c2-95b2-4d29-a5c8-548b75c2269c#scaling-performance)
6. [Security Measures](https://claude.ai/chat/4350c4c2-95b2-4d29-a5c8-548b75c2269c#security-measures)

**🏗️ System Overview**

**Microservices Architecture**

┌─────────────────────────────────────────────────────────────┐

│ API Gateway / Load Balancer │

└─────────────────┬───────────────────────────────────────────┘

│

┌─────────────┼─────────────┬─────────────┬──────────────┐

▼ ▼ ▼ ▼ ▼

┌────────┐ ┌──────────┐ ┌─────────┐ ┌──────────┐ ┌─────────┐

│ User │ │ Wallet │ │ Payment │ │ Merchant │ │ Fraud │

│Service │ │ Service │ │ Service │ │ Service │ │Detection│

└───┬────┘ └────┬─────┘ └────┬────┘ └────┬─────┘ └────┬────┘

│ │ │ │ │

▼ ▼ ▼ ▼ ▼

┌────────┐ ┌──────────┐ ┌─────────┐ ┌──────────┐ ┌─────────┐

│ PG │ │ PG │ │ PG │ │ PG │ │ PG │

│Database│ │ Database │ │Database │ │ Database │ │Database │

└────────┘ └──────────┘ └─────────┘ └──────────┘ └─────────┘

**Technology Stack**

* **Relational Databases**: PostgreSQL (transactional data)
* **NoSQL Database**: Apache Cassandra (logs, sessions, time-series)
* **Cache Layer**: Redis (high-speed caching, rate limiting)
* **Message Queue**: Kafka/RabbitMQ (async communication)
* **Service Mesh**: Istio (service-to-service communication)

**💾 Database Architecture**

**1. User Service Database**

**Purpose**: User identity, authentication, and profile management

**Key Tables**:

* users - Core user accounts with authentication
* user\_profiles - Personal information (GDPR compliant)
* kyc\_documents - KYC verification records

**Design Decisions**:

* Separate authentication from profile data for security
* Email and phone as unique identifiers
* Soft deletes with status flags for compliance
* Optimistic locking with version column

**2. Wallet Service Database**

**Purpose**: Digital wallet management and balance tracking

**Key Tables**:

* wallets - Main wallet with balance fields
* wallet\_transactions - Immutable transaction ledger
* wallet\_holds - Temporary holds for pending payments

**Critical Features**:

-- Balance types explained:

-- balance: Total balance (available + pending)

-- available\_balance: Can be used immediately

-- pending\_balance: Held for pending transactions

**Double-Entry Bookkeeping**: Every transaction creates two entries:

* Debit from sender wallet
* Credit to receiver wallet

**Concurrency Control**:

-- Optimistic locking prevents race conditions

UPDATE wallets

SET balance = balance - amount,

version = version + 1

WHERE wallet\_id = ? AND version = ?

**3. Payment Service Database**

**Purpose**: Process payments through various methods

**Key Tables**:

* payments - Main payment records
* payment\_methods - Saved payment instruments
* cards - Tokenized card data (PCI compliant)
* payment\_transactions - Transaction state machine
* refunds - Refund processing

**Payment Flow States**:

INITIATED → PENDING → AUTHORIZED → CAPTURED

↓ ↓

FAILED CANCELLED → REFUNDED

**Tokenization Strategy**:

* Card numbers never stored in plain text
* Use vault service (e.g., HashiCorp Vault)
* Only last 4 digits and token stored

**4. Transfer Service Database**

**Purpose**: Wallet-to-wallet and payout transfers

**Key Tables**:

* transfers - P2P transfers and payouts

**Idempotency Pattern**:

-- Use unique reference ID to prevent duplicate transfers

INSERT INTO transfers (transfer\_id, ...)

VALUES (?, ...)

ON CONFLICT (transfer\_id) DO NOTHING

**5. Merchant Service Database**

**Purpose**: Merchant onboarding and settlement

**Key Tables**:

* merchants - Merchant accounts
* merchant\_api\_keys - API credentials
* merchant\_settlements - Batch settlements

**API Key Security**:

* Public key for identification
* Secret key hashed (never stored plain)
* Scoped permissions (JSONB)
* Automatic expiration

**6. Fraud Detection Service Database**

**Purpose**: Real-time fraud prevention

**Key Tables**:

* fraud\_rules - Configurable rule engine
* fraud\_checks - Every transaction checked
* blacklist - Blocked entities

**Rule Types**:

* **Velocity**: Transaction frequency limits
* **Amount**: Unusual transaction amounts
* **Location**: Geographic anomalies
* **Device**: Device fingerprinting
* **Pattern**: ML-based pattern detection

**Risk Scoring**:

Risk Score = Σ(triggered\_rule.weight × rule.confidence)

**7. Cassandra for Logs & Sessions**

**Use Cases**:

* High-write throughput (millions of logs/day)
* Time-series data (metrics, analytics)
* Session storage with TTL
* API request/response logging

**Key Features**:

-- Automatic data expiration

WITH default\_time\_to\_live = 86400

-- Efficient time-based queries

PRIMARY KEY ((partition\_key), timestamp)

WITH CLUSTERING ORDER BY (timestamp DESC)

**Partition Strategy**:

* Use composite partition keys
* Keep partitions under 100MB
* Use time bucketing (day/hour)

**8. Redis Caching Strategy**

**Cache Patterns**:

1. Cache-Aside (Lazy Loading):

- Check cache → Miss → Load from DB → Cache it

2. Write-Through:

- Write to DB → Update cache

3. Write-Behind:

- Write to cache → Async write to DB

**Key Patterns**:

# User profile cache

SET user:12345:profile "{...json...}" EX 3600

# Wallet balance (frequently accessed)

SET wallet:67890:balance "1234.56" EX 300

# Rate limiting

INCR api:ratelimit:merchant123:2025-09-30-14:30 EX 60

# Distributed lock

SET lock:transfer:tx123 "processing" NX EX 30

**Cache Invalidation**:

* Time-based (TTL)
* Event-based (message queue triggers)
* Manual (admin operations)

**🔄 Data Flow Patterns**

**Payment Flow Example**

┌─────────┐ ┌─────────┐ ┌──────────┐ ┌──────────┐

│ Client │────→│Payment │────→│ Fraud │────→│ Wallet │

│ │ │ Service │ │ Detection│ │ Service │

└─────────┘ └─────────┘ └──────────┘ └──────────┘

│ │ │

▼ ▼ ▼

┌─────────┐ ┌────────┐ ┌────────┐

│ Payment │ │ Fraud │ │ Wallet │

│ DB │ │ DB │ │ DB │

└─────────┘ └────────┘ └────────┘

**Step-by-Step**:

1. **Initiate Payment**
2. INSERT INTO payments (status='INITIATED')
3. **Fraud Check**
4. INSERT INTO fraud\_checks (risk\_score=calculated)
5. IF risk\_score > threshold THEN DECLINE
6. **Hold Funds** (if wallet payment)
7. BEGIN TRANSACTION;
8. UPDATE wallets SET available\_balance = available\_balance - amount;
9. INSERT INTO wallet\_holds (status='ACTIVE');
10. COMMIT;
11. **Authorize Payment**
12. UPDATE payments SET status='AUTHORIZED'
13. **Capture/Complete**
14. BEGIN TRANSACTION;
15. UPDATE payments SET status='CAPTURED';
16. -- Release hold and transfer funds
17. UPDATE wallet\_holds SET status='CAPTURED';
18. INSERT INTO wallet\_transactions (type='DEBIT');
19. INSERT INTO wallet\_transactions (type='CREDIT');
20. COMMIT;

**Saga Pattern for Distributed Transactions**

**Problem**: Maintaining consistency across multiple services without 2PC

**Solution**: Choreography-based Saga

Payment Service → Wallet Service → Notification Service

↓ ↓ ↓

SUCCESS SUCCESS SUCCESS

↓ ↓ ↓

[COMMIT] [COMMIT] [COMMIT]

OR (if any fails)

↓ ↓ ↓

SUCCESS FAILURE X

↓ ↓

[COMPENSATE] ← [COMPENSATE]

**Implementation**:

// Saga coordinator pattern

async function processPayment(paymentData) {

const sagaId = generateUUID();

try {

// Step 1: Create payment

const payment = await paymentService.create(paymentData);

// Step 2: Fraud check

const fraudCheck = await fraudService.check(payment);

if (fraudCheck.decision === 'DECLINED') {

throw new FraudDetectedException();

}

// Step 3: Hold wallet funds

await walletService.holdFunds({

walletId: payment.senderWalletId,

amount: payment.amount,

referenceId: payment.paymentId

});

// Step 4: Capture payment

await paymentService.capture(payment.paymentId);

// Step 5: Transfer funds

await walletService.transferFunds({

fromWallet: payment.senderWalletId,

toWallet: payment.receiverWalletId,

amount: payment.amount

});

return { success: true, paymentId: payment.paymentId };

} catch (error) {

// Compensating transactions

await compensatePayment(sagaId, error);

throw error;

}

}

**🔐 Consistency Strategies**

**1. Strong Consistency (Within Service)**

**Use Cases**: Financial transactions, balance updates

**Approach**: ACID transactions within single database

-- Example: Wallet transfer with strong consistency

BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;

-- Check sender balance

SELECT balance FROM wallets

WHERE wallet\_id = ? FOR UPDATE;

-- Debit sender

UPDATE wallets

SET balance = balance - ?, version = version + 1

WHERE wallet\_id = ? AND version = ?;

-- Credit receiver

UPDATE wallets

SET balance = balance + ?, version = version + 1

WHERE wallet\_id = ? AND version = ?;

-- Record transaction

INSERT INTO wallet\_transactions (...);

COMMIT;

**2. Eventual Consistency (Across Services)**

**Use Cases**: Notifications, analytics, reporting

**Approach**: Event-driven architecture with message queues

// Publisher (Payment Service)

await publishEvent({

eventType: 'PAYMENT\_COMPLETED',

paymentId: payment.id,

amount: payment.amount,

merchantId: payment.merchantId,

timestamp: Date.now()

});

// Subscribers

// 1. Notification Service → Send email

// 2. Analytics Service → Update metrics

// 3. Settlement Service → Queue for settlement

**Event Store Pattern**:

CREATE TABLE event\_store (

event\_id UUID PRIMARY KEY,

aggregate\_id UUID NOT NULL,

aggregate\_type VARCHAR(50) NOT NULL,

event\_type VARCHAR(100) NOT NULL,

event\_data JSONB NOT NULL,

version INTEGER NOT NULL,

created\_at TIMESTAMP NOT NULL,

UNIQUE (aggregate\_id, version)

);

**3. Idempotency**

**Critical for**: Payment processing, refunds, transfers

**Implementation Strategies**:

**A. Idempotency Key**

// Client sends idempotency key

POST /api/v1/payments

Headers: {

"Idempotency-Key": "unique-client-generated-key"

}

// Server implementation

async function processPayment(data, idempotencyKey) {

// Check if already processed

const existing = await redis.get(`idempotency:${idempotencyKey}`);

if (existing) {

return JSON.parse(existing);

}

// Process payment

const result = await createPayment(data);

// Store result with TTL (24 hours)

await redis.setex(`idempotency:${idempotencyKey}`, 86400,

JSON.stringify(result));

return result;

}

**B. Database Unique Constraint**

CREATE TABLE payment\_idempotency (

idempotency\_key VARCHAR(255) PRIMARY KEY,

payment\_id UUID NOT NULL,

response\_data JSONB,

created\_at TIMESTAMP NOT NULL

);

-- Insert will fail if key exists

INSERT INTO payment\_idempotency VALUES (?, ?, ?, NOW())

ON CONFLICT (idempotency\_key)

DO UPDATE SET created\_at = payment\_idempotency.created\_at

RETURNING payment\_id, response\_data;

**4. Optimistic Locking**

**Purpose**: Prevent lost updates in concurrent scenarios

-- Version-based optimistic locking

UPDATE wallets

SET balance = ?,

available\_balance = ?,

version = version + 1,

updated\_at = NOW()

WHERE wallet\_id = ?

AND version = ?; -- Current version

-- Check affected rows

IF affected\_rows = 0 THEN

RAISE EXCEPTION 'Concurrent modification detected';

END IF;

**Application Code**:

async function updateWalletBalance(walletId, amount, currentVersion) {

let retries = 3;

while (retries > 0) {

try {

const result = await db.query(

`UPDATE wallets

SET balance = balance + $1, version = version + 1

WHERE wallet\_id = $2 AND version = $3`,

[amount, walletId, currentVersion]

);

if (result.rowCount === 0) {

// Reload and retry

const wallet = await getWallet(walletId);

currentVersion = wallet.version;

retries--;

continue;

}

return { success: true };

} catch (error) {

throw error;

}

}

throw new Error('Exceeded retry limit');

}

**5. Distributed Locks (Redis)**

**Use Cases**: Prevent duplicate processing, rate limiting

// Acquire distributed lock

async function acquireLock(resource, ttl = 30000) {

const lockKey = `lock:${resource}`;

const lockValue = generateUUID();

const acquired = await redis.set(

lockKey,

lockValue,

'NX', // Only set if not exists

'PX', // Expiry in milliseconds

ttl

);

return acquired ? lockValue : null;

}

// Release lock

async function releaseLock(resource, lockValue) {

const script = `

if redis.call("get", KEYS[1]) == ARGV[1] then

return redis.call("del", KEYS[1])

else

return 0

end

`;

return await redis.eval(script, 1, `lock:${resource}`, lockValue);

}

// Usage example

async function processTransfer(transferId) {

const lockValue = await acquireLock(`transfer:${transferId}`);

if (!lockValue) {

throw new Error('Transfer already being processed');

}

try {

// Process transfer

await performTransfer(transferId);

} finally {

await releaseLock(`transfer:${transferId}`, lockValue);

}

}

**🚀 Scaling & Performance**

**1. Database Sharding**

**Horizontal Partitioning Strategy**:

Shard Key Selection:

- User Service: user\_id (hash-based)

- Wallet Service: wallet\_id (hash-based)

- Payment Service: merchant\_id (range or hash)

**Sharding Implementation**:

function getShardId(userId) {

const hash = murmurhash(userId);

return hash % TOTAL\_SHARDS;

}

async function getUserData(userId) {

const shardId = getShardId(userId);

const connection = connectionPool[shardId];

return await connection.query('SELECT \* FROM users WHERE user\_id = ?', [userId]);

}

**2. Read Replicas**

┌──────────┐

│ Master │ (Writes only)

│ Database │

└────┬─────┘

│ Replication

├──────────┬──────────┬──────────┐

▼ ▼ ▼ ▼

┌─────────┐ ┌─────────┐ ┌─────────┐ ┌─────────┐

│ Replica │ │ Replica │ │ Replica │ │ Replica │

│ 1 │ │ 2 │ │ 3 │ │ 4 │

└─────────┘ └─────────┘ └─────────┘ └─────────┘

(Read queries distributed via load balancer)

**Connection Routing**:

class DatabaseManager {

async write(query, params) {

return await this.masterConnection.query(query, params);

}

async read(query, params) {

const replica = this.getRandomReplica();

return await replica.query(query, params);

}

getRandomReplica() {

return this.replicas[Math.floor(Math.random() \* this.replicas.length)];

}

}

**3. Caching Strategy**

**Multi-Level Caching**:

Level 1: Application Memory (LRU Cache)

↓

Level 2: Redis (Distributed Cache)

↓

Level 3: Database

**Cache Implementation**:

class CacheService {

async get(key) {

// L1: Check memory cache

let value = memoryCache.get(key);

if (value) return value;

// L2: Check Redis

value = await redis.get(key);

if (value) {

memoryCache.set(key, value);

return value;

}

// L3: Load from database

value = await database.query('SELECT ...');

// Populate caches

await redis.setex(key, 300, value);

memoryCache.set(key, value);

return value;

}

}

**Cache Warming**:

// Pre-populate cache for frequently accessed data

async function warmCache() {

const popularMerchants = await getMostActiveMerchants(100);

for (const merchant of popularMerchants) {

const data = await loadMerchantData(merchant.id);

await redis.setex(`merchant:${merchant.id}`, 3600,

JSON.stringify(data));

}

}

**4. Query Optimization**

**Index Strategy**:

-- Composite indexes for common queries

CREATE INDEX idx\_payments\_merchant\_status\_date

ON payments(merchant\_id, status, created\_at DESC);

-- Partial indexes for specific conditions

CREATE INDEX idx\_active\_payments

ON payments(created\_at)

WHERE status IN ('PENDING', 'AUTHORIZED');

-- Covering indexes to avoid table lookups

CREATE INDEX idx\_payments\_summary

ON payments(merchant\_id, amount, status, created\_at)

WHERE status = 'CAPTURED';

**Query Patterns**:

-- BAD: N+1 query problem

SELECT \* FROM payments WHERE merchant\_id = ?;

-- Then for each payment:

SELECT \* FROM payment\_transactions WHERE payment\_id = ?;

-- GOOD: Join or batch query

SELECT p.\*, pt.\*

FROM payments p

LEFT JOIN payment\_transactions pt ON p.payment\_id = pt.payment\_id

WHERE p.merchant\_id = ?;

**5. Connection Pooling**

const pool = new Pool({

host: 'localhost',

database: 'interpay\_payments',

max: 20, // Maximum pool size

min: 5, // Minimum pool size

idleTimeoutMillis: 30000,

connectionTimeoutMillis: 2000,

});

// Automatic connection management

async function queryDatabase(sql, params) {

const client = await pool.connect();

try {

const result = await client.query(sql, params);

return result.rows;

} finally {

client.release(); // Return to pool

}

}

**6. Cassandra Performance Tuning**

**Partition Size Management**:

-- Use time bucketing to prevent large partitions

CREATE TABLE payment\_logs (

merchant\_id UUID,

bucket TEXT, -- Format: "YYYY-MM-DD-HH"

log\_timestamp TIMESTAMP,

log\_data TEXT,

PRIMARY KEY ((merchant\_id, bucket), log\_timestamp)

) WITH CLUSTERING ORDER BY (log\_timestamp DESC);

-- Query with bucket

SELECT \* FROM payment\_logs

WHERE merchant\_id = ?

AND bucket = '2025-09-30-14'

AND log\_timestamp > ?;

**Batch Writes**:

BEGIN BATCH

INSERT INTO payment\_logs VALUES (?, ?, ?, ?);

INSERT INTO payment\_logs VALUES (?, ?, ?, ?);

INSERT INTO payment\_logs VALUES (?, ?, ?, ?);

APPLY BATCH;

**🔒 Security Measures**

**1. Data Encryption**

**At Rest**:

-- PostgreSQL: Transparent Data Encryption (TDE)

-- Or application-level encryption for sensitive fields

CREATE TABLE cards (

card\_id UUID PRIMARY KEY,

card\_token VARCHAR(255), -- Encrypted token

encrypted\_data BYTEA -- pgcrypto extension

);

-- Encrypt on insert

INSERT INTO cards (encrypted\_data)

VALUES (pgp\_sym\_encrypt('sensitive-data', 'encryption-key'));

-- Decrypt on select

SELECT pgp\_sym\_decrypt(encrypted\_data, 'encryption-key') FROM cards;

**In Transit**:

* TLS 1.3 for all API communications
* mTLS between microservices
* Certificate pinning for mobile apps

**2. PCI DSS Compliance**

**Card Data Handling**:

┌──────────┐ ┌─────────────┐ ┌──────────────┐

│ Client │─────→│ Vault │─────→│ Card │

│ │ PAN │ Service │Token │ Networks │

└──────────┘ └─────────────┘ └──────────────┘

│

▼

(Store token only)

**Never Store**:

* Full PAN (Primary Account Number)
* CVV/CVC codes
* PIN numbers

**Tokenization**:

async function tokenizeCard(cardData) {

// Send to PCI-compliant vault

const token = await vaultService.tokenize({

cardNumber: cardData.number,

expiryMonth: cardData.expiryMonth,

expiryYear: cardData.expiryYear

});

// Store only token and metadata

await db.query(`

INSERT INTO cards (card\_token, card\_brand, last\_four)

VALUES (?, ?, ?)

`, [token, cardData.brand, cardData.number.slice(-4)]);

return token;

}

**3. Authentication & Authorization**

**JWT Token Structure**:

{

"sub": "user-uuid",

"merchant\_id": "merchant-uuid",

"roles": ["user", "merchant"],

"permissions": ["payment.create", "wallet.read"],

"iat": 1696089600,

"exp": 1696093200

}

**API Key Security**:

async function validateApiKey(publicKey, secretKey) {

// Rate limit check

const rateKey = `ratelimit:${publicKey}:${getCurrentMinute()}`;

const count = await redis.incr(rateKey);

await redis.expire(rateKey, 60);

if (count > 1000) {

throw new RateLimitError('API rate limit exceeded');

}

// Validate key

const hashedSecret = await bcrypt.hash(secretKey, storedSalt);

const storedKey = await redis.get(`apikey:${publicKey}`);

if (!storedKey) {

// Cache miss - load from database

const key = await db.query(

'SELECT secret\_key\_hash FROM merchant\_api\_keys WHERE public\_key = ?',

[publicKey]

);

await redis.setex(`apikey:${publicKey}`, 3600, key.secret\_key\_hash);

storedKey = key.secret\_key\_hash;

}

return storedKey === hashedSecret;

}

**4. Fraud Prevention**

**Velocity Checks**:

async function checkVelocity(userId, amount) {

const hourKey = `velocity:${userId}:hour`;

const dayKey = `velocity:${userId}:day`;

// Increment counters

const hourCount = await redis.incr(hourKey);

const dayCount = await redis.incr(dayKey);

// Set expiry on first increment

if (hourCount === 1) await redis.expire(hourKey, 3600);

if (dayCount === 1) await redis.expire(dayKey, 86400);

// Check limits

if (hourCount > 10 || dayCount > 50) {

return { blocked: true, reason: 'Velocity limit exceeded' };

}

return { blocked: false };

}

**Device Fingerprinting**:

function generateDeviceFingerprint(request) {

const components = [

request.headers['user-agent'],

request.headers['accept-language'],

request.ip,

request.headers['accept-encoding']

];

return crypto

.createHash('sha256')

.update(components.join('|'))

.digest('hex');

}

**5. Audit Logging**

**Comprehensive Logging**:

async function auditLog(action) {

const log = {

audit\_id: generateUUID(),

user\_id: action.userId,

service\_name: 'payment-service',

action: action.type,

entity\_type: 'payment',

entity\_id: action.paymentId,

old\_values: action.before,

new\_values: action.after,

ip\_address: action.ipAddress,

user\_agent: action.userAgent,

status: 'SUCCESS',

created\_at: new Date()

};

// Write to PostgreSQL (compliance)

await db.query('INSERT INTO audit\_logs VALUES (?)', [log]);

// Write to Cassandra (analytics)

await cassandra.execute(

'INSERT INTO interpay\_logs.audit\_events VALUES (?)',

[log]

);

}

**📊 Monitoring & Observability**

**Key Metrics to Track**

**System Health**:

* Database connection pool utilization
* Query response times (p50, p95, p99)
* Cache hit/miss ratios
* Queue depth and processing lag

**Business Metrics**:

* Transaction volume and value
* Success/failure rates
* Average transaction time
* Fraud detection accuracy

**Alerting Thresholds**:

alerts:

- name: high\_transaction\_failure\_rate

condition: failure\_rate > 5%

duration: 5m

action: page\_oncall

- name: wallet\_balance\_mismatch

condition: sum(wallet\_transactions) != wallet.balance

action: critical\_alert

- name: fraud\_score\_spike

condition: avg(fraud\_score) > 0.8

duration: 10m

action: notify\_security\_team

**🔄 Disaster Recovery**

**Backup Strategy**

**PostgreSQL**:

# Continuous WAL archiving

archive\_command = 'cp %p /backup/wal/%f'

# Daily full backup

pg\_basebackup -D /backup/full/$(date +%Y%m%d)

# Point-in-time recovery capability

restore\_command = 'cp /backup/wal/%f %p'

**Cassandra**:

# Snapshot-based backups

nodetool snapshot interpay\_logs

# Incremental backups

nodetool backup

**Recovery Time Objective (RTO)**: 15 minutes  
**Recovery Point Objective (RPO)**: 5 minutes

**📈 Capacity Planning**

**Growth Projections**

Year 1: 100K transactions/day

Year 2: 1M transactions/day

Year 3: 10M transactions/day

**Scaling Milestones**:

* **0-100K TPS**: Single database with replicas
* **100K-1M TPS**: Implement sharding
* **1M+ TPS**: Multi-region deployment

**🎯 Best Practices Summary**

1. **Always use database transactions for financial operations**
2. **Implement idempotency for all payment endpoints**
3. **Never store sensitive card data - use tokenization**
4. **Employ optimistic locking for concurrent updates**
5. **Use Saga pattern for distributed transactions**
6. **Cache aggressively but invalidate carefully**
7. **Monitor everything - metrics, logs, traces**
8. **Test failure scenarios regularly (chaos engineering)**
9. **Implement circuit breakers for external services**
10. **Regular security audits and penetration testing**

**Document Version**: 1.0  
**Last Updated**: September 30, 2025  
**Next Review**: December 2025