SDE: System Design and Engineering

Lecture – 5

Introduction to

Payment and Fintech

From Zero to Google: Architecting the Invisible Infrastructure

by

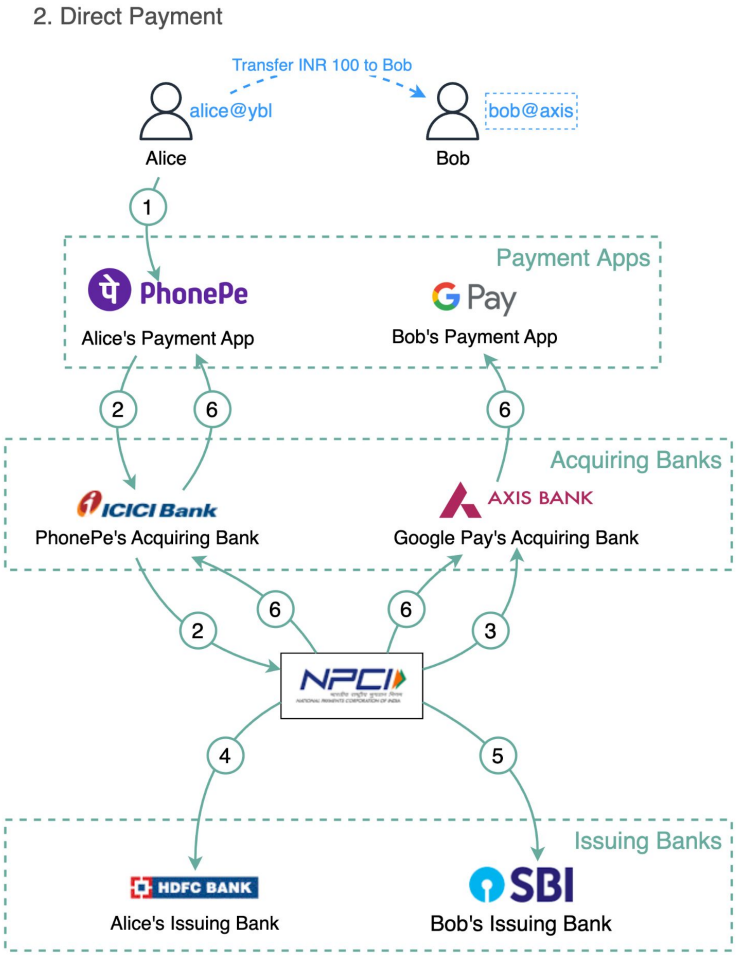
Aatiz Ghimire

Sections

- Introduction to Payments & Fintech Ecosystem(Previous Class)
- Payment Systems & Protocols (Previous Class)
- Digital Wallets & Mobile Payments (Previous Class)
- Payment Security & Reconciliation
- System Design for Scalable Payment Platforms

Unified Payments Interface (UPI)

- **UPI** = Instant, real-time payment system by **National Payments Corporation of India (NPCI)**.
- Handles **60%+ of India's digital retail transactions**.
- Acts as both a **payment markup language** and a **standard for interoperable payments**.
- Works across banks and payment apps.



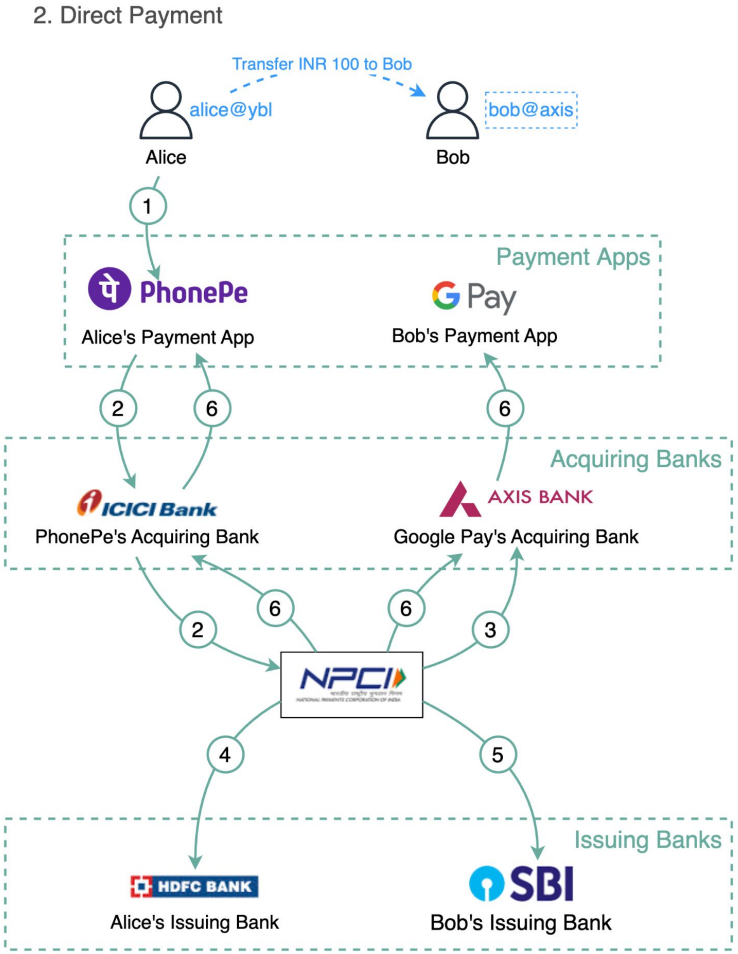
Registration Process

1. **User Onboarding**

- Bob provides mobile number (**+91 12345678**).
- Phone verification via **OTP**.
- Sets up **VPA (Virtual Payment Address)** – e.g., bob@axis.

2. **VPA Creation**

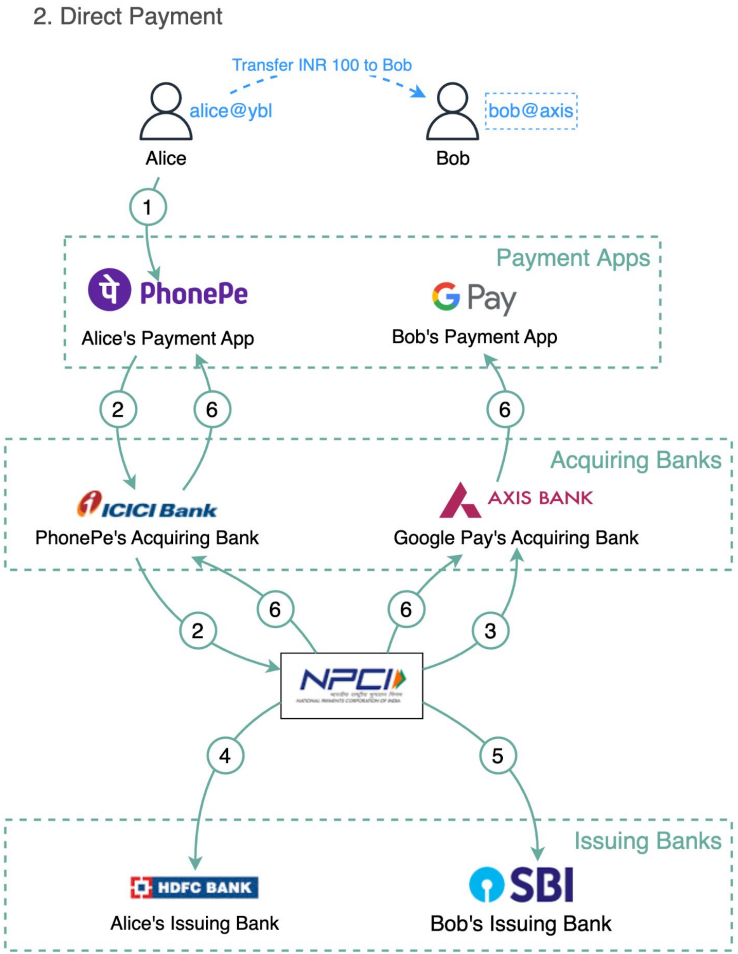
- Payment app sends request to **acquiring bank** to create VPA.
- Acquiring bank confirms and returns VPA.
- Payment app shows VPA to Bob.



Linking Bank Account

Linking Bank Account

- Bob links **SBI account** to bob@axis.
- Request sent to **NPCI** (the UPI switch).
- NPCI resolves VPA → finds issuing bank (SBI).
- Bob authenticates with account details.
- Sets **UPI PIN** (used for 2FA).
- PIN securely stored by issuing bank.



Direct Payment Flow

1. **Payment Initiation**

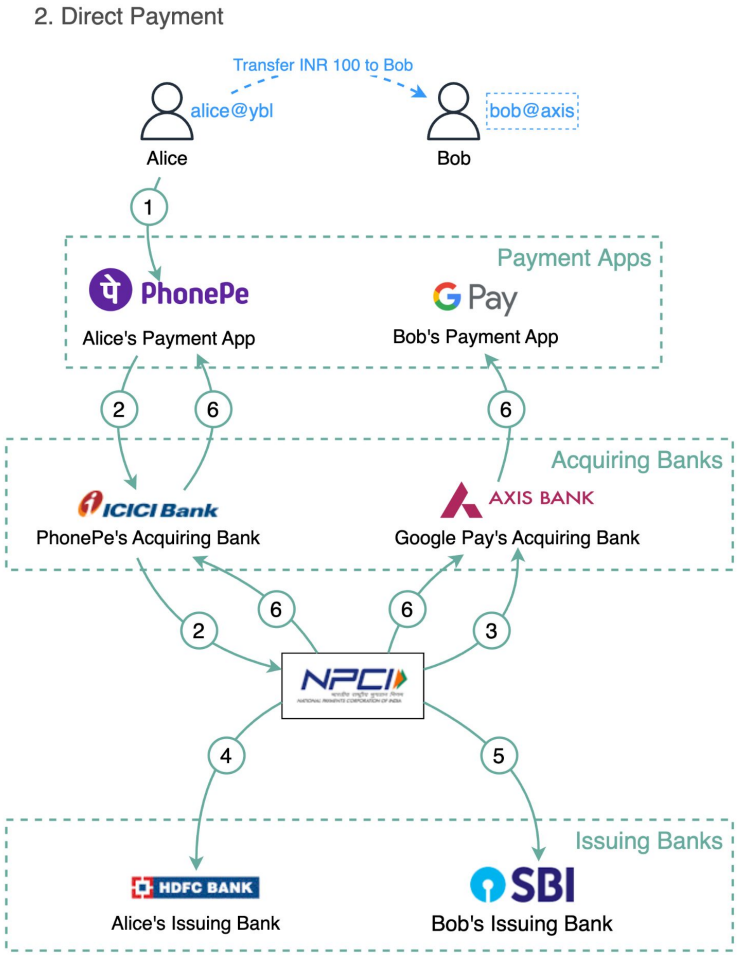
- Alice enters Bob's UPI ID (bob@axis) and amount (INR 100).
- PhonePe (Alice's app) sends request via acquiring bank (ICICI) to NPCI.

2. **Routing & Authorization**

- NPCI resolves VPA → Axis Bank (VPA host) → SBI Bank (Bob's account).
- NPCI debits Alice's **HDFC account** by INR 100.
- NPCI credits Bob's **SBI account** by INR 100.

3. **Confirmation**

- NPCI sends success notification to both payment apps via acquiring banks.



Why UPI is a Model for Nepal

- **Real-time** interbank settlement.
- **Interoperability** across banks & apps.
- **Simple addressing** (VPA instead of long account + IFSC).
- **High adoption** with minimal transaction cost.

How Scan to Pay Works

Scan-to-Pay = Paying by scanning a QR code from a merchant using a **digital wallet** (PayPal, Paytm, Venmo, eSewa, Khalti, etc.).

Two Sub-Processes:

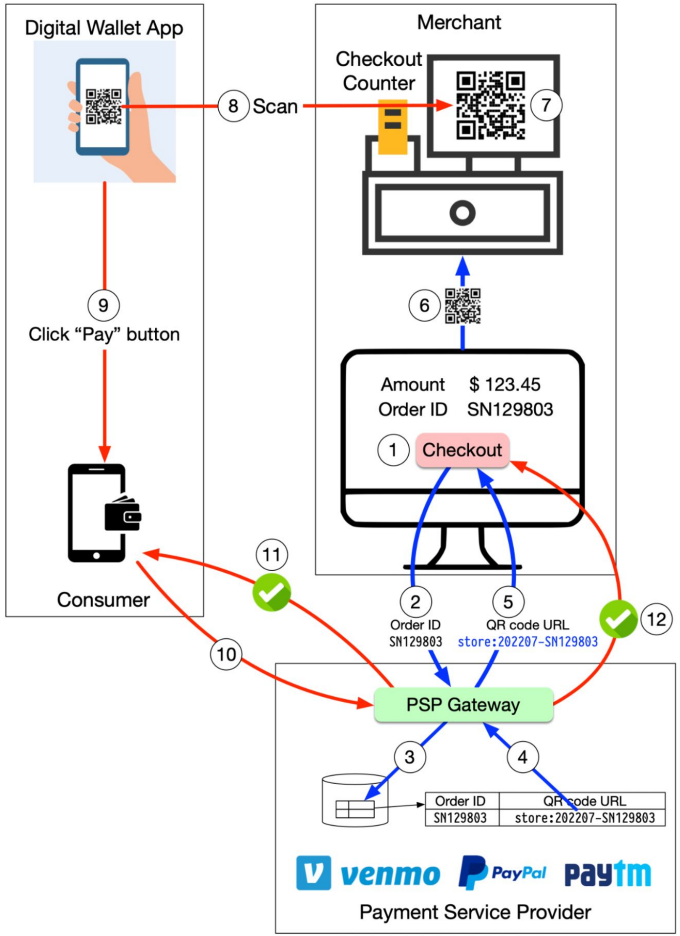
1. **Merchant Generates QR Code** (payment request)

2. **Consumer Scans QR Code** (initiates payment)

Step 1: Merchant Generates QR Code

1. **Order Creation** – Cashier tallies goods (e.g., $123.45), assigns order ID (e.g., SN129803).
2. **Send to PSP** – Merchant system sends **order ID** + **amount** to Payment Service Provider (PSP).
3. **PSP Saves & Generates QR** – Stores transaction in database, creates a **QR code URL**.
4. **Gateway Returns QR URL** – Payment gateway sends QR code URL back to merchant system.
5. **Display at Checkout** – Merchant POS or display screen shows the QR code for scanning.

⏱️ All done in **< 1 second**.



Step 2: Consumer Scans & Pays

1. Open Wallet App – Consumer launches their preferred digital wallet app.

2. **Scan QR Code** – App decodes payment details (order ID, amount, merchant ID).

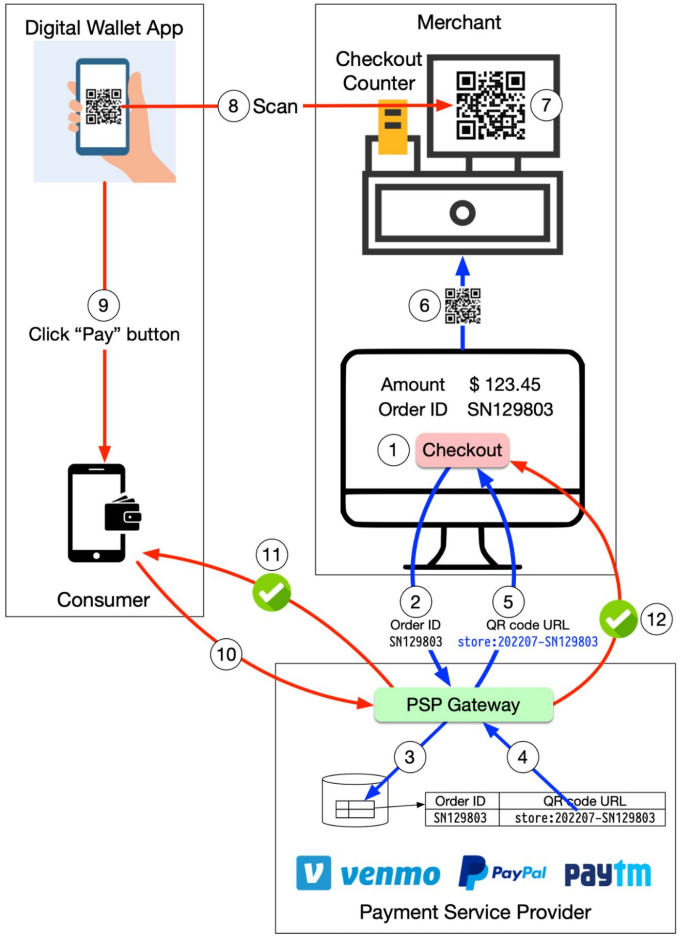
3. **Confirm & Pay** – Consumer verifies amount, taps "Pay."

4. **Wallet → PSP** – Wallet notifies PSP the payment for that QR code is complete.

5. **PSP Updates Records** – Marks QR code as paid in database.

6. **Success Notifications** – PSP sends:

- Success message to **consumer's app**.
- Payment confirmation to **merchant's POS system**.



Nepal Context

Merchant-Presented Static QR: Common in small shops (printed QR).

Merchant-Presented Dynamic QR: Used in supermarkets with POS integration (e.g., Bhatbhateni).

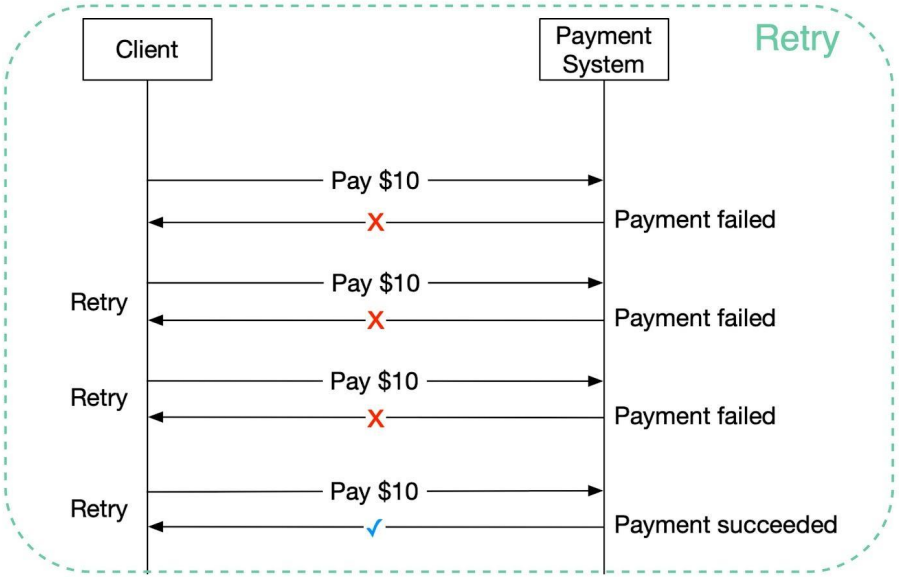
PSPs: eSewa, Khalti, IME Pay, Fonepay Network handle request storage, QR generation, and settlement.

How to Avoid Double Payment

- **Impact:** Overcharging customers damages trust and creates costly refunds.
- **Goal:** Ensure **exactly-once** execution of payment orders.
- **Definition:**
  - **At least once** → Payment eventually succeeds despite failures.
  - **At most once** → Payment never executes more than once.
  - Together = **Exactly once**.

Part 1: Achieving At-Least-Once with Retry

- **When?** Network errors, timeouts, server crashes.
- **How?**
  - Client attempts payment.
  - If failure detected, **retry** request.
  - Continue until success or max retry limit.
- **Example:**
  - Payment request for **$10** fails 3 times.
  - Succeeds on **4th attempt**.



Part 2: Achieving At-Most-Once with Idempotency

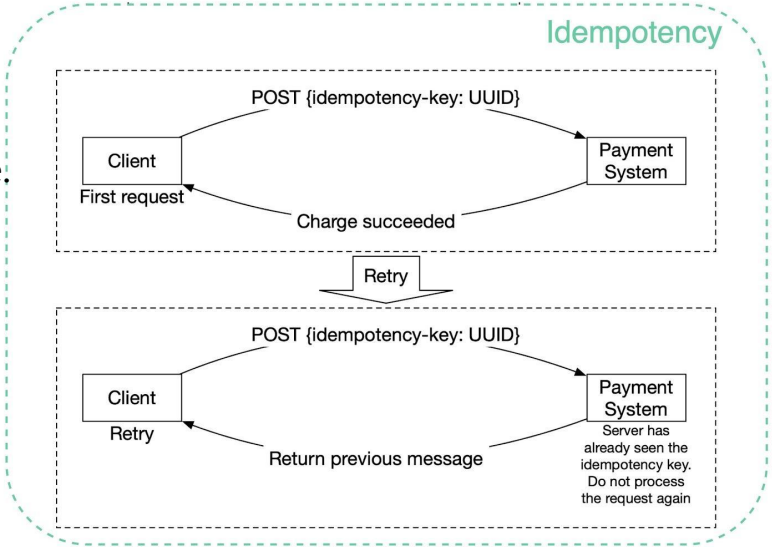
- **Definition:** Same request made multiple times → **same outcome**.

- **Implementation:**

  1. Generate **idempotency key** (UUID) on client side.
  2. Send with every payment request in **HTTP header**:

     Idempotency-Key: 123e4567-e89b-12d3-a456-426614174000
  3. Server stores key + result → reuses result for identical requests.

- **Industry Practice:** Stripe, PayPal recommend UUID keys.



Combined Flow for Exactly-Once

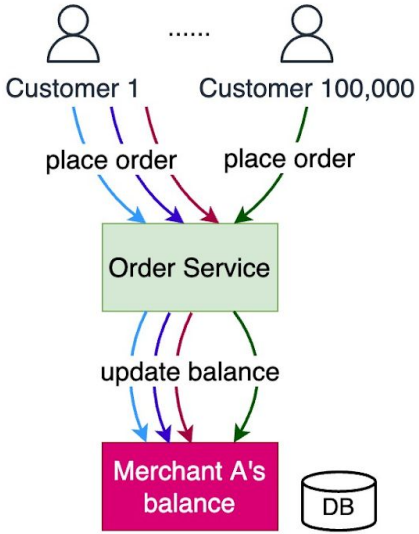
1. **Client sends payment** with idempotency key.
2. **Server checks key**:
    - If new → Process payment, store result.
    - If duplicate → Return stored result.
3. If **network error**, client retries **with same key**.
4. Server ensures **no double charge**.

Hotspot Accounts

A **hotspot account** is one that experiences a **large number of concurrent operations**, creating a bottleneck in the payment system.

Example:

- A big brand (Daraz, Bhatbhateni) launches a flash sale.
- Thousands of concurrent purchase requests hit the same merchant account.
- Continuous **row locking** on the account's balance slows the system.



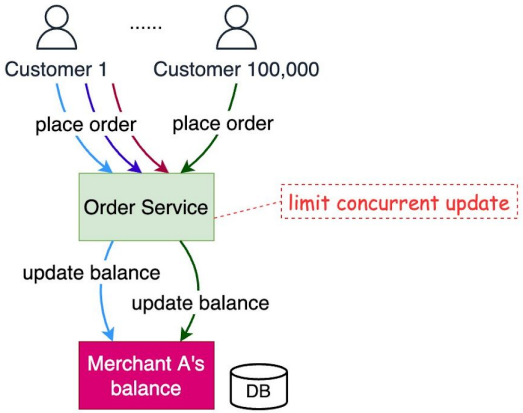
Problems with Hotspot Accounts

- **Database Lock Contention** – Multiple updates block each other.
- **Throughput Degradation** – Slower response times for all users.
- **Potential Downtime** – Overloaded systems risk failure during peak load.

Optimization Strategies

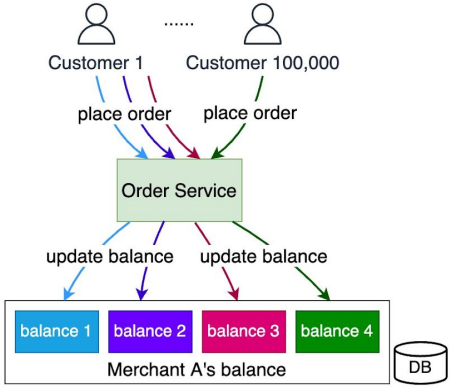
1. Rate Limiting

- Restrict the number of requests per time window.
- Pros: Quick to implement, improves responsiveness.
- Cons: May reject valid transactions, hurting UX.



2. Split into Sub-Accounts

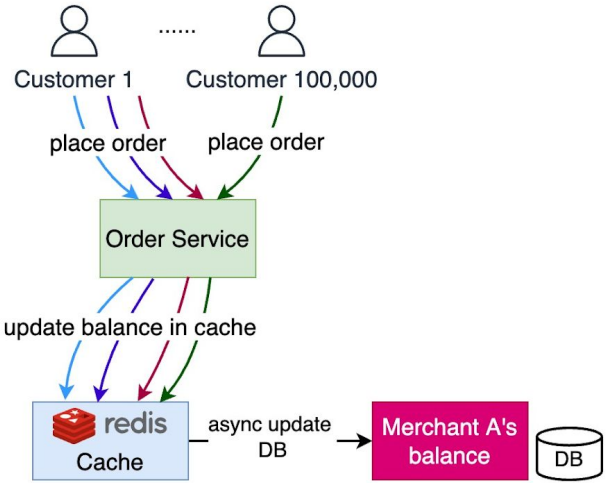
- Divide the main merchant balance into **multiple sub-accounts**.
- Each sub-account locks independently → reduces contention.



Optimization Strategies

3. Cache-First Updates

- Use **in-memory caching** (e.g., Redis) to update balances instantly.
- Asynchronously sync detailed statements to the database later.
- Boosts throughput significantly.



Recommended Hybrid Approach

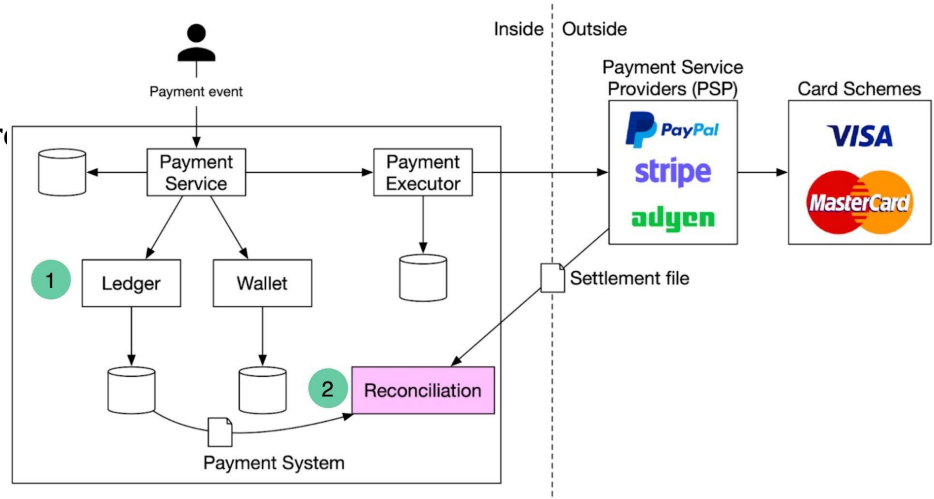
- **Rate-limit** extreme spikes.
- **Cache-first** for real-time responsiveness.
- **Sub-accounts** for load distribution.
- Monitor and dynamically adjust strategies during events (e.g., Daraz 8.8 Day).

Payment Reconciliation Overview

The process of **matching transactions** across systems (merchant, payment processor, ledger, bank) to ensur all amounts align.

Example:

- You buy a $200 watch via PayPal.
- **eCommerce System** – Records order for $200.
- **PayPal** – Records $200 transaction.
- **Ledger** – Records **debit** from buyer & **credit** to seller.



Double-entry Bookkeeping in Ledger

Account	Debit	Credit
buyer	$200	
seller		$200

Common Challenges

1. Data Normalization

- Different systems store timestamps, currency, and formats differently.
- **Fix:** Add a normalization layer to unify formats before comparison.

2. Massive Data Volume

- Millions of transactions to match daily.
- **Fix:**
  - Real-time → Streaming tools (e.g., Apache Flink).
  - Batch → Big data tools (e.g., Hadoop).

3. Cut-off Time Mismatches

- A transaction near midnight may fall on different dates in different systems.
- **Fix:** Mark as "temporary break" → Retry matching next day.

Why Reconciliation Matters & Best Practices

- Prevents **financial discrepancies**.
- Ensures **compliance** with auditing requirements.
- Protects against **fraud and system errors**.
- Acts as a **safety net** for peace of mind.

Best Practices

- Automate reconciliation workflows.
- Use **exactly-once semantics** where possible, but never skip reconciliation.
- Keep detailed audit logs.
- Integrate **alerts** for unmatched transactions.

Principles of High Availability & Scalability in Payment Systems

- Payments are mission-critical — downtime = lost revenue + trust.
- **Source:** Shopify's payment infrastructure best practices.
- **Applicability:** Global + local PSPs (eSewa, Khalti, IME Pay, Fonepay).

Principles of High Availability & Scalability in Payment Systems

Principle 1: Lower Timeouts, Fail Early

- Default timeouts are too high (60s).
- **Best practice:**
  - **Read timeout:** 5s
  - **Write timeout:** 1s
- **Why:** Avoids long waits & frees system resources faster.

Principle 2: Install Circuit Breakers

- Prevents cascading failures.
- Example: Shopify's *Semian* protects HTTP, DB, cache calls.
- **In Nepal:** PSPs can wrap Fonepay API calls with circuit breakers.

Principles of High Availability & Scalability in Payment Systems

Principle 3: Capacity Management

- Formula:
  Throughput = concurrent_requests / avg_processing_time
- Example: 50 requests, 100 ms each → **500 RPS**.
- Use load balancers + horizontal scaling.

Principle 4: Monitoring & Alerting

- Monitor **4 golden signals**:
  - Latency
  - Traffic
  - Errors
  - Saturation

- **Local tie-in:** downtime alerts via Prometheus + Grafana.

Principles of High Availability & Scalability in Payment Systems

Principle 5: Structured Logging

- Centralized, searchable logs.
- JSON log format for easy parsing.
- Store in ELK stack or Loki.

Principle 6: Use Idempotency Keys

- Prevents duplicate charges.
- Use **ULID** instead of random UUID for sortable IDs.
- Store transaction states.

Principles of High Availability & Scalability in Payment Systems

Principle 7: Consistent Reconciliation

- Match internal records with bank/PSP daily.
- Log discrepancies in a database table.
- **In Nepal:** Banks + Fonepay reconciliation cycles = daily/weekly.

Principle 8: Load Testing

- Simulate flash sale or festival traffic.
- Tools: JMeter, k6.
- In Nepal: Dashain/Tihar & mobile top-up traffic surges.

Principles of High Availability & Scalability in Payment Systems

Principle 9: Incident Management

- Define roles:
  - IMOC (Incident Manager on Call)
  - SRM (Support Response Manager)
  - Service Owners
- Use incident runbooks.

Principle 10: Incident Retrospectives

- 3 questions:
  1. What happened?
  2. What incorrect assumptions existed?
  3. How to prevent recurrence?
- Make fixes part of sprints.

Thank you!
Any Questions?