

# Chapter 1: Introduction to System Design

## 1. What is System Design and Why It Matters

### Definition

**System Design** is the process of defining the architecture, components, modules, interfaces, and data flow for a system to satisfy specified requirements. It's about designing large-scale distributed systems that can handle millions of users, process massive amounts of data, and remain reliable, scalable, and maintainable.

Think of it like designing a city rather than a single building:

- **Building a house** = Writing code for a feature
- **Designing a city** = System design (roads, utilities, zones, infrastructure)

### Why System Design Matters

#### 1. Scale and Growth

```
Day 1: 100 users → Simple app on one server works fine
Month 6: 10,000 users → Server starts slowing down
Year 1: 1,000,000 users → System crashes, data loss, angry users
```

Without proper system design, your application cannot handle growth.

#### 2. Real Business Impact

##### Example: E-commerce Site During Sale

- Poor design: Site crashes → Lost revenue (\$100,000s per minute)
- Good design: Site handles 100x traffic → Successful sale

##### Example: Social Media Platform

- Poor design: Feed takes 10 seconds to load → Users leave
- Good design: Feed loads in 200ms → Users stay engaged

#### 3. Cost Optimization

Bad system design wastes money:

#### Poorly Designed System:

- 100 servers needed
- Database reads are inefficient
- Monthly cost: \$50,000

#### Well-Designed System:

- 20 servers with caching
- Optimized database queries
- Monthly cost: \$10,000

## 4. User Experience

Aspect	Poor Design	Good Design
Page Load	5-10 seconds	< 1 second
Downtime	Weekly crashes	99.99% uptime
Features	Basic only	Rich features work smoothly

## Key Goals of System Design

- 1. Scalability** - Handle growth (more users, more data)
- 2. Reliability** - System works consistently without failures
- 3. Availability** - System is accessible when users need it
- 4. Maintainability** - Easy to update and fix
- 5. Performance** - Fast response times
- 6. Cost-effectiveness** - Efficient use of resources

---

## 2. Difference Between Software Design and System Design

### Software Design (Low-Level Design)

**Focus:** How to write the code

**Scope:** Individual components, classes, functions

### Questions Asked:

- What classes do I need?
- Which design pattern should I use?
- How do I structure my code?
- What are the function signatures?

## Example: Building a User Authentication Feature

python

# Software Design - Writing the actual code

```
class User:  
    def __init__(self, username, email, password):  
        self.username = username  
        self.email = email  
        self.password_hash = self._hash_password(password)  
        self.created_at = datetime.now()
```

```
    def _hash_password(self, password):  
        """Hash password using bcrypt"""  
        return bcrypt.hashpw(password.encode('utf-8'), bcrypt.gensalt())
```

```
    def verify_password(self, password):  
        """Verify password against hash"""  
        return bcrypt.checkpw(  
            password.encode('utf-8'),  
            self.password_hash  
)
```

class AuthenticationService:

```
    def __init__(self, database):  
        self.db = database  
        self.session_manager = SessionManager()
```

```
    def login(self, username, password):  
        """Authenticate user and create session"""  
        user = self.db.get_user(username)
```

```
        if not user:  
            raise InvalidCredentialsError()
```

```
        if not user.verify_password(password):  
            raise InvalidCredentialsError()
```

```
        session = self.session_manager.create_session(user)  
        return session.token
```

```
    def logout(self, session_token):  
        """Invalidate user session"""  
        self.session_manager.invalidate_session(session_token)
```

### Concerns:

- Code structure and organization
  - Design patterns (Factory, Singleton, Strategy)
  - Object-oriented principles (SOLID)
  - Code quality and testing
- 

## **System Design (High-Level Design)**

**Focus:** How to architect the entire system

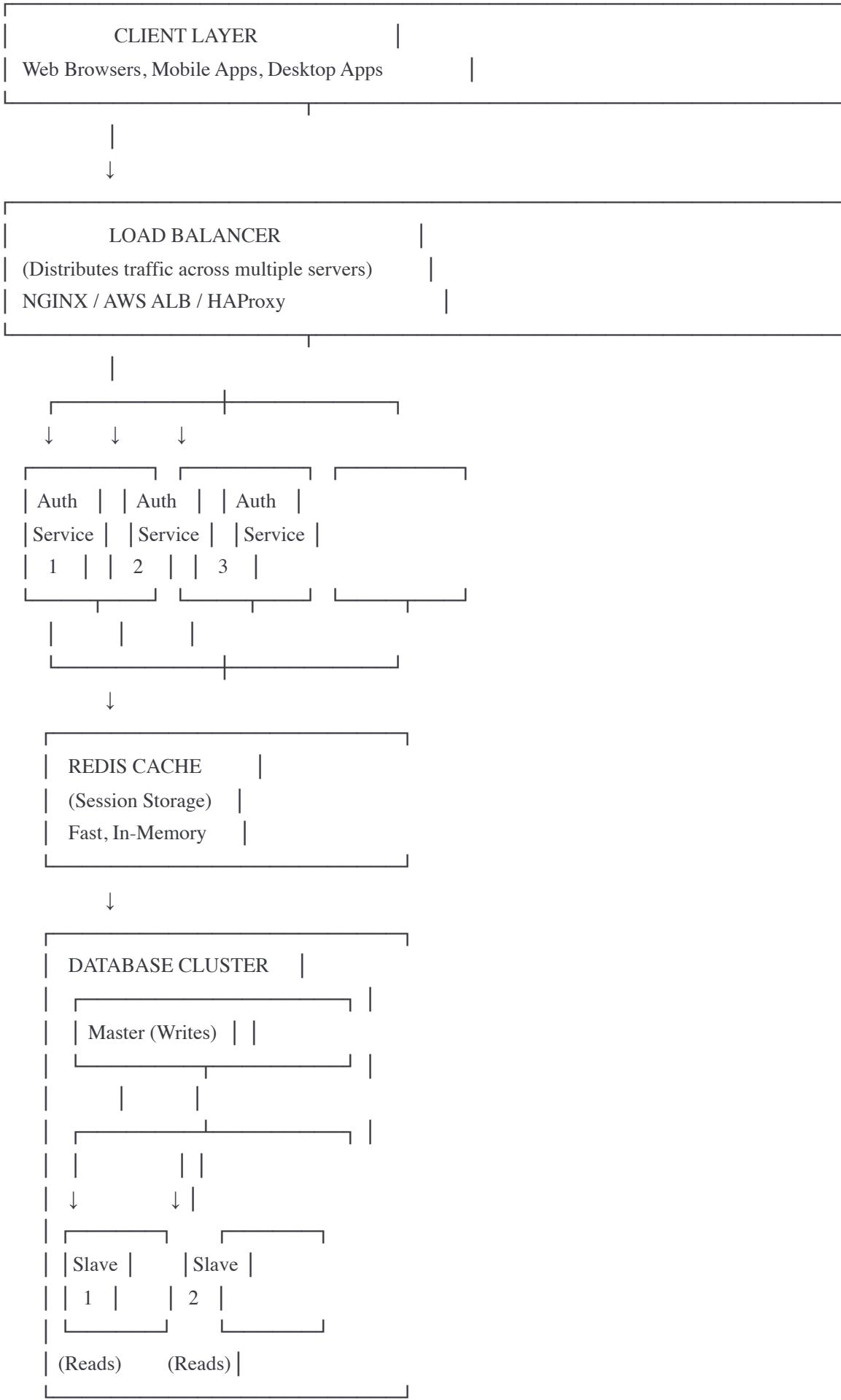
**Scope:** Multiple servers, databases, services, infrastructure

### **Questions Asked:**

- How many servers do we need?
- Where do we store data?
- How do we handle 1 million concurrent users?
- What happens if a server crashes?
- How do we ensure data consistency?

### **Example: Building an Authentication System at Scale**

## System Design for Authentication at Scale:



## Concerns:

- Handling 10 million users
  - Database replication strategy
  - Caching layer for performance
  - Load balancing strategy
  - Failover mechanisms
  - Data consistency
  - Security at scale
- 

## Side-by-Side Comparison

Aspect	Software Design	System Design
Scope	Single application/component	Entire system infrastructure
Scale	Hundreds/thousands of users	Millions/billions of users
Focus	Code structure, classes	Servers, databases, networks
Tools	IDEs, design patterns	Architecture diagrams, cloud services
Questions	"How do I code this?"	"How do we scale this?"
Example	User class with methods	User service across 100 servers
Time to Build	Days to weeks	Months to years
Team Size	1-5 developers	10-1000 engineers

## Analogy

**Software Design** = Designing the interior of a single apartment

- Room layout
- Furniture placement
- Electrical outlets
- Plumbing fixtures

**System Design** = Designing an entire apartment complex

- Building structure
- Water supply system for 500 units
- Electrical grid
- Sewage system
- Fire safety

- Elevator system
  - Parking structure
- 

### 3. System Design Interview Expectations

#### What Companies Are Looking For

System design interviews assess your ability to:

1. Design large-scale systems
2. Make architecture decisions
3. Understand trade-offs
4. Communicate technical ideas
5. Handle ambiguity

#### Interview Format (45-60 minutes)

##### Phase 1: Requirements Clarification (5-10 min)

**Interviewer:** "Design Instagram"

**Bad Response:** *Immediately starts drawing boxes*

**Good Response:**

You: "Let me clarify the requirements first:

Functional Requirements:

- Should users be able to upload photos and videos?
- Do we need a feed feature?
- Should we support comments and likes?
- Are we building stories feature?
- Do we need direct messaging?

Non-Functional Requirements:

- How many users are we expecting? (Scale)
- What's the expected read:write ratio?
- Are we targeting global audience?
- What's our latency requirement?
- Do we need high availability?"

##### Phase 2: Back-of-the-Envelope Calculations (5-10 min)

## Example: Instagram-like System

Assumptions:

- 500 million daily active users (DAU)
- Each user uploads 2 photos per day (on average)
- Each user views 50 photos per day
- Average photo size: 2 MB

Storage Calculations:

- Daily uploads:  $500M \text{ users} \times 2 \text{ photos} = 1 \text{ billion photos/day}$
- Daily storage:  $1B \text{ photos} \times 2 \text{ MB} = 2,000 \text{ TB} = 2 \text{ PB per day}$
- Yearly storage:  $2 \text{ PB} \times 365 = 730 \text{ PB/year}$

Bandwidth Calculations:

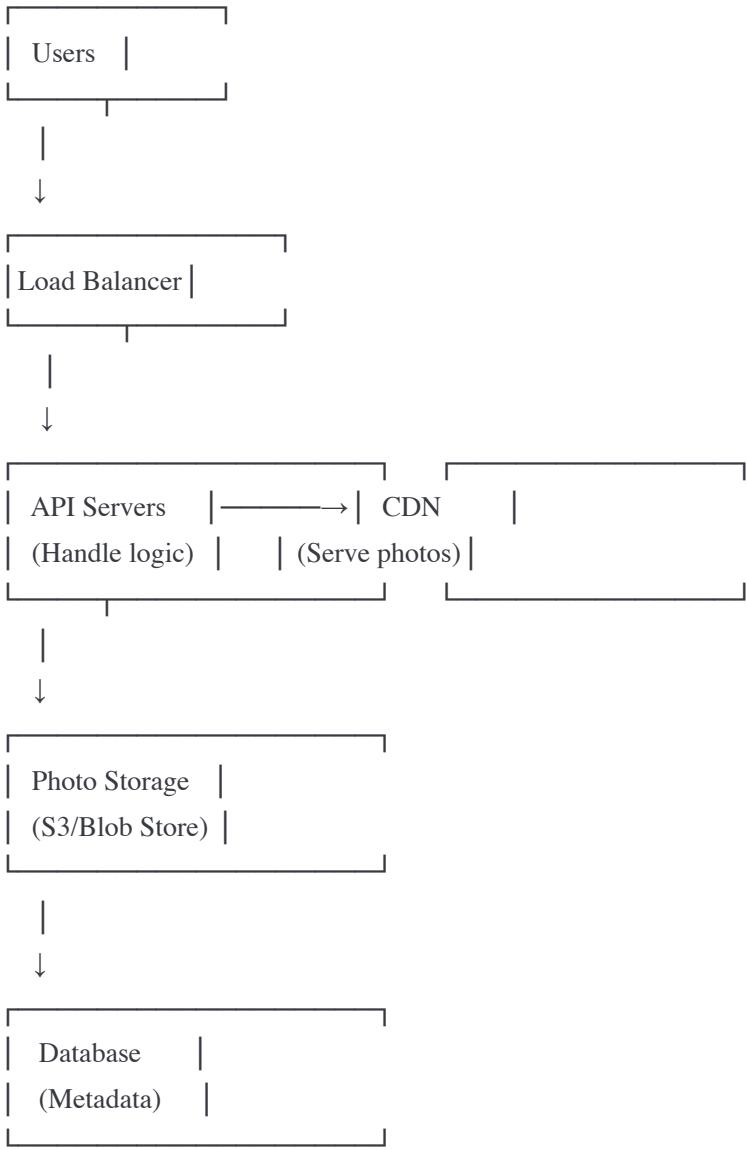
- Upload bandwidth:  $1B \text{ photos} \times 2 \text{ MB} / 86,400 \text{ sec} = 23 \text{ TB/sec}$
- Download bandwidth:  $500M \text{ users} \times 50 \text{ photos} \times 2 \text{ MB} / 86,400 \text{ sec}$   
 $= 50 \text{ billion photos} \times 2 \text{ MB} / 86,400 \text{ sec} = 1.15 \text{ PB/sec}$

This tells us we need:

- Massive storage infrastructure
- CDN for photo delivery
- Distributed system across many servers

## Phase 3: High-Level Design (15-20 min)

Draw the main components:



#### **Phase 4: Deep Dive (15-20 min)**

Interviewer picks specific components to explore:

#### **Example Questions:**

- "How would you design the feed generation algorithm?"
- "How do you ensure photos load quickly worldwide?"
- "What happens if the database goes down?"
- "How do you handle user uploads during peak times?"

#### **What They Want to Hear:**

##### **1. Trade-off discussions:**

- "We could use SQL for ACID guarantees, but NoSQL would give us better scalability"

##### **2. Multiple solutions:**

- "For caching, we could use Redis or Memcached. Redis is better if we need persistence"

### 3. Bottleneck identification:

- "The database could be a bottleneck. We should implement read replicas"

### 4. Practical knowledge:

- "We can use a CDN like CloudFront to reduce latency for global users"

## Phase 5: Wrap-up (5 min)

- Discuss monitoring and alerting
- Talk about scaling strategies
- Mention potential improvements

### Common Mistakes to Avoid

✗ Jumping to solutions without clarifying requirements ✓ Ask questions first

✗ Designing for perfection instead of iteration ✓ Start simple, then improve

✗ Ignoring trade-offs ✓ Discuss pros and cons of each decision

✗ Using buzzwords without understanding ✓ Only mention technologies you can explain

✗ Not considering bottlenecks ✓ Identify what could fail or slow down

✗ Over-engineering for the given scale ✓ Design for the requirements given

### Evaluation Criteria

Area	What They Look For
Problem Solving	Breaking down complex problems
Communication	Explaining ideas clearly
Trade-offs	Understanding pros/cons
Technical Knowledge	Knowing tools and when to use them
Scalability	Thinking about growth
Practicality	Real-world feasibility

---

## 4. Real-World System Design Examples

### Example 1: URL Shortener (Like bit.ly)

**The Problem:** Convert long URLs to short ones and redirect users.

#### Requirements:

- Long URL → Short URL (e.g., bit.ly/abc123)
- Redirect short URL to original

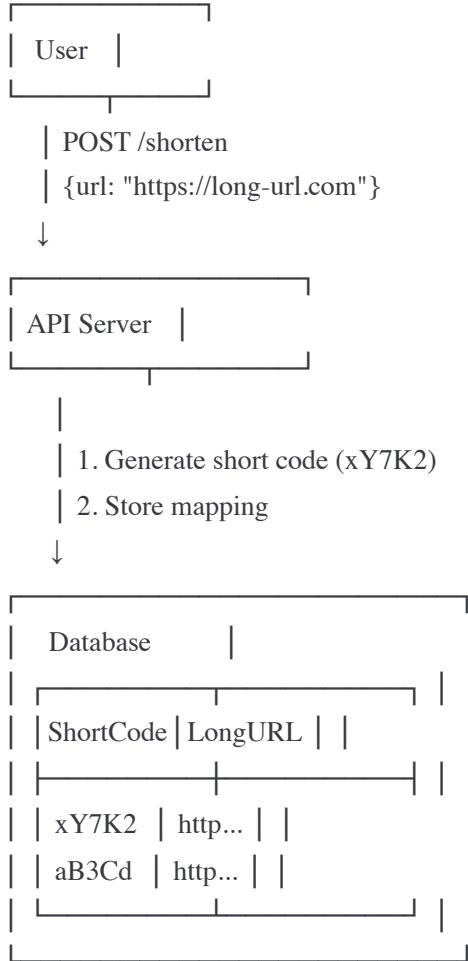
- Handle 100 million URLs
- Low latency (< 100ms)

### **Simple System Design:**

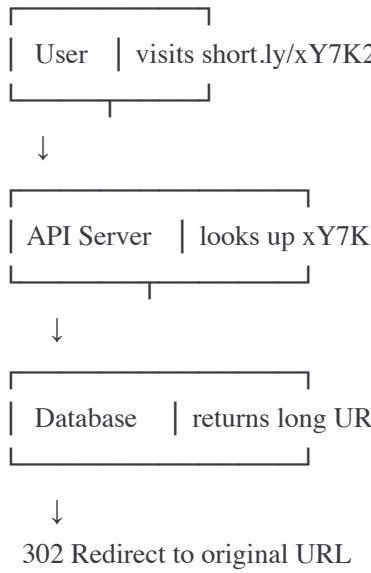
### User Flow:

1. User submits: <https://www.verylongwebsite.com/article/12345>
2. System generates: <https://short.ly/xY7K2>
3. When someone visits xY7K2, redirect to original URL

### Architecture:



### Redirect Flow:



### Key Decisions:

1. How to generate short codes? (Base62 encoding)

2. How to prevent duplicates? (Hash the URL)

3. How to scale reads? (Add caching layer)

---

## **Example 2: Twitter Feed**

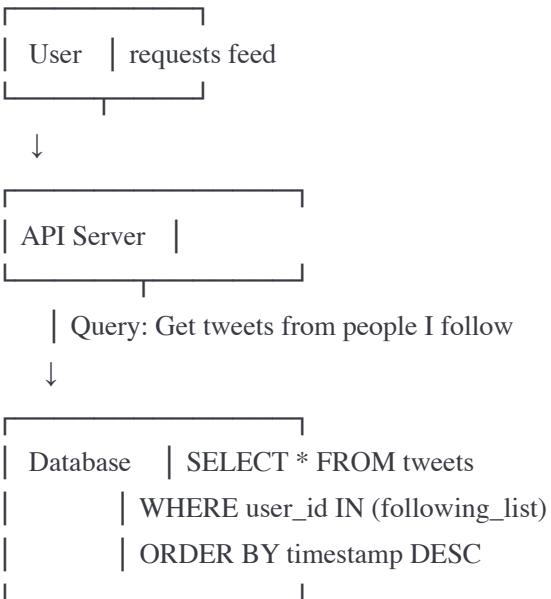
**The Problem:** Show users tweets from people they follow, in real-time.

### **Challenges:**

- User follows 1000 people
- Each person tweets multiple times
- Feed must be fast (< 300ms)
- Tweets from 500 million users

### **System Design:**

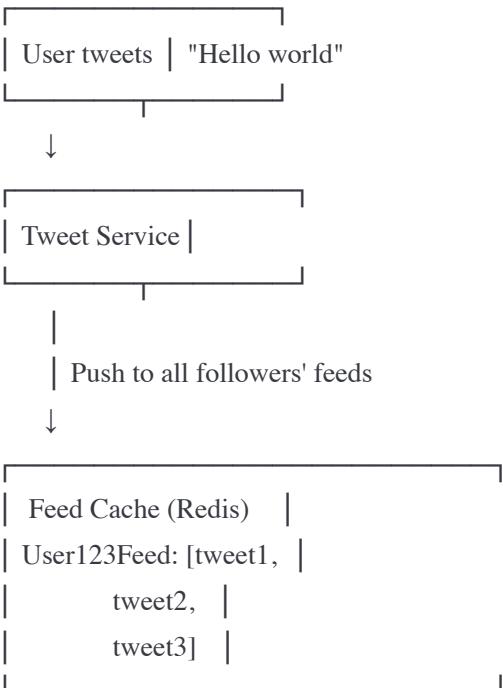
### Approach 1: Pull Model (Fetch on demand)



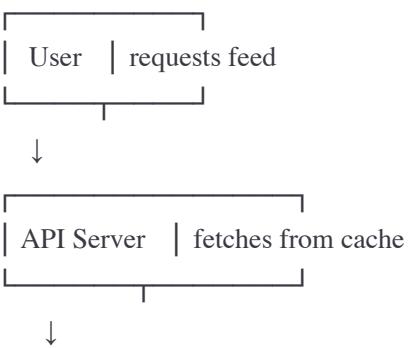
Problem: Too slow! Query takes seconds.

### Approach 2: Push Model (Pre-compute feeds)

When someone tweets:



When user opens app:



Redis Cache | Instant response!

### Trade-off:

- Pull: Slow to read, fast to write
  - Push: Fast to read, slow to write (if you have millions of followers)
  - Hybrid: Push for normal users, pull for celebrities
- 

### Example 3: WhatsApp Message Delivery

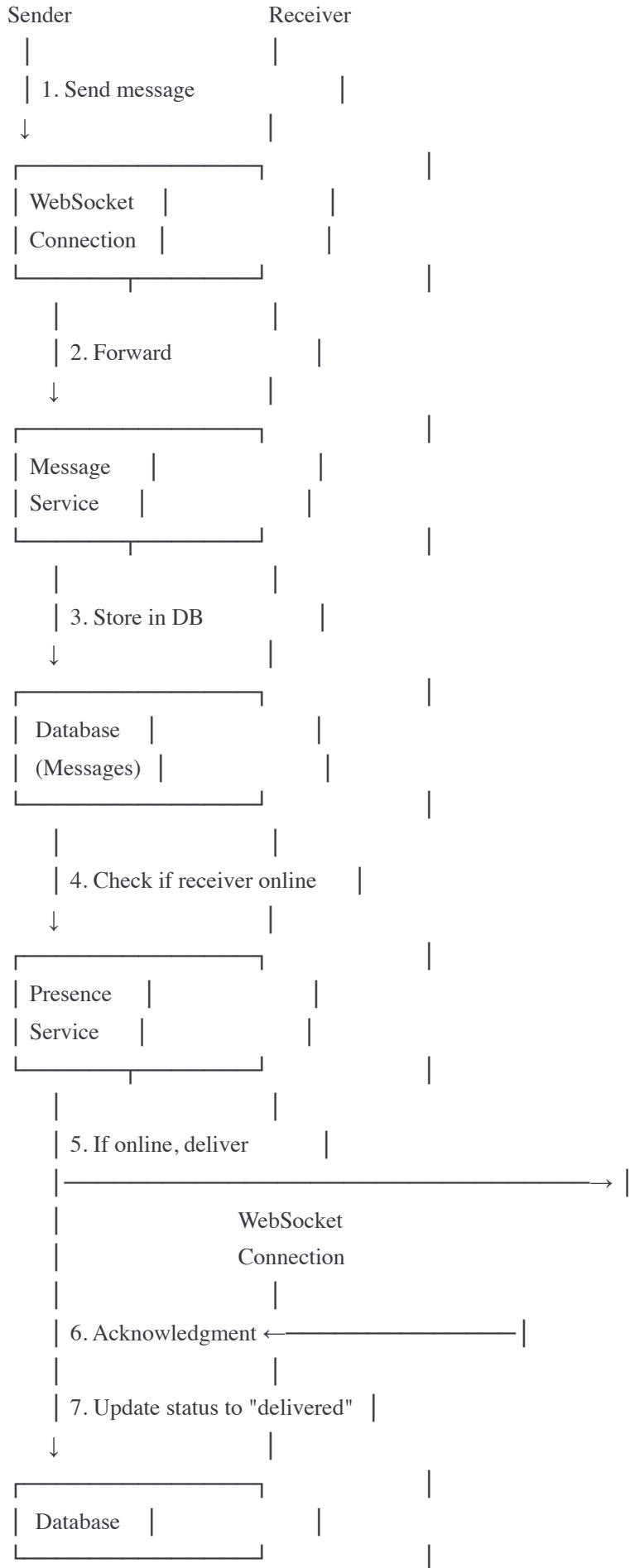
**The Problem:** Send messages between users instantly, reliably.

### Requirements:

- Message delivery in < 1 second
- Messages must not be lost
- Show "delivered" and "read" status
- Support offline users

### System Design:

## Message Flow:



If receiver offline:

- Store in database
- Send push notification
- Deliver when user comes online

## **Key Features:**

1. WebSocket for real-time connection
  2. Message queue for reliability
  3. Database for persistence
  4. Presence tracking for online/offline status
- 

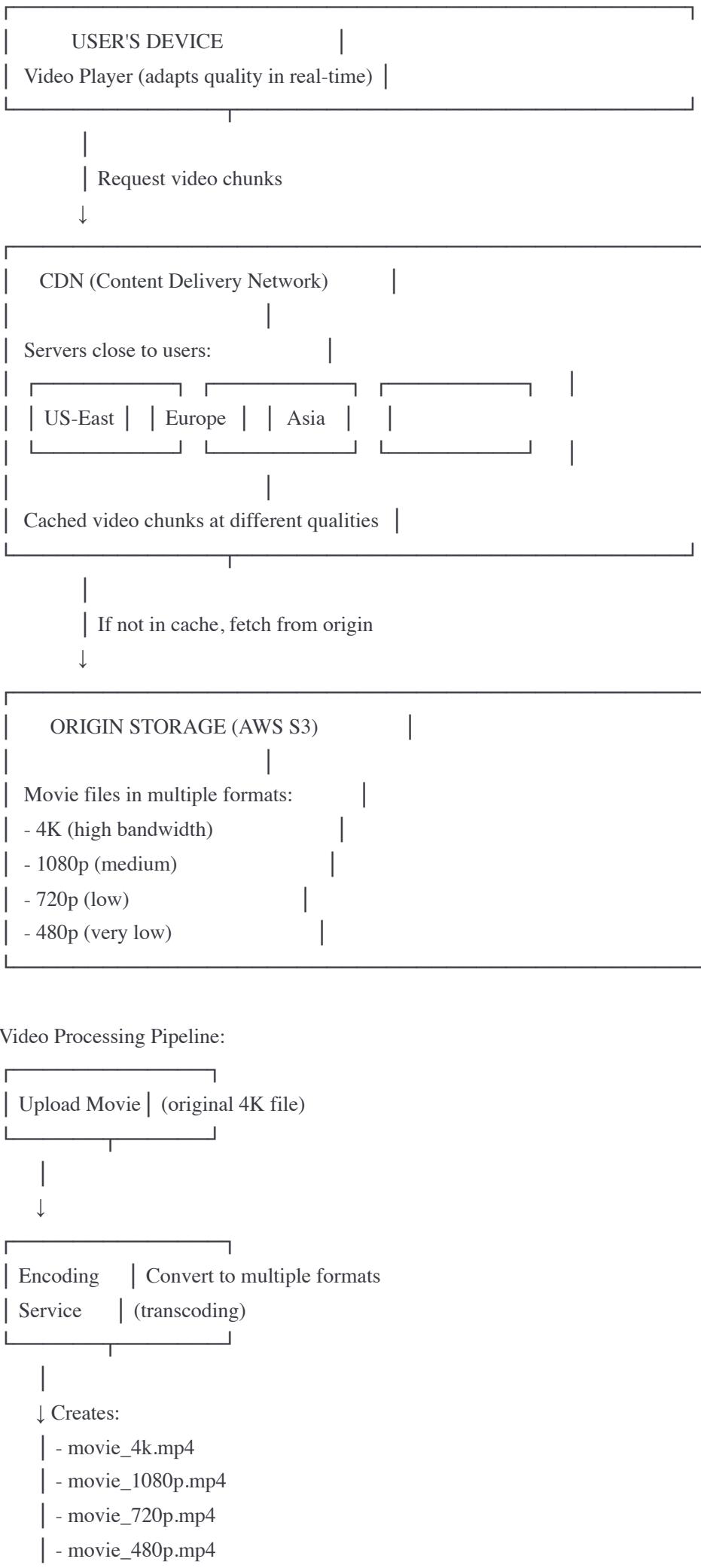
## **Example 4: Netflix Video Streaming**

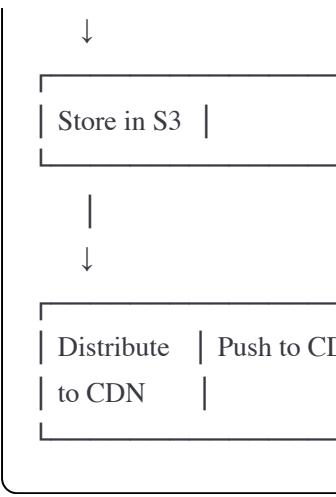
**The Problem:** Stream video to millions of users simultaneously without buffering.

### **Challenges:**

- Large file sizes (1 GB per movie)
- Global audience (different speeds)
- High availability (no downtime)
- Adaptive quality based on connection

### **System Design:**





## How Adaptive Streaming Works:

```

python

# Simplified concept
class VideoPlayer:
    def __init__(self):
        self.current_quality = '1080p'
        self.buffer_health = 100 # percentage

    def monitor_and_adapt(self):
        # Check every 2 seconds
        if self.buffer_health < 20:
            # Poor connection, lower quality
            self.switch_to_lower_quality()
        elif self.buffer_health > 80:
            # Good connection, try higher quality
            self.switch_to_higher_quality()

    def switch_to_lower_quality(self):
        quality_levels = ['4k', '1080p', '720p', '480p']
        current_index = quality_levels.index(self.current_quality)
        if current_index < len(quality_levels) - 1:
            self.current_quality = quality_levels[current_index + 1]
            # Request next chunks in lower quality

```

## Key Takeaways

1. **System Design is about scale** - How to build systems that handle millions of users
2. **It's different from coding** - Focus is on architecture, not implementation details
3. **Interviews test thinking, not memorization** - Understand concepts, trade-offs, and decisions
4. **Real systems are complex** - But start simple and iterate

**5. Every decision has trade-offs** - There's no perfect solution, only appropriate ones

## Next Steps

In Chapter 2, we'll dive into Computer Architecture Basics to understand:

- How computers process data
- Latency vs throughput
- Network fundamentals
- Why these matter for system design

Practice Questions:

1. Design a basic news feed system
2. Design a ride-sharing app like Uber (simplified)
3. Design a notification system

Remember: Start simple, ask questions, and explain your thinking!