COMPREHENSIVE GUIDE

# System Design for Beginners

A comprehensive guide to building scalable, reliable systems from the ground up

Architecture Scalability Reliability

OVERVIEW

# What We'll Cover

01

## Introduction to System Design

Understanding fundamentals and why system design matters

02

## Core Design Principles

CAP theorem, availability, consistency, and reliability

03

## Scalability Fundamentals

Horizontal vs vertical scaling strategies

04

## System Architecture Patterns

Monolithic vs microservices approaches

05

## Key Components & Building Blocks

Load balancers, caching, CDNs, and more

06

## Communication Patterns

Synchronous vs asynchronous messaging

07

## Database Design Strategies

SQL vs NoSQL, sharding, and partitioning

08

## Putting It All Together

Design framework and common patterns

CHAPTER 01

# Introduction to System Design

Understanding the fundamentals and why system design matters for building modern applications

CHAPTER 01

# What is System Design?

## Definition

System design is the process of defining the **architecture, components, modules, interfaces**, and data for a system to satisfy specified requirements. It's the blueprint that guides how software is built and scaled.

## Two Levels of Design

### High-Level Design (HLD)

Focuses on system architecture, major components, and their interactions. Answers "What are the building blocks?"

### Low-Level Design (LLD)

Details the actual implementation, class diagrams, and database schemas. Answers "How do we build it?"

## Why It Matters

### Scalability

Handle growth in users, data, and traffic

### Reliability

Ensure system availability and fault tolerance

### Career Growth

Essential for senior engineering roles

## Key Insight

Good system design is about making informed trade-offs based on requirements. There's no one-size-fits-all solution.

CHAPTER 01

# Functional vs Non-Functional Requirements

## Functional

What the system does

Define specific behaviors, features, and capabilities the system must provide. These are the **visible functions** users interact with.

### User Features

User registration, login, profile management

### Business Logic

Payment processing, order management

### Data Operations

CRUD operations, search, filtering

> Example: "Users can create short URLs from long URLs"

## Non-Functional

How the system performs

Define quality attributes and constraints on how the system operates. These determine **user experience** and system success.

### Performance

Response time < 200ms, support 10K concurrent users

### Scalability

Handle 10x traffic growth without redesign

### Availability

99.99% uptime, fault-tolerant design

> Example: "URL redirection must complete in < 100ms"

**Critical Point:** Both types are essential. Functional requirements define what to build; non-functional requirements define how well it must work.

CHAPTER 02

# Core Design Principles

The fundamental principles that guide every design decision in distributed systems

CHAPTER 02

# The CAP Theorem

## The Fundamental Trade-off

In distributed systems, you can only guarantee **two out of three** properties simultaneously. This theorem shapes every architectural decision.

**C** **Consistency**

All nodes see the same data at the same time. Every read receives the most recent write.

**A** **Availability**

Every request receives a response, without guarantee it contains the most recent write.

**P** **Partition Tolerance**

System continues to operate despite network failures between nodes.

## CP vs AP Systems

**CP Systems** Consistency + Partition Tolerance

Prioritize data accuracy over availability. When a partition occurs, the system may refuse requests to ensure consistency.

Banking systems · Inventory management · HBase · MongoDB (configurable)

**AP Systems** Availability + Partition Tolerance

Prioritize system responsiveness. May serve stale data during partitions but remains operational.

Social media feeds · DNS · Cassandra · DynamoDB

Key Insight: In distributed systems, network partitions will happen. Therefore, **P is mandatory**. The real choice is between C and A.

## Interview Tip

Always ask: "Does this system need to prioritize consistency or availability?" This should be one of the first questions in any system design discussion.

CHAPTER 02

# Key System Qualities

## Scalability

The ability to handle growth in users, traffic, or data without performance degradation.

- Horizontal scaling (add servers)
- Vertical scaling (upgrade hardware)

## Availability

The percentage of time the system is operational and accessible to users.

- 99.9% = 8.76 hrs downtime/year
- 99.99% = 52 mins downtime/year

## Reliability

The ability to operate correctly and recover from failures without data loss.

- Fault tolerance mechanisms
- Redundancy and replication

## Latency

The time taken to respond to a request. Critical for user experience.

- Network latency (round-trip time)
- Processing latency (computation)

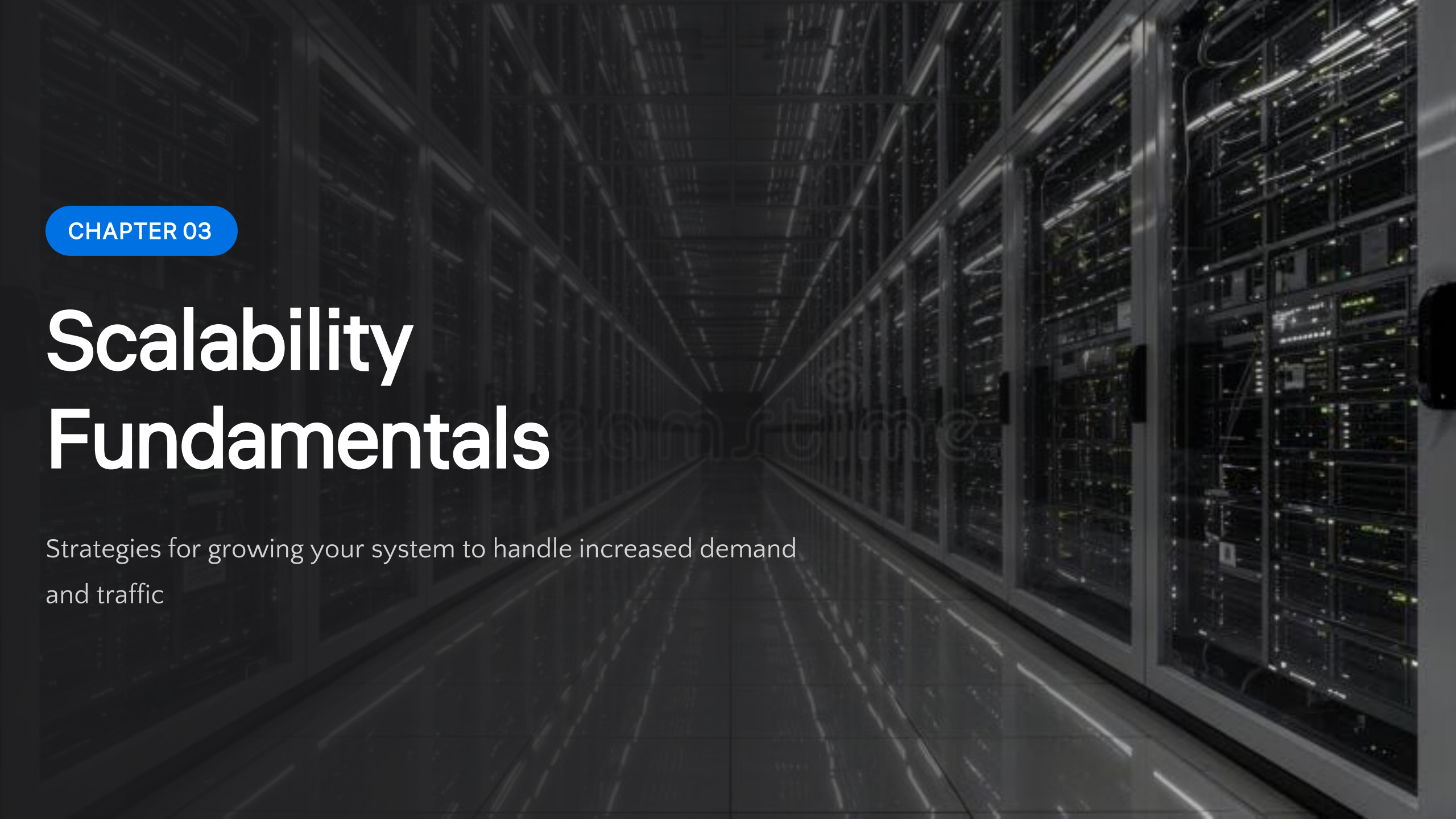
## Throughput

The number of requests the system can handle per unit of time.

- Requests per second (RPS)
- Transactions per second (TPS)

## Design Impact

These qualities often conflict. Improving one may compromise another. System design is about finding the right balance for your specific requirements.



CHAPTER 03

# Scalability Fundamentals

Strategies for growing your system to handle increased demand and traffic

CHAPTER 03

# Horizontal vs Vertical Scaling

## Horizontal Scaling

Scale Out

Add **more machines** to distribute the workload across multiple servers.

### Advantages

- No single point of failure
- Unlimited scaling potential
- Better fault tolerance
- Cost-effective at scale

### Challenges

- Complex architecture
- Data consistency issues
- Network communication overhead
- Requires load balancing

**Best for:** Large-scale applications, high availability needs, unpredictable growth

## Vertical Scaling

Scale Up

Upgrade **existing machine** with more CPU, RAM, or storage.

### Advantages

- Simple to implement
- No code changes needed
- Faster inter-process communication
- Data consistency maintained

### Challenges

- Hardware limitations exist
- Single point of failure
- Downtime during upgrades
- Expensive at high end

**Best for:** Small applications, startups, predictable workloads

**Real-World Approach:** Most systems start with vertical scaling for simplicity, then transition to horizontal scaling as they grow. Many use a hybrid approach.

CHAPTER 04

# System Architecture Patterns

Common architectural approaches and when to use them

CHAPTER 04

# Monolithic vs Microservices

## Monolithic

Single Unified Codebase

All components are tightly coupled in a single application. One codebase, one deployment unit.

### Advantages

- Simple to develop and test
- Easy deployment (one artifact)
- Better performance (no network calls)
- Simple debugging and monitoring

### Challenges

- Tight coupling between components
- Difficult to scale individual parts
- Technology stack locked
- Large codebase becomes unwieldy

**Best for:** Small teams, MVPs, simple applications

## Microservices

Independent Services

Application is split into small, independent services. Each service handles a specific business function.

### Advantages

- Independent deployment and scaling
- Technology diversity per service
- Fault isolation
- Team autonomy

### Challenges

- Distributed system complexity
- Network latency and failures
- Data consistency challenges
- Operational overhead

**Best for:** Large teams, complex domains, need for independent scaling

Decision Framework: Start monolithic, extract to microservices when you have clear boundaries and pain points. Don't start with microservices "just because."

CHAPTER 05

# Key Components & Building Blocks

Essential components every scalable system needs

CHAPTER 05

# Load Balancing

## What is Load Balancing?

A load balancer distributes incoming network traffic across multiple servers to ensure **no single server bears too much load**. It acts as a traffic cop, routing requests to available resources.

### Key Benefits

- ✓ Prevents any single server from becoming a bottleneck
- ✓ Improves application availability and reliability
- ✓ Enables horizontal scaling by adding more servers
- ✓ Provides failover when servers go down

### Health Checks

Load balancers continuously monitor server health and automatically remove unhealthy servers from the pool, routing traffic only to healthy instances.

## Load Balancing Algorithms

### Round Robin

Requests are distributed sequentially to each server in rotation. Simple and effective for stateless applications.

Simple | Stateless | Even distribution

### Least Connections

Routes to the server with the fewest active connections. Adapts to variable request processing times.

Dynamic | Adaptive | Better for long requests

### # IP Hash

Uses client's IP address to determine which server receives the request. Ensures session persistence.

Session affinity | Consistent routing

CHAPTER 05

# Caching Strategies

## What is Caching?

Caching stores copies of frequently accessed data in fast, temporary storage to **reduce latency** and **decrease load** on primary data stores.

### Why Cache?

- 10x Memory access is ~10x faster than database queries
- Reduces database load and infrastructure costs
- Improves user experience with faster responses

## Cache Locations

**Client**
Browser cache

**CDN**
Edge servers

**Application**
In-memory

**Database**
Query cache

## Cache Eviction Policies

When cache is full, these policies determine which data to remove:

### LRU Least Recently Used

Removes data that hasn't been accessed for the longest time. Best for workloads with temporal locality.

### LFU Least Frequently Used

Removes data accessed least often. Better for stable, predictable access patterns but more complex to implement.

### TTL Time-To-Live

Data expires after a predefined time period. Ideal for session data and time-sensitive information.

## Cache Invalidation

One of the hardest problems in computer science. Strategies include: write-through (update cache and DB together), write-back (update cache first), and TTL-based expiration.

CHAPTER 05

# Content Delivery Networks (CDN)

## What is a CDN?

A CDN is a **geographically distributed network of servers** that caches and delivers content closer to users, reducing latency and improving performance.

### How It Works

1. User requests content (image, video, webpage)
2. DNS routes request to nearest CDN edge server
3. If cached, content served directly from edge
4. If not cached, fetch from origin and cache for future

### Popular CDN Providers

Cloudflare | AWS CloudFront | Akamai | Fastly

## Key Benefits

### Reduced Latency

Content served from edge servers closer to users, reducing round-trip time

### Origin Offload

Reduces load on origin servers, lowering infrastructure costs

### Improved Availability

Distributed architecture provides redundancy and fault tolerance

### Security Features

DDoS protection, WAF, SSL/TLS encryption at the edge

## Common Use Cases

- Static assets
- Video streaming
- Software downloads
- API responses

CHAPTER 06

# Communication Patterns

How components talk to each other in distributed systems

CHAPTER 06

# Synchronous vs Asynchronous Communication

## Synchronous

Request-Response Pattern

The caller **waits** for a response before proceeding. Creates tight coupling between services.

### Advantages

- Simple to understand and implement
- Immediate feedback on success/failure
- Consistent state across services
- Easy error handling

### Challenges

- Tight coupling between services
- Cascading failures possible
- Slower overall response time
- Requires services to be available

**Protocols:** HTTP/REST, gRPC, GraphQL

**Best for:** Real-time queries, simple operations, strong consistency needs

## Asynchronous

Event-Driven Pattern

The caller **doesn't wait** for a response. Services communicate through events or messages.

### Advantages

- Loose coupling between services
- Better fault tolerance
- Improved scalability
- Services can be temporarily unavailable

### Challenges

- More complex to implement
- Eventual consistency
- Harder to debug and trace
- Requires message infrastructure

**Technologies:** Message queues, Event buses, Kafka, RabbitMQ

**Best for:** Background processing, high throughput, decoupled systems

**Hybrid Approach:** Modern systems often use both patterns. Synchronous for user-facing queries, asynchronous for background processing and event propagation.

CHAPTER 06

# Message Queues & Event Streaming

## RabbitMQ

Message Queue

A **message broker** that enables applications to communicate by sending and receiving messages through queues.

### Key Characteristics

- **Push-based:** Messages pushed to consumers
- **Message deletion:** Removed after acknowledgment
- **Smart broker:** Manages message routing
- **Throughput:** 4K-10K messages/second

### Best Use Cases

Background jobs | Task queues | RPC patterns

## Apache Kafka

Event Streaming Platform

A **distributed event streaming platform** designed for high-throughput, fault-tolerant, real-time data pipelines.

### Key Characteristics

- **Pull-based:** Consumers fetch messages
- **Message retention:** Persists for configured time
- **Replay capability:** Can re-read old messages
- **Throughput:** 1M+ messages/second

### Best Use Cases

Event sourcing | Log aggregation | Real-time analytics

**Key Difference:** RabbitMQ is designed for task distribution (messages consumed and deleted), while Kafka is designed for event streaming (events persisted and replayable).

CHAPTER 07

# Database Design Strategies

Scaling and organizing data for performance and reliability

CHAPTER 07

# SQL vs NoSQL: Choosing the Right Database

## SQL

Relational Databases

Structured data with predefined schemas. Tables with rows and columns, supporting complex relationships.

### Key Features

- ✓ **ACID compliance:** Atomicity, Consistency, Isolation, Durability
- ✓ **Structured schema:** Predefined tables and relationships
- ✓ **Complex queries:** JOINs, aggregations, subqueries
- ✓ **Strong consistency:** Immediate data consistency

**Examples:** PostgreSQL, MySQL, SQL Server, Oracle

**Best for:** Complex relationships, transactional systems, financial data

## NoSQL

Non-Relational Databases

Flexible schemas designed for specific data models: documents, key-value, wide-column, or graph.

### Key Features

- ✓ **Flexible schema:** Dynamic, evolving data structures
- ✓ **Horizontal scaling:** Easy to distribute across servers
- ✓ **High write throughput:** Optimized for write-heavy workloads
- ✓ **Eventual consistency:** May have temporary inconsistencies

**Examples:** MongoDB (Document), Redis (Key-Value), Cassandra (Wide-column), Neo4j (Graph)

**Best for:** Unstructured data, high write loads, rapid iteration

**Modern Approach:** Many systems use both! SQL for transactional data, NoSQL for logs, caching, and unstructured content. This is called polyglot persistence.

CHAPTER 07

# Database Sharding & Partitioning

## What is Sharding?

Sharding is splitting a large database into **smaller, more manageable pieces** called shards, distributed across multiple servers.

### Why Shard?

- ✓ Handle datasets larger than single server capacity
- ✓ Distribute write load across multiple servers
- ✓ Improve query performance with smaller datasets
- ✓ Geographic distribution for lower latency

### Partitioning vs Sharding

**Partitioning** divides data within a single database. **Sharding** distributes data across multiple database servers.

## Sharding Strategies

### Range-Based

Data divided by ranges (e.g., User IDs 1-1000 on Shard 1, 1001-2000 on Shard 2).

Simple | Good for range queries | Hotspot risk

### Hash-Based

Hash function distributes data evenly across shards (e.g., hash(user_id) % num_shards).

Even distribution | No hotspots | Range queries hard

### Geographic

Data partitioned by location (e.g., EU users on EU servers, US users on US servers).

Low latency | Compliance | Uneven distribution

Important: Sharding adds significant complexity. Only shard when a single database can't handle your load. Consider alternatives like read replicas and caching first.

CHAPTER 08

# Putting It All Together

Applying system design principles to real-world scenarios

CHAPTER 08

# Design Process Framework

## Step-by-Step Approach

### 1 Understand Requirements

Clarify functional requirements (features) and non-functional requirements (scalability, availability, latency). Ask questions to fill gaps.

### 2 Estimate Scale

Calculate expected traffic, data volume, and storage needs. This informs your technology choices and architecture decisions.

### 3 Design High-Level Architecture

Identify major components: load balancers, application servers, databases, caches. Draw the system diagram.

### 4 Deep Dive into Components

Detail each component: database schema, API design, caching strategy, communication patterns between services.

### 5 Identify and Address Bottlenecks

Analyze potential bottlenecks: database queries, network latency, single points of failure. Propose solutions.

### 6 Discuss Trade-offs

Every design decision involves trade-offs. Discuss alternatives and justify your choices based on requirements.

## Key Principles

### Start Simple

Begin with a basic design and iterate. Don't over-engineer from the start.

### Embrace Trade-offs

There's no perfect solution. Every choice has pros and cons.

### Communicate Clearly

Explain your reasoning. Interviewers value clear thinking over perfect answers.

## Common Questions to Ask

- What's the expected scale?
- Read-heavy or write-heavy?
- Consistency or availability priority?
- What's the acceptable latency?

CHAPTER 08

# Common Design Patterns Summary

## Load Balancer

Distributes traffic across multiple servers to prevent overload and improve availability.

**Use when:** Multiple servers, need high availability

## Caching

Stores frequently accessed data in fast storage to reduce latency and database load.

**Use when:** Read-heavy workloads, high latency

## Database Sharding

Splits large databases across multiple servers to handle more data and traffic.

**Use when:** Data exceeds single server capacity

## Message Queue

Decouples services by allowing asynchronous communication through message passing.

**Use when:** Background processing, service decoupling

## API Gateway

Single entry point for clients, handling routing, authentication, and rate limiting.

**Use when:** Microservices architecture

## Circuit Breaker

Prevents cascading failures by stopping requests to failing services temporarily.

**Use when:** Fault tolerance, external service calls

**Remember:** These patterns are tools, not rules. Choose based on your specific requirements and constraints. The best designers know when NOT to use a pattern.

# Key Takeaways

System design is about making **informed trade-offs** based on requirements. There's no one-size-fits-all solution.

Start simple, understand your constraints, and **evolve your design** as needs grow. Don't over-engineer prematurely.

Master the fundamentals: **scalability, availability, and consistency**. These principles guide every design decision.

Keep Learning  Practice Designing  Discuss Trade-offs