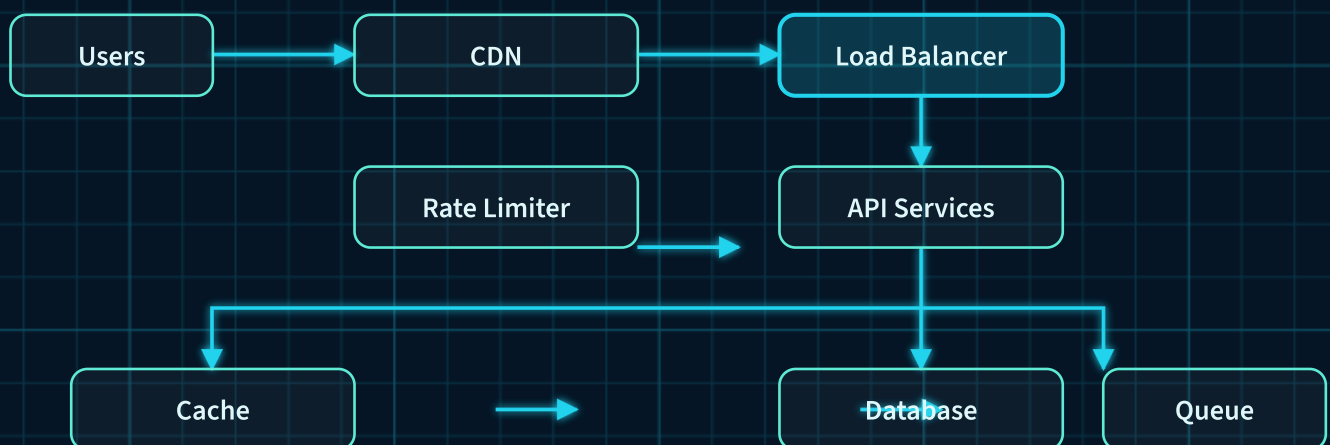


SYSTEM DESIGN

From Zero to Hero · Build systems for millions

Scalable, reliable systems in simple language: caching, sharding, queues, CAP, consensus, and 25+ real case studies from WhatsApp to Uber, with facts, humour and answers.



blueprint v2.0 · scale: 10 million users

Load Balancing

Caching

Sharding

Replication

CAP

Kafka

Rate Limiting

Consistent Hashing

Microservices

CDN

Saga

Raft

WRITTEN BY

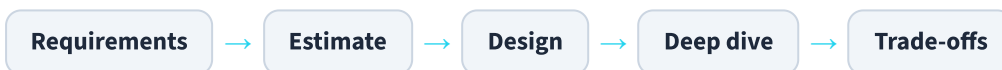
Shreyash Tripathi

JOB-READY SERIES • COMPLETE EDITION



System Design

From Zero to Hero



WRITTEN BY

Shreyash Tripathi

Complete Edition, 2026

© Shreyash Tripathi. All rights reserved.

Preface

Every app you love, from UPI payments to WhatsApp to IRCTC at 10 a.m. on a Tatkal morning, is a **system**: many computers, networks and databases working together so that millions of people can press a button at the same time and still get the right answer. **System design** is the skill of planning that system before you build it.

Most system design material is written for people who already know the answers. This book is written for people who are starting from zero. It follows the style of a school textbook on purpose: clear chapters, numbered topics, examples from daily life, small checks after every topic, and exercises with complete answers at the end of every chapter. You should never feel lost, and you should never have to search elsewhere to check whether your answer was right.

System design can feel dry, so we have added a little masala. Every chapter has "Funny Side" boxes, because a joke about a cache that forgot everything is easier to remember than a definition. The jokes are for memory; the facts are for real.

Read slowly. Draw every diagram yourself. Answer the questions before you look at the answers. If you do that for all 28 chapters, you will be able to design real systems and walk into a system design interview with confidence.

Shreyash Tripathi

How This Book Is Organised

Part	Chapters	What you learn
Part I: Foundations	1 to 5	What system design is, estimation, networking, APIs, security
Part II: Data	6 to 12	SQL, NoSQL, indexes, scaling, caching, replication, sharding
Part III: Distributed Systems	13 to 15	CAP, consensus, transactions across services, queues and streams
Part IV: Production Systems	16 to 20	Microservices, reliability, monitoring, storage and search, real-time
Part V: Interview and Case Studies	21 to 28	The interview method and 25+ complete designs

Features of Every Chapter

DO YOU KNOW?

Real facts from real companies and history.

FUNNY SIDE

A joke or analogy that makes the idea stick.

KEY POINT

The one line to remember for exams and interviews.

ACTIVITY

Something to draw, calculate or try yourself.

THINK AND DISCUSS

Open questions with no single right answer.

COMMON MISTAKE

Traps that beginners (and seniors) fall into.

CHECK YOUR PROGRESS

After every topic: short questions with answers, so you can test yourself immediately.

At the end of every chapter you will find **What Have We Learnt?** (summary), **Key Terms**, **Exercises** (MCQ, fill in the blanks, true or false, short and long answers, a design challenge) and a complete **Answer Key**.

Suggested Study Plan (12 Weeks, about 1 Hour a Day)

Weeks	Read	Practice
1 to 2	Chapters 1 to 5	Estimate traffic for 3 apps you use daily
3 to 5	Chapters 6 to 12	Design the database for a food delivery app
6 to 7	Chapters 13 to 15	Explain CAP and Saga to a friend without notes
8 to 9	Chapters 16 to 21	Draw your own project's architecture
10 to 12	Chapters 22 to 28	Two case studies a week, timed at 45 minutes, then compare

Contents

PART I: FOUNDATIONS

1	Introduction to System Design	8
2	Estimation and Numbers Every Engineer Should Know	37
3	Networking Fundamentals	63
4	APIs and Communication	98
5	Authentication, Authorization and Security	128

PART II: DATA

6	Relational Databases and SQL	157
7	NoSQL Databases and Choosing the Right Store	184
8	Indexes and Storage Engines	212
9	Scaling and Load Balancing	236
10	Caching and CDNs	261
11	Replication	293
12	Partitioning, Sharding and Consistent Hashing	319

PART III: DISTRIBUTED SYSTEMS

13	Distributed Systems Fundamentals	346
14	Distributed Transactions, Idempotency and IDs	376
15	Message Queues, Pub/Sub and Stream Processing	403

PART IV: PRODUCTION SYSTEMS

16	Architecture Styles and Microservices	431
17	Reliability, Resilience and Rate Limiting	463
18	Observability and Monitoring	491
19	Storage Systems, Search and Special Data Structures	515
20	Real-Time Systems	545

PART V: INTERVIEW AND CASE STUDIES

21	The System Design Interview	570
22	Case Studies I: Building Blocks	594
23	Case Studies II: Web Classics	621
24	Case Studies III: Messaging	644
25	Case Studies IV: Social and Search	666
26	Case Studies V: Media and Files	694
27	Case Studies VI: Location and Booking	721
28	Case Studies VII: Money and Data	750

APPENDICES

A	Practice Project: Design Your Own Product	785
B	Quick Revision Cheat Sheet	787
C	Glossary	789
D	Sample Question Paper with Answers	792
E	Further Reading and Resources	796

PART I

Foundations

How the internet works, how to estimate, how services talk, and how to keep them safe.

CHAPTER 1

Introduction to System Design

"It works on my laptop" is not a system design. It is a confession.

After studying this chapter, you will be able to

- Explain what system design is and tell the difference between high-level design (HLD) and low-level design (LLD).
- Describe the client-server model and the simple single-server setup most products start with.
- Separate functional requirements from non-functional requirements for any product.
- Define the key qualities of a system: scalability, availability, reliability, latency, throughput, consistency, durability, maintainability and cost.
- Identify single points of failure and bottlenecks in a simple design.
- Read and use the "nines" availability table and reason about trade-offs.
- Trace how a product grows from one server to many, and know who does system design in a company.

Imagine Raju runs a small chai stall near a college gate. On a normal day, about 50 students come. Raju takes the order, boils the tea, pours it, collects the money and washes the glasses. One person, one stove, one kettle. It works perfectly.

Then a famous food blogger posts a video: "Best cutting chai in the city!" The next morning, 800 students are standing in line. Raju's single stove cannot boil tea fast enough. The glasses run out. People leave angry. If Raju falls sick, the whole stall is closed. Raju now has a **system design** problem: how do you take something that works for 50 people and make it work for 800, or 80,000, without it falling apart?

Software has exactly the same problem. A website that runs happily on one laptop for ten friends will collapse when a million people open it at once. This book is about how engineers plan for that. In this first chapter we will learn the basic words, the basic ideas and, most importantly, the basic way of thinking.

1.1 What Is System Design?

System design is the process of deciding the parts of a software system, what each part does, and how the parts talk to each other, so that the whole system meets its goals. Those goals are not only

"it should work". They include "it should be fast", "it should not lose data", "it should stay up even if a machine dies" and "it should not cost more than we earn".

Writing code is like cooking one dish. System design is like planning the kitchen of a big wedding: how many stoves, how many cooks, where the food is stored, how plates reach 2,000 guests, and what happens if the gas cylinder finishes in the middle of the event.

1.1.1 High-Level Design (HLD)

High-level design looks at the system from far away, like a map of a city. It answers questions such as:

- What are the main components? (web servers, databases, caches, queues, storage)
- How does data flow from the user to the database and back?
- Which database type do we use: relational or NoSQL?
- How do we handle ten times more users next year?
- What happens if one data centre goes down?

An HLD is usually drawn as boxes and arrows. It does not care about variable names or class structure.



Figure 1.1: A typical high-level design view: big boxes, no code details.

1.1.2 Low-Level Design (LLD)

Low-level design zooms into one box. It answers questions such as:

- What classes, interfaces and functions will the app server have?
- What are the exact database tables, columns and indexes?
- What are the API endpoints, their inputs and outputs?
- Which design patterns (Factory, Strategy, Observer) do we use?

If HLD is the city map, LLD is the floor plan of one building, showing every room and every door.

Aspect	High-Level Design (HLD)	Low-Level Design (LLD)
Zoom level	Whole system	One component or module
Main question	Which parts, and how do they connect?	How is each part built inside?
Output	Architecture diagram, data flow, technology choices	Class diagrams, schemas, API contracts, pseudocode
Typical owner	Architects, senior and staff engineers	Developers and tech leads

Aspect	High-Level Design (HLD)	Low-Level Design (LLD)
Example decision	"Use a message queue between order service and email service"	"The <code>OrderService</code> class has a <code>placeOrder(cartId)</code> method that returns an <code>OrderId</code> "
Interview style	"Design Instagram"	"Design a parking lot class structure"

KEY POINT

HLD decides *what* boxes exist and how they talk. LLD decides *how* each box is built inside. This book mostly focuses on HLD, with LLD details where they help.

1.1.3 Why Does System Design Matter?

Bad system design rarely shows up on day one. It shows up on the day you become successful, which is the worst possible day for things to break. Good design matters because:

1. **Users leave slow apps.** If a page takes too long to load, people close it and go elsewhere.
2. **Downtime costs money and trust.** When a payment app is down during a festival sale, the company loses both revenue and reputation.
3. **Lost data cannot be "un-lost".** A bank that loses a transaction has a legal problem, not just a technical one.
4. **Rewrites are expensive.** Changing the core design of a system with millions of users is like changing the engine of a plane while it is flying.
5. **Cloud bills are real.** A careless design can cost ten times more in servers than a careful one.

The good news: you do not need to design for a billion users on day one. You need to design for today, with a clear idea of what you would change when you grow. That is the mindset this book teaches.

DO YOU KNOW?

In the early years of Twitter (around 2007 to 2008), the site went down so often under heavy load that its error page, a cartoon whale being lifted by small birds, became famous as the "Fail Whale". Twitter later rebuilt large parts of its system to handle the growth.

✓ CHECK YOUR PROGRESS 1.1

1. What is system design in one sentence?

Answer: System design is deciding the components of a software system, their responsibilities and how they communicate, so that the system meets its functional and non-functional goals such as speed, uptime and cost.

2. Is choosing between PostgreSQL and Cassandra an HLD or an LLD decision?

Answer: It is an HLD decision, because it is about which major component (type of database) the system uses, not about internal code structure.

3. Give one example of an LLD artefact.

Answer: A database schema with exact tables, columns and indexes, or a class diagram for a module, or an API contract with request and response fields.

1.2 The Client-Server Model

Almost every system in this book follows the **client-server model**. A **client** is the program that asks for something. A **server** is the program that answers. Your browser, your mobile app, even your smart TV are clients. The machines run by Google, Swiggy or IRCTC are servers.

1.2.1 How a Request Travels

1. You type `www.example.com` or tap a button in an app.
2. The client finds the server's address (using DNS, covered in Chapter 3).
3. The client opens a connection and sends a **request**, for example "GET me the home page".
4. The server does some work: reads a database, runs some logic, builds a response.
5. The server sends a **response** back: the page, some JSON data, or an error.
6. The client shows the result to you.



Figure 1.2: The request-response cycle of the client-server model.

Everyday analogy: In a restaurant, you (the client) give your order to the waiter. The kitchen (the server) cooks it and the waiter brings it back. You do not walk into the kitchen and cook yourself. The kitchen serves hundreds of customers; each customer only deals with their own plate.

1.2.2 Why Not Let Clients Do Everything?

Why do we need servers at all? Why not keep all the data on each phone?

- **Shared data:** Your WhatsApp message must reach your friend's phone. Someone in the middle must hold and pass it.
- **Trust:** Your bank balance cannot live on your phone, or you could just edit it to one crore rupees.
- **Power:** Searching the whole internet needs thousands of machines, not one phone.

- **Updates:** Fixing a bug on the server fixes it for everyone at once.

1.2.3 Thin Clients, Thick Clients and Peer-to-Peer

A **thin client** does very little work itself (an old web page that just shows HTML from the server). A **thick client** does a lot of work locally (a mobile game, or Google Docs, which runs a lot of logic in your browser). Most modern apps are somewhere in between.

There is also the **peer-to-peer (P2P)** model, where every machine is both client and server. BitTorrent works this way: you download pieces of a file from many other users and upload pieces to others. P2P removes the central server but makes control, security and search much harder. Most products you use daily are client-server.

😄 FUNNY SIDE

The client-server relationship is like a typical Indian family WhatsApp group. The clients (relatives) keep sending requests ("Good morning" images, forwarded news). The server (the one patient cousin) has to respond to all of them, never sleeps and is blamed when anything goes wrong.

✅ CHECK YOUR PROGRESS 1.2

1. In the client-server model, who starts the conversation?

Answer: The client. It sends a request, and the server sends a response. (Later we will see techniques like WebSockets where the server can also push data, but the connection is still opened by the client.)

2. Why can't a bank keep your balance only on your phone?

Answer: Because the phone is not trusted: the user could change the data. Shared, trusted data must live on a server controlled by the bank.

3. Name one system that uses peer-to-peer instead of client-server.

Answer: BitTorrent, where users download and upload file pieces directly from each other.

1.3 Starting Simple: The Single-Server Monolith

Almost every famous product started small. When you build version one of an app, you usually have:

- One server (a virtual machine in the cloud, or even a spare computer).
- One application, containing all the code: login, products, orders, payments, emails. This is called a **monolith**.
- One database, often on the same machine.



Figure 1.3: The classic starting point: everything on one machine.

1.3.1 Why Start With a Monolith?

A monolith is not a mistake. It is usually the right choice at the start because:

- **It is simple.** One codebase, one deployment, one place to look for bugs.
- **It is cheap.** One small server can cost less than a family dinner per month.
- **It is fast to change.** When you are still discovering what users want, you need speed, not ceremony.
- **Function calls are fast.** Parts of a monolith talk through in-memory function calls, which take nanoseconds, instead of network calls, which take milliseconds.

1.3.2 Where the Monolith Hurts

As traffic grows, the single server starts to struggle:

- **CPU runs out:** too many requests to process.
- **Memory runs out:** too many sessions and too much data to hold.
- **Disk becomes slow:** the database fights the app for disk access.
- **One crash kills everything:** if the machine dies, the whole product is offline.
- **Deployments are scary:** every small change requires restarting the only server.

Raju's chai stall with one stove is a monolith. It is perfect for 50 customers. At 800 customers, he needs more stoves, more helpers and a backup plan if the stove breaks.

KEY POINT

Start simple. Make it work, then make it scale. Good system designers do not add complexity before there is a reason for it, but they do know what they will change when the reason arrives.

DO YOU KNOW?

Instagram had about 13 employees when Facebook agreed to buy it for about 1 billion US dollars in 2012. Its early engineering blog posts described a fairly simple stack (Django, PostgreSQL, Redis and a few other tools), showing that a small team with simple, well-chosen tools can serve tens of millions of users.

✓ CHECK YOUR PROGRESS 1.3

1. What is a monolith?

Answer: A monolith is a single application in which all features (login, orders, payments and so on) live in one codebase and are deployed together as one unit.

2. Give two reasons why a startup should begin with a monolith.

Answer: It is simple to build, deploy and debug, and it is cheap. It also lets the team change features quickly while they learn what users want.

3. What happens to a single-server system if that server's hard disk fails?

Answer: The entire product goes down and data may be lost, because there is no second copy and no second machine. The server is a single point of failure.

1.4 Functional and Non-Functional Requirements

Before designing anything, you must know what you are building. Requirements come in two types.

1.4.1 Functional Requirements

Functional requirements describe *what* the system does: its features. They usually read like "a user can...". For a food delivery app:

- A user can search restaurants near them.
- A user can add items to a cart and place an order.
- A user can pay using UPI, card or cash on delivery.
- A delivery partner can see assigned orders and update status.
- A user can track the delivery partner live on a map.

1.4.2 Non-Functional Requirements

Non-functional requirements (NFRs) describe *how well* the system does it: speed, uptime, safety, scale. For the same app:

- Search results should load in under 200 milliseconds for 99% of requests.
- The ordering system should be available 99.95% of the time.
- No confirmed payment should ever be lost.
- The system should handle 5 times normal traffic during dinner peak and festival sales.
- Location updates on the map may be a few seconds old; that is acceptable.

Question	Functional	Non-functional
What does it describe?	Features and behaviour	Qualities and constraints
Typical sentence	"The user can upload a photo."	"Upload must finish within 2 seconds on 4G."
Who usually defines it?	Product managers, business	Engineers with business input

Question	Functional	Non-functional
Effect on design	Decides which APIs and data exist	Decides the architecture: caches, replicas, queues, regions
Easy to test?	Yes: the feature works or it does not	Needs load tests, monitoring and measurement

Here is a surprising truth: two systems with exactly the same functional requirements can have completely different designs because of non-functional requirements. A to-do app for one family and a to-do app for 100 million users have the same features but are very different systems.

⚠️ COMMON MISTAKE

Beginners jump straight to drawing boxes ("let's use Kafka and Kubernetes!") without asking about requirements. Always clarify first: How many users? How fast? How much data? Can we lose any data? Is a slightly old value acceptable? The answers decide the design.

1.4.3 Constraints and Assumptions

Besides requirements, every design has **constraints**: things you cannot change. Examples: "budget is 2 lakh rupees per month", "data of Indian users must stay in India", "the team knows Java, not Go", "we must launch before Diwali". Write your **assumptions** down too, like "we assume 10% of users are active at the same time". If an assumption turns out wrong, you know which part of the design to revisit.

✏️ ACTIVITY 1.1

Pick an app you use every day (for example a UPI payment app, a cab app or a music app). Write five functional requirements and five non-functional requirements for it. For each NFR, try to include a number (time, percentage or users).

✅ CHECK YOUR PROGRESS 1.4

1. Classify: "A user can reset their password by email."

Answer: Functional requirement, because it describes a feature the system offers.

2. Classify: "The password reset email must arrive within 30 seconds."

Answer: Non-functional requirement, because it describes how well (how fast) the feature must work.

3. Why do non-functional requirements shape architecture more than functional ones?

Answer: Because the same features can be built on one server or on thousands; it is the scale, speed, uptime and safety targets that force choices like caching, replication, sharding and multiple regions.

1.5 Key Qualities of a System

Engineers use a set of standard words for non-functional qualities. You will meet these words in every chapter, so let us learn them carefully.

1.5.1 Scalability

Scalability is the ability of a system to handle more load by adding resources. Load can mean more users, more requests per second, more data or more complex work.

- **Vertical scaling (scale up):** buy a bigger machine: more CPU, more RAM, faster disk. Simple, but there is a ceiling (the biggest machine money can buy), it gets expensive fast, and it is still one machine that can fail.
- **Horizontal scaling (scale out):** add more machines and share the work. Harder (you need load balancing, and data must be shared or split), but almost no ceiling, and if one machine dies the others continue.

Analogy: Raju can buy a giant industrial stove (vertical) or set up five normal stoves with five helpers (horizontal). The giant stove has a limit and, if it breaks, everything stops. Five stoves can grow to ten.

Point	Vertical scaling	Horizontal scaling
How	Bigger machine	More machines
Code changes needed	Usually none	Often yes (stateless servers, sharding)
Upper limit	Hard limit of the largest machine	Very high
Failure	Single point of failure remains	Can survive individual machine failures
Cost curve	Gets very expensive at the top end	Grows roughly linearly with commodity machines
Good for	Early stage, databases that are hard to split	Web/app servers, large-scale systems

1.5.2 Availability

Availability is the percentage of time the system is up and able to serve requests. If a system was usable for 99.9% of the minutes in a year, its availability is 99.9%. Availability is usually written as "nines" (see Section 1.7).

Formula: **Availability** = $\text{Uptime} \div (\text{Uptime} + \text{Downtime})$.

An ATM that works 23 hours a day and is "under maintenance" for 1 hour has about 95.8% availability. Customers who come during that hour do not care that the ATM was perfect the other 23 hours.

1.5.3 Reliability

Reliability is the ability of a system to keep working *correctly* over time, even when things go wrong (hardware faults, software bugs, human mistakes). A reliable system gives the right answer,

every time, and recovers from faults.

Availability and reliability are related but different. A system can be "available" (it responds) but unreliable (it responds with wrong data). Imagine a railway enquiry counter that is always open but sometimes tells you the wrong platform. Available, yes. Reliable, no. Two common measures are **MTBF** (mean time between failures: how long it usually runs before failing) and **MTTR** (mean time to repair: how long it takes to fix). High MTBF and low MTTR give high availability.

1.5.4 Latency

Latency is the time taken to complete one request, from the moment it is sent to the moment the response arrives. It is measured in milliseconds (ms). Lower is better.

We rarely talk about "average" latency alone, because averages hide pain. Instead we use **percentiles**:

- **p50 (median)**: half the requests are faster than this.
- **p95**: 95% of requests are faster than this.
- **p99**: 99% of requests are faster than this; the slowest 1% are slower.

If p50 is 80 ms but p99 is 3 seconds, then one in every hundred users waits 3 seconds or more. On a page that makes many backend calls, one slow call can slow the whole page, so the slow "tail" matters a lot.

1.5.5 Throughput

Throughput is how much work the system completes per unit of time: requests per second (RPS or QPS), messages per second, or megabytes per second.

Latency vs throughput: think of a Mumbai local train. Latency is how long one passenger takes to go from Churchgate to Borivali. Throughput is how many passengers the line carries per hour. You can increase throughput by running more trains (or longer ones) without making any single journey faster. You can also make one journey faster (fast train) without carrying more people. They are related but not the same.

1.5.6 Consistency

Consistency means all users see the same, correct, up-to-date data. When data is copied to several machines, a change on one copy takes some time to reach the others. During that time, different users may see different values.

- **Strong consistency**: after a write finishes, every read sees it. Needed for bank balances and seat booking.
- **Eventual consistency**: after a write, copies will become the same *eventually* (usually within milliseconds or seconds). Fine for like counts, view counts and news feeds.

If your friend's Instagram post shows 1,203 likes on your phone and 1,201 on hers for a second, nobody is hurt. If IRCTC sells the same Tatkal berth to two people, that is a big problem. We study this deeply in later chapters on replication and the CAP theorem.

1.5.7 Durability

Durability means that once data is saved (committed), it will not be lost, even if machines crash or a disk fails. Systems achieve durability by writing to disk, keeping multiple copies (replication), and taking backups in different locations.

Durability is different from availability. A system may be down for an hour (not available) but come back with all data intact (durable). The reverse, being up but having lost your data, is usually much worse.

💡 DO YOU KNOW?

Amazon S3, launched in 2006, is designed for "eleven nines" (99.999999999%) of durability for stored objects, according to AWS documentation. It achieves this by storing data redundantly across multiple devices and facilities. Note that this durability number is separate from S3's availability target, which is lower.

1.5.8 Maintainability

Maintainability is how easy it is for engineers to understand, operate, fix and change the system over time. Most of a system's cost comes after launch, during years of maintenance. Signs of a maintainable system:

- **Simplicity:** new engineers understand it quickly.
- **Operability:** good logs, metrics, alerts and dashboards, so problems are seen early.
- **Evolvability:** new features can be added without breaking old ones.

1.5.9 Cost

Every design decision has a price: servers, storage, network bandwidth, third-party services, and engineering time. A design that is perfect but costs more than the business earns is a bad design. Good engineers ask "is this extra nine of availability worth its cost?" Often a design that is 95% as good at 30% of the cost is the right answer.

Quality	Simple question it answers	Typical measure	Chai stall version
Scalability	Can it grow?	Max users or QPS supported	Can Raju serve 800 people?
Availability	Is it up?	% uptime (nines)	Is the stall open when I arrive?
Reliability	Does it work correctly over time?	Error rate, MTBF	Do I always get the tea I ordered?
Latency	How fast is one request?	ms at p50, p99	How long do I wait for my cup?
Throughput	How much work per second?	QPS, MB/s	Cups served per hour
Consistency	Does everyone see the same truth?	Staleness, guarantees	Does the price board match what the helper charges?

Quality	Simple question it answers	Typical measure	Chai stall version
Durability	Is saved data safe?	Probability of data loss	Is the credit notebook safe from rain?
Maintainability	Is it easy to run and change?	Time to fix or ship changes	Can a new helper learn the routine in a day?
Cost	Can we afford it?	Rupees per month, per user	Is milk and gas cheaper than what customers pay?

✓ CHECK YOUR PROGRESS 1.5

1. What is the difference between latency and throughput?

Answer: Latency is the time for one request to complete. Throughput is the number of requests (or amount of work) completed per second. A system can have high throughput but also high latency, or low latency but low throughput.

2. Why do engineers look at p99 latency instead of only the average?

Answer: Averages hide the slow requests. p99 shows the experience of the slowest 1% of requests, which often affects many users, especially when a page makes many backend calls.

3. A system is down for one hour but loses no data. Which quality was hurt, availability or durability?

Answer: Availability was hurt. Durability was fine, because no saved data was lost.

4. Which consistency model would you choose for a like counter, and why?

Answer: Eventual consistency. A like count being slightly behind for a second harms no one, and eventual consistency is cheaper and faster at scale.

1.6 Single Points of Failure and Bottlenecks

1.6.1 Single Point of Failure (SPOF)

A **single point of failure** is any one component whose failure brings down the whole system. In our one-server monolith, the server itself is a SPOF. So is the database, the network switch it is connected to, the power supply, and even the one engineer who knows the root password.

How to remove a SPOF: add **redundancy**, which means having a spare that can take over.

- Two or more app servers behind a load balancer.
- A database replica that can be promoted if the primary fails.
- Two load balancers (active-passive) so the load balancer itself is not a SPOF.
- Servers in more than one data centre or availability zone.
- Documentation and on-call rotations, so not just one person can fix things.

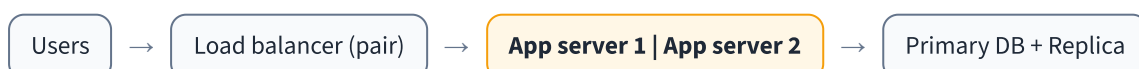


Figure 1.4: Adding redundancy at every layer to remove single points of failure.

Analogy: A wedding with only one cook is a SPOF. If the cook gets food poisoning (ironic!), there is no dinner. Smart families hire a caterer with a team, and keep a backup halwai's number handy.

1.6.2 Bottlenecks

A **bottleneck** is the slowest part of a system that limits the overall throughput. The name comes from a bottle: no matter how big the bottle is, liquid can only pour out as fast as the narrow neck allows.

Common bottlenecks:

- **Database:** the most common one. Many app servers can be added easily, but they all hit the same database.
- **CPU:** heavy computation like image resizing or encryption.
- **Disk I/O:** too many reads and writes to a slow disk.
- **Network bandwidth:** sending large videos or files.
- **A single lock or counter:** every request waiting for the same resource.
- **An external API:** a payment gateway that only accepts a certain number of calls per second.

A key rule: **improving anything other than the bottleneck does not help much.** If the database can handle only 2,000 queries per second, adding ten more app servers will not raise throughput above roughly 2,000 queries per second. You must fix the database (add caching, replicas, indexes or sharding).

Fixing one bottleneck usually reveals the next one. This is normal. Scaling is a journey of moving the bottleneck to a place where it hurts less.

FUNNY SIDE

Every Indian office has a human bottleneck: the one manager who must approve everything. Ten people can prepare documents in parallel, but they all wait at one desk for one signature. Adding more employees does not help. Adding a second approver does.

ACTIVITY 1.2

Draw the path of a UPI payment from your phone to the shopkeeper's bank account as boxes and arrows (phone app, UPI app server, NPCI, your bank, shopkeeper's bank). Circle every box that could be a single point of failure if it had no backup. Then mark which box you think could become a bottleneck on a big sale day.

✓ CHECK YOUR PROGRESS 1.6

1. Define a single point of failure.

Answer: A component whose failure alone causes the whole system to stop working, because there is no backup to take over.

2. Your site is slow. CPU on app servers is 20%, but the database CPU is 98%. What is the bottleneck, and will adding more app servers help?

Answer: The database is the bottleneck. Adding app servers will not help, and might even make it worse by sending more queries. You should reduce database load with caching, indexes, read replicas or sharding.

3. What is the general technique for removing a SPOF?

Answer: Redundancy: run two or more copies of the component so that if one fails, another takes over automatically.

1.7 Availability and the "Nines"

Availability targets are written as a number of nines. "Three nines" means 99.9%. "Five nines" means 99.999%. Each extra nine reduces allowed downtime by ten times, and usually costs much more to achieve.

Availability	Name	Downtime per year	Downtime per month (30 days)	Downtime per week
90%	One nine	36.5 days	3 days	16.8 hours
99%	Two nines	3.65 days	7.2 hours	1.68 hours
99.5%	Two and a half nines	1.83 days	3.6 hours	50.4 minutes
99.9%	Three nines	8.76 hours	43.2 minutes	10.1 minutes
99.95%	Three and a half nines	4.38 hours	21.6 minutes	5.04 minutes
99.99%	Four nines	52.6 minutes	4.32 minutes	1.01 minutes
99.999%	Five nines	5.26 minutes	25.9 seconds	6.05 seconds

Table 1.1: The nines table (year taken as 365 days).

1.7.1 How to Calculate Downtime

Allowed downtime = $(1 - \text{availability}) \times \text{time period}$.

Example: 99.9% per year. $1 - 0.999 = 0.001$. One year = $365 \times 24 = 8,760$ hours. $0.001 \times 8,760 = 8.76$ hours.

Example: 99.99% per month of 30 days. $0.0001 \times (30 \times 24 \times 60) = 0.0001 \times 43,200 \text{ minutes} = 4.32$ minutes.

1.7.2 Availability in Series and in Parallel

Real systems are chains of components. How do we combine their availability?

In series (a request needs all of them): multiply. If the app server is 99.9% available and the database is 99.9% available, and a request needs both:

$$A_{\text{total}} = 0.999 \times 0.999 = 0.998 \quad (\text{about } 99.8\%)$$

The chain is *weaker* than each part. More components in a chain means lower availability.

In parallel (any one of them is enough): the system fails only if all copies fail at the same time. If each of two app servers is 99.9% available and they fail independently:

$$\begin{aligned} A_{\text{total}} &= 1 - (1 - 0.999) \times (1 - 0.999) \\ &= 1 - 0.001 \times 0.001 \\ &= 1 - 0.000001 \\ &= 0.999999 \quad (\text{about } 99.9999\%) \end{aligned}$$

Redundancy makes the system *stronger* than each part. (In real life, failures are not fully independent: both servers may share a power supply or a buggy release, so the true number is lower. Still, the idea holds.)

1.7.3 SLA, SLO and SLI

- **SLI (Service Level Indicator):** the actual measurement, for example "percentage of requests that succeeded in under 300 ms".
- **SLO (Service Level Objective):** the internal target for an SLI, for example "99.9% of requests succeed in under 300 ms over 30 days".
- **SLA (Service Level Agreement):** a promise to customers, often with a penalty (like service credits) if broken. SLAs are usually a little looser than SLOs, to keep a safety margin.

The gap between 100% and the SLO is called the **error budget**. If your SLO is 99.9%, you are allowed 0.1% of bad minutes. If you have budget left, you can ship features faster; if you have used it up, you slow down and focus on stability.

💡 DO YOU KNOW?

The idea of the "error budget" was popularised by Google's Site Reliability Engineering (SRE) book, published in 2016. Google argues that 100% is the wrong reliability target for almost everything, because users cannot tell the difference between very high availability and 100%, while the cost of chasing perfection is enormous.

🤔 THINK AND DISCUSS

Your phone's mobile network is maybe 99% reliable where you live. If a website is 99.999% available but the user's network is 99%, what availability does the user actually experience? Is it worth spending crores to go from four nines to five nines for a consumer app?

✓ CHECK YOUR PROGRESS 1.7

1. How much downtime per year does 99.99% availability allow?

Answer: About 52.6 minutes per year ($0.0001 \times 525,600$ minutes).

2. A request passes through a load balancer (99.99%), an app server (99.9%) and a database (99.9%). What is the approximate availability of the chain?

Answer: Multiply them: $0.9999 \times 0.999 \times 0.999 \approx 0.9979$, so about 99.79%.

3. What is an error budget?

Answer: The amount of unreliability allowed by the SLO, that is 100% minus the SLO. For a 99.9% SLO, the error budget is 0.1% of requests or time, which the team can "spend" on risky changes.

1.8 The Trade-off Mindset

If you remember only one sentence from this chapter, let it be this: **in system design, there are no perfect solutions, only trade-offs.** Every choice makes something better and something else worse. The skill is choosing what to sacrifice, and saying it out loud.

1.8.1 Common Trade-offs

Trade-off	You gain	You pay	Example
Consistency vs availability	Always-correct data	May refuse requests during network problems	Bank transfer vs social media likes
Latency vs consistency	Fast reads from a nearby copy	Data may be slightly stale	Reading a news feed from a cache
Performance vs cost	Faster responses	More servers, more RAM, more money	Keeping everything in memory
Simplicity vs scalability	Easy to build and debug	Harder to grow later	Monolith vs microservices
Read speed vs write speed	Fast reads (indexes, precomputed feeds)	Slower writes, more storage	Adding many database indexes
Durability vs latency	Data safely on several disks	Each write waits for copies	Synchronous replication
Build vs buy	Full control	Engineering time and maintenance	Running your own database vs a managed cloud database

1.8.2 How to Reason About a Trade-off

1. **Name the options.** "We can cache user profiles or always read from the database."

2. **Name what each option gives and costs.** "Caching is faster and reduces DB load, but profiles may be up to 60 seconds old."
3. **Check against requirements.** "Is 60-second-old profile data acceptable? Yes, for display names."
4. **Decide and write it down.** "We cache with a 60-second expiry, and invalidate on profile update."
5. **Know the escape route.** "If staleness causes complaints, we can reduce expiry or switch to write-through caching."

Analogy: Choosing between the Mumbai local and a taxi. The local is cheap and avoids traffic, but it is crowded and fixed to a timetable. The taxi is comfortable and door-to-door, but expensive and stuck in traffic during rush hour. Neither is "best". It depends on your time, budget and luggage. Systems are the same.

KEY POINT

When someone asks "which is better, SQL or NoSQL?" the correct first answer is "it depends on the requirements". Then you explain what it depends on. That is the system design mindset.

✓ CHECK YOUR PROGRESS 1.8

1. What does it mean to say "system design is about trade-offs"?

Answer: Every design choice improves some qualities and worsens others (for example speed vs cost, or consistency vs availability). There is no perfect choice; the designer picks the option that best fits the requirements and states the downside clearly.

2. Give one trade-off involved in adding a cache.

Answer: A cache makes reads faster and reduces database load, but data in the cache may be stale, and it adds memory cost and complexity (invalidation).

1.9 How Systems Grow: From One Server to Millions of Users

Let us follow an imaginary startup called **ChaiCart**, an app that delivers tea and snacks to offices. This story shows, step by step, how real systems evolve. Each step solves the most painful problem of that moment. You will study every component in detail in later chapters.

1.9.1 Stage 1: One Server (about 100 users)

Two founders write a monolith and run it, with its database, on one cloud server. Cost is low, deployment is one command. Life is good.



Figure 1.5: Stage 1.

1.9.2 Stage 2: Separate the Database (about 1,000 users)

Problem: the app and the database fight over CPU and memory. **Fix:** move the database to its own server. Now each can be sized and tuned separately. A managed database service also gives

automatic backups.



Figure 1.6: Stage 2: app and database on separate machines.

1.9.3 Stage 3: Load Balancer and Multiple App Servers (about 10,000 users)

Problem: one app server cannot handle lunchtime traffic, and every deployment causes downtime. **Fix:** put a **load balancer** in front of two or more app servers. Make app servers **stateless** (store sessions in a shared place, not in server memory), so any server can handle any request. Now you can deploy one server at a time without downtime.



Figure 1.7: Stage 3: horizontal scaling of the app tier.

1.9.4 Stage 4: Cache and CDN (about 100,000 users)

Problem: the database is now the bottleneck; most queries read the same menu and the same popular items. Images load slowly for users far away. **Fix:** add a **cache** (like Redis) for frequently read data, and a **CDN** (content delivery network) to serve images, JavaScript and CSS from servers close to users.

1.9.5 Stage 5: Database Replication (about 500,000 users)

Problem: even with a cache, read traffic is heavy, and if the database dies, everything stops. **Fix:** add **read replicas**. Writes go to the primary; reads can go to replicas. If the primary fails, a replica can be promoted. The database is no longer a SPOF.



Figure 1.8: Stage 5: primary-replica database setup.

1.9.6 Stage 6: Queues and Async Work (about 1 million users)

Problem: placing an order is slow because the app sends an SMS, an email, updates analytics and notifies the shop, all while the user waits. **Fix:** put slow, non-urgent tasks into a **message queue**. The order is saved quickly, and background **workers** process the queue. Users get a fast response; spikes are absorbed by the queue.

1.9.7 Stage 7: Sharding, Services and Multiple Regions (10 million users and beyond)

Problem: the primary database cannot handle all writes; the codebase is so big that 100 engineers step on each other's toes; users in other countries see high latency. **Fixes:**

- **Sharding:** split data across many databases (for example by city or user ID).
- **Microservices:** split the monolith into services owned by separate teams (orders, payments, delivery, search).
- **Multiple data centres or regions:** serve users from nearby, and survive the loss of a whole region.

- **Observability:** central logging, metrics, tracing and alerts, because you can no longer "just SSH and check".

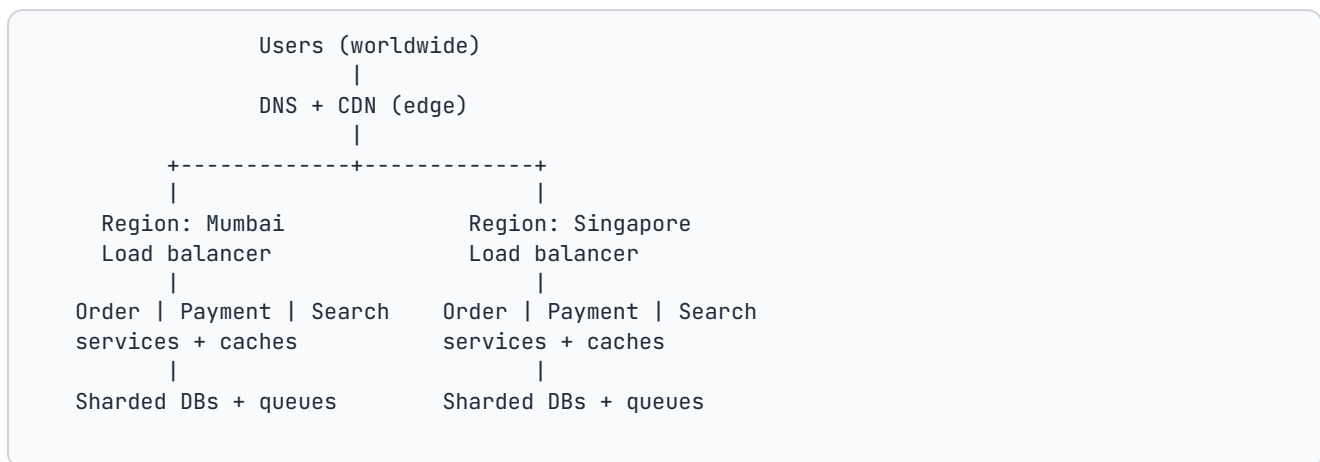


Figure 1.9: Stage 7: a large-scale, multi-region architecture (simplified).

Stage	Approx. users	Main pain	Main fix
1	100	None yet	Single server monolith
2	1,000	App and DB compete for resources	Separate DB server
3	10,000	App server overloaded, downtime on deploy	Load balancer, stateless app servers
4	100,000	Repeated DB reads, slow static files	Cache, CDN
5	500,000	DB reads heavy, DB is SPOF	Read replicas, failover
6	1 million	Slow requests doing too much	Message queues, workers
7	10 million+	Write limits, team size, global latency	Sharding, microservices, multi-region

The user numbers above are only rough guides. A video app may hit stage 4 problems with far fewer users than a text app. The order of steps can also change. What matters is the pattern: **find the current bottleneck, fix it in the simplest way, repeat.**

💡 DO YOU KNOW?

Netflix began moving from its own data centres to Amazon Web Services after a major database corruption in 2008 stopped it from shipping DVDs for about three days. The migration took around seven years and was completed in early 2016, as described on the Netflix technology blog.

💡 DO YOU KNOW?

When Facebook announced it would buy WhatsApp in 2014, WhatsApp served about 450 million monthly users with roughly 50 to 60 employees, of whom only a few dozen were engineers. Its servers were built on Erlang, a language designed by Ericsson for telephone switches that must handle huge numbers of connections and stay up.

⚠ COMMON MISTAKE

Jumping to Stage 7 on day one. A five-person startup running 30 microservices, Kubernetes and three databases spends all its time on infrastructure instead of customers. Design for the next stage, not the final stage.

✓ CHECK YOUR PROGRESS 1.9**1. Why must app servers be stateless before you put them behind a load balancer?**

Answer: Because the load balancer may send a user's next request to a different server. If session data lives in one server's memory, the user would appear logged out. Storing state in a shared store (like Redis or the database) lets any server handle any request.

2. What problem do message queues solve in Stage 6?

Answer: They move slow, non-urgent work (emails, SMS, analytics) out of the user's request path. The user gets a fast response, and workers process the tasks in the background, which also absorbs traffic spikes.

3. What is the repeated pattern in the growth story?

Answer: Find the current bottleneck or single point of failure, fix it in the simplest way that works, and repeat as traffic grows.

1.10 Who Does System Design?

System design is not only the job of one "architect" sitting in an ivory tower. Many people take part.

Role	System design work
Software engineer (junior)	Designs small features and APIs, writes LLD for modules, learns from design reviews.
Senior engineer	Designs services end to end, chooses data stores, writes design documents, reviews others' designs.
Staff / principal engineer	Designs across many teams, sets technical direction, solves the hardest scaling problems.
Software / solutions architect	Defines overall architecture and standards, often maps business needs to technology.
Site Reliability Engineer (SRE) / DevOps	Designs for availability, monitoring, deployment, capacity planning and incident response.
Engineering manager / tech lead	Balances design quality with deadlines, team skills and cost.
Product manager	Provides functional requirements and business priorities; helps decide trade-offs.
Security engineer	Reviews designs for threats: authentication, encryption, data leaks.

1.10.1 Design Documents and Reviews

In many companies, before building a significant feature, an engineer writes a **design document** (also called an RFC or tech spec). A typical design doc has: background and goals, non-goals, requirements, proposed design with diagrams, alternatives considered, trade-offs, risks, rollout plan and monitoring. Other engineers review it and ask hard questions. It is much cheaper to find a flaw in a document than in production.

1.10.2 System Design Interviews

Many tech companies test system design in interviews for mid-level and senior roles. A common structure for a 45 to 60 minute round:

1. Clarify requirements (functional and non-functional) and scope.
2. Estimate scale: users, QPS, storage (Chapter 2 teaches this).
3. Define APIs and the data model.
4. Draw the high-level design.
5. Deep dive into one or two components (database choice, caching, scaling).
6. Discuss bottlenecks, failures and trade-offs.

Interviewers care less about a "right answer" and more about clear thinking, asking good questions and explaining trade-offs.

✓ CHECK YOUR PROGRESS 1.10

1. Name three sections commonly found in a design document.

Answer: Any three of: goals and non-goals, requirements, proposed design with diagrams, alternatives considered, trade-offs, risks, rollout plan, monitoring.

2. What is the first step in a system design interview?

Answer: Clarifying requirements and scope: asking what features are needed, how many users, and what non-functional goals (latency, availability, consistency) matter.

1.11 How to Study This Book

System design can feel huge. There are hundreds of tools and thousands of blog posts. This book is arranged so that each chapter builds on the previous ones.

1.11.1 The Learning Path

1. **Foundations** (Chapters 1 to 3): vocabulary, estimation and networking. Do not skip these; everything else stands on them.
2. **Building blocks:** load balancers, caches, databases, replication, sharding, queues, storage, CDNs and so on. Each chapter covers one building block deeply.
3. **Distributed systems ideas:** consistency, consensus, CAP, failures and how to handle them.

4. **Case studies:** designing real systems step by step (URL shortener, chat app, news feed, video streaming, ride sharing and more).

1.11.2 Study Tips

- **Draw everything.** Take a notebook and draw every flow diagram yourself. Drawing forces understanding.
- **Ask "why" three times.** Why a cache? To reduce DB load. Why is DB load a problem? It is the bottleneck. Why is it the bottleneck? Because it is a single machine handling all reads.
- **Do the Check Your Progress questions** before reading the answers. Cover the answer with your hand if you must.
- **Do the activities.** Running a command like `ping` or `nslookup` teaches more than reading about it.
- **Relate to apps you use.** When you open Swiggy, Instagram or IRCTC, ask: what is happening behind the screen right now?
- **Read engineering blogs.** Companies like Netflix, Uber, Discord, Cloudflare and Meta publish detailed posts about real problems. After a few chapters, you will understand them.
- **Explain to a friend.** If you can explain consistent hashing to a friend over chai, you have learnt it.
- **Revise with the summary and key terms** at the end of each chapter.

😄 FUNNY SIDE

System design learning has three stages. Stage 1: "What is a load balancer?" Stage 2: "Let's add a load balancer, Kafka, Redis and a service mesh to my to-do app!" Stage 3: "It depends." Congratulations in advance on reaching Stage 3.

🤔 THINK AND DISCUSS

Think of a time an app failed you: a payment stuck, a booking site crashing at 10 a.m. Tatkal time, a video buffering. Which quality from Section 1.5 was missing? What might the engineers have done differently?

✅ CHECK YOUR PROGRESS 1.11

1. Why should you not skip the foundation chapters?

Answer: Because later chapters use their vocabulary, estimation skills and networking knowledge all the time. Without them, building blocks and case studies will be hard to follow.

2. Name two study habits recommended in this section.

Answer: Any two of: draw every diagram yourself, ask "why" repeatedly, attempt questions before reading answers, do the activities, relate concepts to apps you use, read engineering blogs, explain to a friend.

What Have We Learnt?

- System design is deciding the components of a system, their roles and how they communicate, to meet both feature and quality goals.
- High-level design (HLD) shows the big boxes and data flow; low-level design (LLD) shows classes, schemas and APIs inside one box.
- Most systems follow the client-server model: clients send requests, servers send responses.
- Products usually start as a single-server monolith, which is simple and cheap, but becomes a bottleneck and a single point of failure as they grow.
- Functional requirements describe what the system does; non-functional requirements describe how well it does it, and they drive architecture.
- Key qualities: scalability, availability, reliability, latency, throughput, consistency, durability, maintainability and cost.
- Vertical scaling means a bigger machine; horizontal scaling means more machines.
- Latency is measured with percentiles like p50, p95 and p99, not only averages.
- A single point of failure is removed by redundancy; a bottleneck limits the whole system's throughput.
- Each extra "nine" of availability cuts allowed downtime by ten times; 99.9% allows about 8.76 hours per year.
- Availability multiplies in series (weaker) and improves in parallel (stronger).
- SLIs measure, SLOs set targets, SLAs make promises; the error budget is 100% minus the SLO.
- System design is all about trade-offs: name the options, their costs, check against requirements and decide.
- Systems grow step by step: separate DB, load balancer, cache and CDN, replicas, queues, then sharding, services and multiple regions.

Key terms: system design, high-level design (HLD), low-level design (LLD), client, server, client-server model, peer-to-peer, monolith, functional requirement, non-functional requirement, constraint, scalability, vertical scaling, horizontal scaling, availability, reliability, MTBF, MTTR, latency, percentile (p50, p95, p99), throughput, QPS, consistency, strong consistency, eventual consistency, durability, maintainability, single point of failure (SPOF), redundancy, bottleneck, nines, SLI, SLO, SLA, error budget, trade-off, load balancer, stateless, cache, CDN, read replica, message queue, sharding, microservices, design document.

Exercises

A. Multiple Choice Questions

- Which of the following is a low-level design decision?
 - Using a CDN for images
 - Choosing NoSQL over SQL
 - Defining the columns and indexes of the orders table
 - Deploying in two regions
- "The app must respond within 200 ms for 99% of requests" is a:
 - Functional requirement
 - Non-functional requirement
 - Constraint on budget
 - Design pattern
- Adding more RAM and CPU to the same server is called:
 - Horizontal scaling
 - Sharding
 - Vertical scaling
 - Replication
- How much downtime per year does 99.9% availability allow, approximately?
 - 52.6 minutes
 - 8.76 hours
 - 3.65 days
 - 5.26 minutes
- Two components each with 99% availability are placed in series. The total availability is about:
 - 99.99%
 - 99%
 - 98.01%
 - 100%
- Which quality ensures that saved data is not lost even if a disk fails?
 - Latency
 - Durability
 - Throughput
 - Maintainability
- A like counter that shows slightly different numbers on two phones for a second is an example of:
 - Strong consistency
 - Eventual consistency
 - Data loss
 - Vertical scaling
- The database is at 99% CPU while app servers are at 15%. Adding more app servers will:
 - Greatly increase throughput
 - Not help much, because the database is the bottleneck
 - Remove the need for a database
 - Make the database faster
- Which metric tells you about the slowest 1% of requests?
 - p50
 - Average
 - p99
 - Throughput

10. The error budget for an SLO of 99.95% is:

- | | |
|---------|-----------|
| a. 0.5% | b. 0.05% |
| c. 5% | d. 99.95% |

B. Fill in the Blanks

1. A single application that contains all features and is deployed as one unit is called a _____.
2. The time taken to complete one request is called _____, while the number of requests completed per second is called _____.
3. A component whose failure brings down the whole system is called a _____.
4. The internal target for a service level indicator is called an _____.
5. Placing two independent servers in parallel _____ the overall availability.
6. A _____ serves static files like images from servers close to the user.

C. True or False

1. Every startup should begin with microservices to be ready for scale.
2. A system can be available but not reliable.
3. Vertical scaling has no upper limit.
4. Adding components in series reduces overall availability.
5. Durability and availability mean the same thing.

D. Short Answer Questions

1. Differentiate between HLD and LLD with one example of each.
2. Why are non-functional requirements important in system design?
3. What does "stateless app server" mean, and why does it help scaling?
4. Explain the difference between availability and durability with an example.
5. What is a bottleneck? Name three common bottlenecks.
6. What is the difference between an SLO and an SLA?

E. Long Answer Questions

1. Explain the nine key qualities of a system (scalability, availability, reliability, latency, throughput, consistency, durability, maintainability, cost) with an everyday example for each.
2. Describe how a startup's architecture evolves from one server to a multi-region system. For each stage, state the problem and the fix.
3. Compare vertical and horizontal scaling. When would you choose each?
4. "System design is about trade-offs." Explain this statement with at least four examples of trade-offs.

F. Design Challenge

Your college wants an online system for students to register for elective courses. About 5,000 students register in a single 2-hour window at the start of each semester, and every course has limited seats. Write: (1) four functional and four non-functional requirements, (2) a simple high-level design with boxes and arrows, (3) the single points of failure in your design and how you would remove them, and (4) one important trade-off you made.

Answer Key: Chapter 1

A. MCQ

1. (c) Table columns and indexes are inside-the-box details.
2. (b) It describes how fast, not what feature.
3. (c) A bigger machine is vertical scaling.
4. (b) $0.001 \times 8,760 \text{ hours} = 8.76 \text{ hours}$.
5. (c) $0.99 \times 0.99 = 0.9801$.
6. (b) Durability is about not losing saved data.
7. (b) Copies agree eventually.
8. (b) Throughput is limited by the bottleneck.
9. (c) p99 marks the boundary of the slowest 1%.
10. (b) $100\% - 99.95\% = 0.05\%$.

B. Fill in the Blanks

1. monolith
2. latency; throughput
3. single point of failure (SPOF)
4. SLO (Service Level Objective)
5. increases
6. CDN (content delivery network)

C. True or False

1. False. Most startups should begin with a simple monolith and split it only when there is a real need.
2. True. It may respond but with wrong or corrupted results.
3. False. You are limited by the largest machine available, and cost rises steeply near the top.
4. True. Availabilities multiply, so the product is lower than each part.
5. False. Availability is about being up; durability is about not losing saved data.

D. Short Answers

1. HLD describes the overall architecture: major components and how they connect, for example "app servers behind a load balancer, with Redis cache and PostgreSQL". LLD describes the internals of one component, for example the `Order` class with its methods or the exact `orders` table schema.
2. Non-functional requirements (speed, uptime, scale, durability, cost) decide the architecture. The same features can be built on one server or across thousands; NFRs tell you which one you need, and whether you need caches, replicas, queues or multiple regions.
3. A stateless app server does not keep user-specific data (like sessions) in its own memory between requests; such state lives in a shared store. This lets a load balancer send any request to any server, makes it easy to add or remove servers, and means a crashed server loses nothing important.
4. Availability is the fraction of time the system is usable; durability is the guarantee that committed data is never lost. A bank website down for maintenance for one hour has lower availability but

full durability if no transactions are lost. A system that stays up but loses yesterday's orders has a durability failure.

5. A bottleneck is the slowest component that limits the throughput of the whole system, like the narrow neck of a bottle. Common bottlenecks: the database, CPU-heavy processing, disk I/O, network bandwidth, a shared lock, or a rate-limited external API.
6. An SLO is an internal target for a measured indicator (for example 99.9% of requests under 300 ms). An SLA is a formal promise to customers, usually with penalties such as refunds or credits if broken. SLAs are usually set a bit looser than SLOs for safety.

E. Long Answers

1. Model answer points:

- Scalability: handling more load by adding resources; a chai stall adding stoves and helpers.
- Availability: percentage of time the system is usable; an ATM that is open 24 hours vs one often "under maintenance".
- Reliability: correct operation over time despite faults; an enquiry counter that always gives the right platform.
- Latency: time for one request; how long you wait for your tea.
- Throughput: work done per unit time; cups served per hour or passengers carried per hour by local trains.
- Consistency: everyone sees the same correct data; IRCTC must never sell one berth twice, while likes can be eventually consistent.
- Durability: saved data survives failures; a credit notebook kept safe with a photocopy elsewhere.
- Maintainability: ease of understanding, operating and changing; a new helper can learn the routine quickly.
- Cost: money and effort to run; the design must cost less than the value it creates.

2. Model answer points:

- Stage 1: single server with app and DB; simple and cheap.
- Stage 2: app and DB compete for resources; move DB to a separate server.
- Stage 3: app server overloaded and deployments cause downtime; add load balancer and multiple stateless app servers.
- Stage 4: repeated reads overload DB and static files load slowly; add a cache and a CDN.
- Stage 5: DB reads still heavy and DB is a SPOF; add read replicas and automatic failover.
- Stage 6: requests do too much slow work; add message queues and background workers.
- Stage 7: write limits, large teams and global users; shard the DB, split into microservices, deploy in multiple regions, add observability.
- The general pattern: find the current bottleneck, fix it simply, repeat.

3. Model answer points:

- Vertical scaling: bigger machine; no code changes; simple.
- Limits of vertical: hard ceiling, steep cost, still a single point of failure, upgrades may need downtime.

- Horizontal scaling: add more machines; almost unlimited growth; survives single machine failures.
- Costs of horizontal: needs load balancing, stateless design, data partitioning, more operational complexity.
- Choose vertical early on, or for components that are hard to split (a relational database at moderate scale).
- Choose horizontal for stateless web/app tiers and for large-scale systems needing high availability.
- In practice, companies use both: scale up until it is costly, then scale out.

4. Model answer points:

- No design is perfect; each choice improves some qualities and worsens others.
- Consistency vs availability: banks prefer correctness; social apps prefer staying up.
- Latency vs consistency: caches are fast but may be stale.
- Performance vs cost: more servers and RAM cost more money.
- Simplicity vs scalability: monoliths are easy now, microservices scale teams later.
- Read vs write speed: indexes speed up reads but slow writes.
- Durability vs latency: waiting for multiple copies makes writes slower.
- A good designer names options, costs, checks requirements, decides, and documents the reason and an escape route.

F. Design Challenge: Model Solution

Functional: students log in with college ID; view available electives with remaining seats; register for a course; drop a course before the deadline. Admins can add courses and see registrations. **Non-functional:** no course may be over-booked (strong consistency on seat counts); pages load within 1 second during the 2-hour window; system available 99.9% during registration week; registrations must never be lost (durable). Peak load is modest: 5,000 students in 2 hours, with perhaps most arriving in the first few minutes, so a few hundred requests per second at most.

Design: Students → load balancer → two stateless app servers → one relational database (primary) with a standby replica. Course listing (read-heavy) can be cached for a few seconds; seat booking goes straight to the database inside a transaction that checks and decrements the seat count (for example `UPDATE courses SET seats = seats - 1 WHERE id = ? AND seats > 0`), so two students can never take the last seat.

SPOFs and fixes: a single app server (fixed by running two behind a load balancer); a single database (fixed by a standby replica with automatic failover and regular backups); a single load balancer (use a managed load balancer, which is redundant, or an active-passive pair).

Trade-off: we chose strong consistency for seat counts over maximum speed, so booking requests may queue slightly on the database at peak, but no course is ever over-booked. Seat counts shown on the listing page may be a few seconds stale because of caching; the final check at booking time is always exact.

CHAPTER 2

Estimation and Numbers Every Engineer Should Know

An engineer who cannot estimate is like a cook who cannot guess how much rice to make for 500 wedding guests: someone either goes hungry or eats rice for a week.

After studying this chapter, you will be able to

- Recall the approximate latency of common operations, from an L1 cache read to a cross-continent network round trip.
- Convert confidently between powers of two, powers of ten and data units (KB, MB, GB, TB, PB).
- Use the back-of-the-envelope method to estimate QPS, peak QPS, storage, bandwidth and cache memory.
- Estimate how many servers a system needs.
- Work through full estimates for systems like Twitter, WhatsApp, YouTube and a URL shortener.
- Avoid the most common estimation mistakes.

Every Indian mother is a master of estimation. Ten guests are coming for lunch. "Each person eats about three rotis, kids eat two, Sharma uncle eats six. Keep ten extra, just in case." Within seconds, she knows to make about 40 rotis. She does not need exact numbers. She needs numbers that are *roughly right* and leave a *safety margin*.

System design works the same way. Before you choose a database or decide how many servers to buy, you must know roughly how big the problem is. Is it 10 requests per second or 10 lakh requests per second? Is it 10 gigabytes of data or 10 petabytes? The answer changes the whole design. A 10 GB database fits on one machine; a 10 PB dataset needs a fleet.

In this chapter we learn the numbers every engineer should carry in their head, and a simple step-by-step method for quick estimates, often called **back-of-the-envelope calculation**, because you should be able to do it on the back of an envelope (or a chai stall bill).

2.1 Latency Numbers Every Engineer Should Know

Computers do different things at wildly different speeds. Reading from the CPU's cache is millions of times faster than sending a packet across the world. Knowing these numbers tells you *where*

time goes and what to avoid.

2.1.1 The Classic Table

The table below is based on the famous list popularised by Jeff Dean of Google (originally compiled with Peter Norvig's help). Hardware has improved since, so treat these as **orders of magnitude**, not exact values.

Operation	Approximate time	In human terms (if 1 ns = 1 second)
L1 cache reference	0.5 ns	Half a second: a heartbeat
Branch mispredict	5 ns	5 seconds
L2 cache reference	7 ns	7 seconds
Mutex lock/unlock	25 to 100 ns	About a minute
Main memory (RAM) reference	100 ns	About 1.5 minutes: making a cup of tea
Compress 1 KB with a fast compressor	3 to 10 μ s	An hour or two
Send 1 KB over a 1 Gbps network	10 μ s	About 3 hours
Read 4 KB randomly from SSD	about 150 μ s	About 1.7 days
Read 1 MB sequentially from RAM	about 250 μ s	About 3 days
Round trip within the same data centre	about 500 μ s (0.5 ms)	About 6 days
Read 1 MB sequentially from SSD	about 1 ms	About 12 days
Hard disk (HDD) seek	about 10 ms	About 4 months
Read 1 MB sequentially from HDD	about 20 ms	About 8 months
Packet round trip California → Netherlands → California	about 150 ms	About 5 years

Table 2.1: Latency numbers ($ns = \text{nanosecond} = 10^{-9} s$, $\mu s = \text{microsecond} = 10^{-6} s$, $ms = \text{millisecond} = 10^{-3} s$).

Modern NVMe SSDs and newer RAM are faster than the original numbers (reading 1 MB sequentially from RAM can take tens of microseconds today). But the *ratios* still hold, and that is what matters for design.

2.1.2 Lessons From the Table

- 1. Memory is fast, disk is slow, network is slower.** Reading from RAM is about 100,000 times faster than an HDD seek. That is why caches exist.
- 2. Avoid disk seeks.** HDDs are good at reading long continuous stretches, bad at jumping around. Databases and log systems are designed to write sequentially.
- 3. Sequential beats random.** Even on SSDs, sequential reads are much faster per byte than random ones.

4. **Cross-continent round trips are expensive.** About 150 ms is a hard floor set by the speed of light in fibre. No amount of money makes light faster. To be fast for distant users, you must put servers (or a CDN) near them.
5. **Network inside a data centre is cheap compared with across the world.** A call to another service in the same data centre costs about half a millisecond; it is fine to make a few, but not hundreds in a row.
6. **Compress before sending** over slow links: compressing is cheap compared to sending more bytes across the internet.

💡 DO YOU KNOW?

Light travels at about 300,000 km per second in vacuum, but only about 200,000 km per second inside optical fibre. Mumbai to New York is roughly 12,500 km in a straight line, so even a perfect straight fibre would need about 62 ms one way, or about 125 ms for a round trip. Real cables are not straight, so real round trips are usually higher.

😄 FUNNY SIDE

If reading from L1 cache is picking up the cup of chai in your hand, reading from RAM is walking to the kitchen, reading from SSD is a trip to the local market, an HDD seek is a train journey to your hometown, and a cross-continent round trip is waiting for your cousin in America to reply to "Happy Diwali" on WhatsApp.

✏️ ACTIVITY 2.1

Open a terminal and run `ping google.com` and then `ping` a website hosted far away (for example a university site in another continent). Note the "time=" values. The first is often a few milliseconds to a few tens of milliseconds (a nearby server), the second could be 150 to 300 ms. You have just measured round-trip latency.

✅ CHECK YOUR PROGRESS 2.1

1. Roughly how many times faster is a RAM reference than an HDD seek?

Answer: About 100,000 times (100 ns vs 10 ms = 10,000,000 ns).

2. Why can't money buy a 10 ms round trip between India and the USA?

Answer: Because the speed of light in fibre limits it. The distance alone forces a round trip of over 100 ms. The only fix is to place servers or caches closer to users.

3. What is the approximate round-trip time within a single data centre?

Answer: About 0.5 ms (500 µs).

2.2 Powers of Two, Powers of Ten and Data Units

2.2.1 Data Units

A **bit** is a 0 or 1. A **byte** is 8 bits. One English character in ASCII or simple UTF-8 takes 1 byte. A Hindi character in UTF-8 usually takes 3 bytes. Emojis usually take 4.

Power of two	Exact value	Approx. power of ten	Name	Short
2^{10}	1,024	10^3 (thousand)	Kilobyte	KB
2^{20}	1,048,576	10^6 (million)	Megabyte	MB
2^{30}	1,073,741,824	10^9 (billion)	Gigabyte	GB
2^{40}	about 1.1×10^{12}	10^{12} (trillion)	Terabyte	TB
2^{50}	about 1.13×10^{15}	10^{15} (quadrillion)	Petabyte	PB
2^{60}	about 1.15×10^{18}	10^{18}	Exabyte	EB

Table 2.2: Data units. For estimation, treat 2^{10} as 1,000.

Strictly speaking, 1,024 bytes is a **kibibyte (KiB)** and 1,000 bytes is a kilobyte (KB). Disk manufacturers use powers of ten, operating systems often use powers of two. For estimation the 2.4% difference does not matter, so we round freely.

2.2.2 Indian and International Numbers

Many Indian readers think in lakhs and crores. Here is the bridge:

Indian	International	Power of ten
1 thousand	1 thousand	10^3
1 lakh	100 thousand	10^5
10 lakh	1 million	10^6
1 crore	10 million	10^7
100 crore	1 billion	10^9
1 lakh crore	1 trillion	10^{12}

2.2.3 Time Numbers to Remember

Period	Exact seconds	Estimation value
1 minute	60	60
1 hour	3,600	about 3.6×10^3

Period	Exact seconds	Estimation value
1 day	86,400	about 10^5 (1 lakh)
1 month (30 days)	2,592,000	about 2.5×10^6
1 year	31,536,000	about 3×10^7

The most useful trick in this whole chapter: **one day $\approx$ 100,000 seconds**. So "X per day" $\div$ 100,000 $\approx$ "X per second". The true value is 86,400, so this shortcut underestimates the per-second rate by about 14%. That is acceptable for a first estimate, and you can multiply by 1.16 if you want to be more careful.

2.2.4 Sizes of Common Things

Item	Typical size (rough)
A user ID or timestamp (64-bit integer)	8 bytes
A UUID	16 bytes (36 characters as text)
A short text message or tweet text	100 to 300 bytes
A database row with metadata	200 bytes to 1 KB
A JSON API response	1 to 50 KB
A compressed profile photo / thumbnail	20 to 200 KB
A phone camera photo (compressed)	2 to 5 MB
A minute of compressed audio (128 kbps)	about 1 MB
A minute of 1080p video (about 5 Mbps)	about 40 MB
An hour of 1080p video	about 2 to 3 GB

COMMON MISTAKE

Mixing up bits and bytes. Network speeds are quoted in **bits** per second (Mbps, Gbps). File sizes are in **bytes** (MB, GB). A "100 Mbps" connection downloads at most about 12.5 MB per second, not 100 MB. Lowercase b = bit, uppercase B = byte.

✓ CHECK YOUR PROGRESS 2.2

1. How many bytes are roughly in 1 TB? Write as a power of ten.

Answer: About 10^{12} bytes (one trillion bytes).

2. How many seconds are there in a day, approximately, for estimation?

Answer: About 10^5 (100,000). The exact value is 86,400.

3. A 1 Gbps link: how many megabytes per second can it carry at most?

Answer: 1 Gbps = 1,000 Mbps ÷ 8 = 125 MB per second (in practice slightly less due to protocol overhead).

4. What is 10 crore in international notation?

Answer: 100 million (10^8).

2.3 The Back-of-the-Envelope Method

A back-of-the-envelope estimate is a quick, rough calculation that gets you within the right **order of magnitude** (the right power of ten). If the real answer is 3,000 QPS, an estimate of 2,000 or 5,000 is great. An estimate of 30 or 300,000 is wrong.

2.3.1 Why Estimate?

- To decide if one server is enough or you need a hundred.
- To decide if data fits in memory, on one disk, or needs a distributed store.
- To spot the bottleneck before building anything (is it storage, bandwidth or CPU?).
- To estimate cost.
- In interviews, to show you understand scale and to justify design choices.

2.3.2 The Step-by-Step Method

1. **Write down assumptions clearly.** Daily active users, actions per user, size of each item, read:write ratio, how long data is kept.
2. **Round numbers aggressively.** 86,400 becomes 10^5 ; 1,024 becomes 1,000; 365 becomes about 400 when you want a safe upper bound (or keep 365 when easy).
3. **Use powers of ten.** Multiply the leading digits, add the exponents. $3 \times 10^6 \times 2 \times 10^3 = 6 \times 10^9$.
4. **Compute traffic (QPS)** for reads and writes separately.
5. **Compute peak traffic** by applying a peak factor.
6. **Compute storage** per day, then per year, then for the retention period, then multiply by replication.
7. **Compute bandwidth** (incoming and outgoing).
8. **Compute cache memory** using the 80/20 rule.
9. **Compute number of servers.**

10. **Sanity check.** Does the answer make sense? Compare with things you know.

2.3.3 Powers-of-Ten Arithmetic

Multiply: $(a \times 10^m) \times (b \times 10^n) = (a \times b) \times 10^{(m+n)}$

Divide: $(a \times 10^m) \div (b \times 10^n) = (a \div b) \times 10^{(m-n)}$

Example: 500 million $\times$ 2 KB
 $= 5 \times 10^8 \times 2 \times 10^3$ bytes
 $= 10 \times 10^{11}$ bytes
 $= 10^{12}$ bytes = 1 TB

KEY POINT

The goal is the right order of magnitude and clearly stated assumptions, not decimal precision. A tidy estimate with stated assumptions is more valuable than a precise number nobody can check.

CHECK YOUR PROGRESS 2.3

1. What does "order of magnitude" mean in estimation?

Answer: The power of ten of the answer. Being within the same order of magnitude means your estimate is roughly within a factor of a few (not 10 times off) of the true value.

2. Compute $4 \times 10^7 \times 5 \times 10^2$.

Answer: $20 \times 10^9 = 2 \times 10^{10}$.

3. Why should you write down assumptions first?

Answer: So that anyone (including you later) can check them, and if an assumption changes, you know exactly which numbers to recompute.

2.4 Traffic: From DAU to QPS and Peak QPS

2.4.1 DAU and MAU

MAU (monthly active users) counts unique users who used the product at least once in a month. DAU (daily active users) counts unique users in a day. DAU is always less than or equal to MAU. The ratio DAU/MAU shows how "sticky" an app is: a messaging app may have a high ratio (people use it daily), while a train booking app may have a low one (people use it when they travel).

2.4.2 QPS Formula

QPS means queries (or requests) per second. The basic formula:

Average QPS = $(\text{DAU} \times \text{actions per user per day}) \div 86,400$
 $\approx (\text{DAU} \times \text{actions per user per day}) \div 10^5$

Example: 10 million DAU, each user makes 20 requests per day.

```
Total requests/day = 10^7 × 20 = 2 × 10^8
Average QPS ≈ 2 × 10^8 ÷ 10^5 = 2,000 QPS
(exact: 2 × 10^8 ÷ 86,400 ≈ 2,315 QPS)
```

2.4.3 Read QPS and Write QPS

Most systems read far more than they write. Social media might be 100:1 (you scroll many posts but post rarely). A logging system might be the opposite (lots of writes, few reads). Always separate them, because reads and writes are scaled differently: reads with caches and replicas, writes with sharding and queues.

2.4.4 Peak Factor

Traffic is not flat. People use food apps at lunch and dinner, cricket apps during matches, and IRCTC at exactly 10 a.m. for Tatkal. The **peak factor** is how much higher peak traffic is than the average.

```
Peak QPS = Average QPS × peak factor
```

- A common default assumption: peak factor of **2 to 3**.
- For spiky events (flash sales, ticket booking openings, live sports, New Year midnight messages), it can be **10 or more**.

You design capacity for the **peak**, not the average. A bridge designed for average traffic collapses on festival day.

💡 DO YOU KNOW?

Live cricket streaming creates some of the sharpest traffic peaks anywhere. During the 2023 Cricket World Cup final, Disney+ Hotstar reported concurrent viewership peaks of tens of millions (it reported about 5.9 crore concurrent viewers during the India vs Australia final). Such peaks can be many times higher than normal daily traffic, and spike sharply at key moments of the match.

🤔 THINK AND DISCUSS

IRCTC Tatkal booking opens at a fixed time every morning. Most of the day's booking traffic arrives in the first few minutes. Is a peak factor of 3 reasonable here? How would you estimate the real peak? What tricks (queues, waiting rooms, captchas) could reduce the effective peak?

✓ CHECK YOUR PROGRESS 2.4

1. An app has 50 million DAU, and each user makes 10 requests per day. What is the average QPS?

Answer: $5 \times 10^7 \times 10 = 5 \times 10^8$ requests/day. Divide by 10^5 : about 5,000 QPS (exact about 5,800).

2. If the peak factor is 3, what peak QPS should you design for?

Answer: About 15,000 QPS (or about 17,400 using the exact average).

3. Why separate read QPS from write QPS?

Answer: Because they are scaled with different techniques (reads with caches and replicas, writes with sharding and batching), and their volumes are often very different.

2.5 Storage Estimation

Storage estimation answers: how much disk do we need, today and in five years?

2.5.1 The Formula

```
Storage/day    = new items per day × size per item
Storage total = storage/day × 365 × years of retention × replication factor
```

Remember to include:

- **Metadata:** IDs, timestamps, user IDs, indexes. Often as large as the content itself for small items.
- **Media:** photos and videos are usually far bigger than text. Estimate them separately.
- **Replication:** data is usually stored 3 times for durability (a common default).
- **Growth:** traffic often grows each year. Add a buffer.
- **Derived copies:** thumbnails, multiple video resolutions, search indexes, backups.

2.5.2 Example

A notes app: 1 million new notes per day, each 2 KB including metadata, kept for 5 years, replicated 3 times.

```
Per day    =  $10^6 \times 2 \text{ KB} = 2 \times 10^9 \text{ bytes} = 2 \text{ GB/day}$ 
Per year   =  $2 \text{ GB} \times 365 \approx 730 \text{ GB} \approx 0.73 \text{ TB}$ 
5 years    =  $3.65 \text{ TB}$ 
× 3 replicas ≈ 11 TB
```

11 TB is manageable: a few machines, or one managed database with large disks. No need for exotic technology.

✓ CHECK YOUR PROGRESS 2.5

1. Why do we multiply storage by a replication factor?

Answer: Because data is stored in multiple copies (often 3) for durability and availability, so the raw disk needed is that many times the logical data size.

2. An app stores 500,000 photos per day at 2 MB each. How much storage per day before replication?

Answer: $5 \times 10^5 \times 2 \times 10^6 \text{ bytes} = 10^{12} \text{ bytes} = 1 \text{ TB per day.}$

2.6 Bandwidth Estimation

Bandwidth is the amount of data moving per second. Estimate **ingress** (data coming into your system, like uploads) and **egress** (data going out, like downloads and video streams) separately. Egress is usually much larger and, on cloud providers, it is usually what you pay for.

```
Ingress = write QPS × average request size
Egress  = read QPS  × average response size
```

Example: an image site with 1,000 uploads per second of 500 KB and 20,000 image views per second of 500 KB.

```
Ingress = 1,000 × 500 KB = 500 MB/s ≈ 4 Gbps
Egress  = 20,000 × 500 KB = 10 GB/s ≈ 80 Gbps
```

80 Gbps of egress is far too much for a handful of servers. This number tells you immediately: you need a **CDN** to serve images from the edge.

For streaming, think in **concurrent users × bitrate**: 1 million people watching a 5 Mbps stream need $5 \times 10^6 \text{ Mbps} = 5 \text{ Tbps}$ of egress.

⚠ COMMON MISTAKE

Forgetting to convert bytes to bits when comparing with network capacity. 500 MB/s is 4,000 Mbps = 4 Gbps, not 0.5 Gbps. Multiply bytes by 8 to get bits.

✓ CHECK YOUR PROGRESS 2.6

1. What is the difference between ingress and egress?

Answer: Ingress is data flowing into the system (uploads, writes). Egress is data flowing out to users (downloads, page views, streams).

2. 2,000 requests per second each returning 50 KB. What is the egress in Mbps?

Answer: $2,000 \times 50 \text{ KB} = 100 \text{ MB/s.} \times 8 = 800 \text{ Mbps.}$

2.7 Cache Memory Estimation and the 80/20 Rule

The **80/20 rule** (Pareto principle) says that, in many systems, about 20% of the items get about 80% of the traffic. A few viral videos, popular products or famous celebrities get most of the views. So if we keep that hot 20% in a fast in-memory cache, we can serve most reads without touching the database.

2.7.1 The Formula

$$\text{Cache memory} \approx 0.2 \times (\text{daily read requests} \times \text{size per item})$$

Strictly, it should be 20% of the *unique* items read per day, because the same item is read many times. Using total requests gives a safe upper bound. If you know the number of unique items, use it.

2.7.2 Example

A news site serves 10 million article reads per day, average article (cached rendered form) 10 KB.

$$\begin{aligned} \text{Daily read data} &= 10^7 \times 10 \text{ KB} = 10^{11} \text{ bytes} = 100 \text{ GB} \\ \text{Cache (20\%)} &= 20 \text{ GB} \end{aligned}$$

20 GB fits easily in the RAM of a single cache server (machines with 64 to 256 GB of RAM are common). For safety and availability, you might use two or three cache nodes.

KEY POINT

If your hot data fits in the memory of a few machines, caching is a cheap and powerful way to cut database load and latency. The 80/20 estimate tells you quickly whether it fits.

CHECK YOUR PROGRESS 2.7

1. State the 80/20 rule as applied to caching.

Answer: Roughly 20% of items receive about 80% of the requests, so caching that hot 20% serves most reads from memory.

2. A service reads 200 million items per day, each 1 KB. Estimate cache memory needed.

Answer: $2 \times 10^8 \times 1 \text{ KB} = 200 \text{ GB}$ per day; 20% is about 40 GB of cache.

2.8 Estimating the Number of Servers

Once you know peak QPS, you can estimate how many servers you need.

$$\text{Servers} = \text{Peak QPS} \div \text{QPS one server can handle} \quad (\text{then add headroom})$$

2.8.1 How Much Can One Server Handle?

This depends hugely on the work per request. Some rough, commonly used assumptions:

Type of work	Rough QPS per server
Serving cached static content	10,000 to 100,000+
Simple API with cache hits (key lookups)	a few thousand to tens of thousands
Typical web app request with a DB call and some logic	about 100 to 1,000
Heavy computation (image processing, ML inference on CPU)	about 1 to 100
In-memory cache node (like Redis) simple GET/SET	about 100,000 (order of magnitude)
Single relational DB, simple indexed queries	thousands to low tens of thousands

You can also reason from CPU time: if each request needs 10 ms of CPU and a server has 16 cores, one server can do at most $16 \times (1,000 \div 10) = 1,600$ requests per second at 100% CPU. Running at about 50 to 70% leaves room for spikes, so plan around 800 to 1,100.

2.8.2 Headroom and Redundancy

- Do not plan servers to run at 100%. Target about 50 to 70% utilisation at peak.
- Add at least one extra server (N+1) so you survive a failure; across availability zones, plan to survive losing one whole zone.

Example: peak 30,000 QPS, each server handles 1,000 QPS comfortably. $30,000 \div 1,000 = 30$ servers. With headroom for zone failure across 3 zones (each zone must carry half the load if one is lost), about 45 servers. A rough answer of "about 40 to 50" is perfect.

✓ CHECK YOUR PROGRESS 2.8

1. Each request needs 20 ms of CPU. A server has 8 cores. What is the theoretical max QPS per server?

Answer: $8 \times (1,000 \div 20) = 8 \times 50 = 400$ QPS at 100% CPU. Plan for around 200 to 280.

2. Why add headroom instead of running servers at 100% CPU?

Answer: Latency shoots up as utilisation approaches 100% because requests queue, and you need spare capacity to absorb sudden spikes and to survive a server or zone failure.

2.9 Fully Worked Examples

Now let us put everything together. Read each example slowly with a pen in hand. Notice how every number comes from a stated assumption.

2.9.1 Example 1: A Twitter-like Service

Assumptions

- 200 million DAU.
- Each user posts 2 tweets per day on average.
- Each user views 100 tweets per day (timeline scrolling).
- Tweet text plus metadata: about 1 KB (text up to 280 characters, IDs, timestamps, counters, indexes).
- 10% of tweets contain an image of about 500 KB (after compression).
- Data kept for 5 years. Replication factor 3. Peak factor 3.

Traffic

```

Tweets/day    = 2 × 108 users × 2 = 4 × 108
Write QPS     = 4 × 108 ÷ 105 = 4,000/s    (exact ≈ 4,600)
Peak writes   ≈ 12,000/s

Views/day     = 2 × 108 × 100 = 2 × 1010
Read QPS      = 2 × 1010 ÷ 105 = 200,000/s  (exact ≈ 231,000)
Peak reads    ≈ 600,000/s
Read:write    ≈ 50:1

```

Storage

```

Text/day      = 4 × 108 × 1 KB = 4 × 1011 B = 400 GB/day
Text/year     ≈ 400 GB × 365 ≈ 146 TB
Media/day     = 4 × 107 images × 500 KB = 2 × 1013 B = 20 TB/day
Media/year    ≈ 20 TB × 365 ≈ 7.3 PB
5 years       ≈ (0.146 + 7.3) PB × 5 ≈ 37 PB
× 3 copies    ≈ 110 PB

```

Bandwidth (assume a viewed tweet response averages 1 KB text, and 10% of views load a 500 KB image, often a smaller version in practice):

```

Text egress   = 2 × 105 × 1 KB = 200 MB/s ≈ 1.6 Gbps
Image egress  = 2 × 104 × 500 KB = 10 GB/s ≈ 80 Gbps (average)
Peak          ≈ 240 Gbps → images must go through a CDN

```

Cache: 20% of daily read data. Text reads are $2 \times 10^{10} \times 1 \text{ KB} = 20 \text{ TB}$ per day of reads, but unique tweets read are far fewer (popular tweets are read millions of times). Caching recent and popular tweets (say the last day's 400 GB of new tweets plus hot older ones) needs on the order of a few TB of RAM, spread over tens of cache servers.

Conclusions: the system is heavily read-dominated, so we precompute timelines and cache aggressively. Media dominates storage and bandwidth, so we use object storage plus a CDN. Text data alone is moderate (hundreds of TB over years) but still needs sharding.

2.9.2 Example 2: A WhatsApp-like Messaging Service

Assumptions

- 500 million DAU.

- Each user sends 40 messages per day.
- Average message: 100 bytes of text, about 200 bytes with metadata (sender, receiver, timestamp, message ID, status).
- Messages are stored on the server only until delivered, plus 30 days of backup buffer for undelivered or multi-device sync (simplified assumption).
- At peak, about 40% of DAU are online at the same time.
- Peak factor 3 (New Year and festival greetings can spike far higher).

Traffic

```
Messages/day = 5 × 108 × 40 = 2 × 1010 (20 billion)
Average QPS  = 2 × 1010 ÷ 105 = 200,000 msgs/s (exact ≈ 231,000)
Peak         ≈ 600,000 to 700,000 msgs/s
```

Each message is both written (send) and read (deliver), plus delivery and read receipts, so internal operations are a few times higher.

Storage

```
Per day    = 2 × 1010 × 200 B = 4 × 1012 B = 4 TB/day
30 days   = 120 TB
× 3 copies ≈ 360 TB
```

If instead we stored every message forever: 4 TB × 365 ≈ 1.5 PB per year before replication. That is why the storage policy (keep until delivered vs keep forever) is a big design decision.

Connections

```
Concurrent users = 0.4 × 5 × 108 = 2 × 108 (200 million)
If one server holds about 500,000 persistent connections:
Servers = 2 × 108 ÷ 5 × 105 = 400 servers (plus headroom → ~600)
```

Bandwidth (text only):

```
200,000 msgs/s × 200 B = 40 MB/s ≈ 320 Mbps average
```

Text bandwidth is small. Photos, voice notes and videos dominate, and are stored in object storage and sent separately.

Conclusions: the hard part of a chat system is not storage or bandwidth for text. It is holding hundreds of millions of persistent connections and routing each message to the right connection with low latency.

💡 DO YOU KNOW?

In 2012, WhatsApp engineers wrote that they had tuned a single server (running Erlang on FreeBSD) to hold more than 2 million concurrent TCP connections. That kind of per-server efficiency is a big reason a small team could serve hundreds of millions of users.

2.9.3 Example 3: A YouTube-like Video Platform

Assumptions

- 500 hours of video uploaded every minute (YouTube publicly stated a figure of about this size around 2019).
- After encoding into several resolutions (240p to 1080p and more), each hour of video uses about 3 GB of storage in total (a simplified assumption).
- 100 million DAU, each watching 5 videos per day; average watch time 10 minutes per video.
- Average streaming bitrate about 5 Mbps (around 1080p).
- Replication factor 3 for storage.

Upload and storage

```
Hours uploaded/day = 500 × 60 × 24 = 720,000 hours
Storage/day        = 720,000 × 3 GB ≈ 2.16 PB/day
Storage/year       ≈ 2.16 × 365 ≈ 790 PB ≈ 0.8 EB
× 3 copies         ≈ 2.4 EB per year
```

Views

```
Views/day          = 10^8 × 5 = 5 × 10^8
Average view QPS   = 5 × 10^8 ÷ 10^5 = 5,000 video starts/s
```

Concurrent viewers and bandwidth

```
Watch minutes/day  = 5 × 10^8 × 10 = 5 × 10^9 minutes
Minutes in a day    = 1,440
Avg concurrent     = 5 × 10^9 ÷ 1,440 ≈ 3.5 million viewers
Egress             = 3.5 × 10^6 × 5 Mbps ≈ 17.5 Tbps (average)
Peak (× 2)         ≈ 35 Tbps
```

Conclusions: video platforms are dominated by storage and egress bandwidth. Tens of terabits per second can only be served by a large CDN with caches placed inside internet service providers. Upload processing (transcoding into many resolutions) is a huge CPU job done asynchronously by queues and worker fleets.

💡 DO YOU KNOW?

Netflix runs its own content delivery network called Open Connect. It places Netflix-owned cache servers inside internet service providers' networks around the world, so most video bytes travel only a short distance to viewers instead of across the internet.

2.9.4 Example 4: A URL Shortener

Assumptions

- 100 million new short URLs created per month.
- Read:write ratio 100:1 (each link is clicked many times).

- Each record (short code, long URL, creation time, owner, expiry): about 500 bytes.
- Data kept for 5 years.

Traffic

```
Writes/month = 10^8
Write QPS    = 10^8 ÷ (2.5 × 10^6 s) ≈ 40/s
Read QPS     = 40 × 100 = 4,000/s
Peak reads   ≈ 10,000 to 12,000/s
```

Storage

```
URLs in 5 years = 10^8 × 12 × 5 = 6 × 10^9 (6 billion)
Storage         = 6 × 10^9 × 500 B = 3 × 10^12 B = 3 TB
× 3 copies      = 9 TB
```

Length of the short code: using Base62 characters (a to z, A to Z, 0 to 9):

```
62^6 ≈ 5.7 × 10^10 (about 57 billion)
62^7 ≈ 3.5 × 10^12 (about 3.5 trillion)
```

6 characters give 57 billion codes, enough for 6 billion URLs, but 7 characters give a lot of extra room for growth. Either is defensible; say why you chose it.

Bandwidth: reads $4,000/s \times 500 \text{ B} = 2 \text{ MB/s}$. Tiny. (The response is just a redirect.)

Cache: daily reads = $4,000 \times 86,400 \approx 3.5 \times 10^8$. $20\% \times 3.5 \times 10^8 \times 500 \text{ B} \approx 35 \text{ GB}$. Fits in the RAM of one or two cache servers.

Servers: 12,000 peak QPS of very cheap lookups, mostly from cache: a handful of app servers (say 5 to 10, including redundancy) is plenty.

Conclusions: a URL shortener is small in storage and bandwidth but read-heavy. A cache in front of a simple key-value store handles it comfortably. The interesting design questions are about generating unique short codes, not about scale.

System	Avg QPS (main op)	Storage growth	Main bottleneck
Twitter-like	~200K reads/s, ~4K writes/s	~7.5 PB/year (mostly media)	Read fan-out, media egress
WhatsApp-like	~200K messages/s	~4 TB/day of text (if kept)	Persistent connections, routing
YouTube-like	~5K video starts/s	~0.8 EB/year before replication	Egress bandwidth, storage, transcoding
URL shortener	~4K reads/s, ~40 writes/s	~3 TB in 5 years	Unique key generation, read latency

Table 2.3: Summary of the four worked examples (from our assumptions, not real company figures).

 ACTIVITY 2.2

Estimate for Instagram Stories: assume 300 million DAU, 30% post one story per day (a 1 MB photo or a 5 MB video, equal split), stories deleted after 24 hours, each story viewed 50 times. Calculate: uploads per second, storage needed at any one time, and average egress in Gbps. Then compare your numbers with a friend's.

 CHECK YOUR PROGRESS 2.9

1. In the WhatsApp-like example, why is text storage and bandwidth not the main challenge?

Answer: Because text messages are tiny (hundreds of bytes), so even 20 billion per day is only a few TB and a few hundred Mbps. The real challenge is holding hundreds of millions of persistent connections and routing messages to them quickly.

2. How many unique codes does a 7-character Base62 short code give?

Answer: $62^7 \approx 3.5$ trillion.

3. Why does the YouTube-like example conclude that a CDN is essential?

Answer: Because average egress is around 17.5 Tbps and peak is higher. No central data centre can economically serve that; caching video close to users through a CDN is the only practical way.

2.10 Common Mistakes in Estimation

1. **Confusing bits and bytes.** Network in bits, storage in bytes. Divide or multiply by 8.
2. **Forgetting peak traffic.** Designing for the average guarantees failure at the busiest, most important moment.
3. **Forgetting replication and backups.** Raw data $\times 3$ (or more) is the real disk need.
4. **Ignoring metadata and indexes.** For small items, metadata can double the size.
5. **Ignoring media.** One photo equals thousands of text messages. Always estimate media separately.
6. **Mixing DAU and MAU.** Use DAU for daily traffic.
7. **Spending too long on precision.** 86,400 vs 100,000 is not where the design is decided.
8. **Not stating assumptions.** Numbers without assumptions are impossible to review.
9. **Unit slips** such as dividing by 1,000 instead of 1,000,000 when going from KB to GB. Write the units in every line.
10. **Not drawing a conclusion.** An estimate is useful only if it changes a decision: "40 GB cache fits on one machine", "we need a CDN", "we must shard".

 FUNNY SIDE

Interviewer: "How much storage do we need?" Candidate (after 15 minutes of long division): "Exactly 3,214,567,890,112 bytes." Interviewer: "Great. Does it fit on one machine?" Candidate: "...I'll need another 15 minutes." Round your numbers, friends.

 CHECK YOUR PROGRESS 2.10**1. Name three common estimation mistakes.**

Answer: Any three of: confusing bits and bytes, ignoring peak traffic, forgetting replication, ignoring metadata, ignoring media, mixing DAU and MAU, over-precision, unstated assumptions, unit slips, not drawing a conclusion.

2. What should every estimate end with?

Answer: A design conclusion, such as "fits in memory", "needs a CDN", "needs sharding" or "a few servers are enough".

2.11 Cheat Table

Copy this table into your notebook. It covers most estimates you will ever need.

Item	Value to remember
Seconds per day	$86,400 \approx 10^5$
Seconds per month	$\approx 2.5 \times 10^6$
Seconds per year	$\approx 3 \times 10^7$
1 million requests/day	≈ 12 requests/second
1 billion requests/day	$\approx 12,000$ requests/second
KB, MB, GB, TB, PB	$10^3, 10^6, 10^9, 10^{12}, 10^{15}$ bytes
Bits to bytes	$\div 8$ (1 Gbps $\approx$ 125 MB/s)
RAM reference	≈ 100 ns
SSD random read	≈ 100 to 150 μ s
HDD seek	≈ 10 ms
Round trip in same data centre	≈ 0.5 ms
Cross-continent round trip	≈ 150 ms
Default peak factor	2 to 3 (10+ for flash events)
Default replication factor	3
Cache sizing	$\approx 20\%$ of daily read data (80/20 rule)

Item	Value to remember
Target server utilisation	50 to 70% at peak
Typical web server (DB-backed requests)	≈ 100 to 1,000 QPS
Base62 code space	$62^6 \approx 57$ billion, $62^7 \approx 3.5$ trillion
1080p stream	≈ 5 Mbps

Table 2.4: Estimation cheat table.

✓ CHECK YOUR PROGRESS 2.11

1. Using the cheat table, how many requests per second is 300 million requests per day?

Answer: 1 million/day ≈ 12 /s, so 300 million/day $\approx 3,600$ /s (or $3 \times 10^8 \div 10^5 = 3,000$ /s by the rough rule).

2. What utilisation should you plan for at peak, and why?

Answer: About 50 to 70%, to keep latency low and leave room for spikes and failures.

What Have We Learnt?

- Latency differs hugely between operations: L1 cache ≈ 0.5 ns, RAM ≈ 100 ns, SSD random read ≈ 150 μ s, HDD seek ≈ 10 ms, cross-continent round trip ≈ 150 ms.
- Memory is much faster than disk, and disk is often faster than long network trips; this is why caches and CDNs exist.
- For estimation, $2^{10} \approx 10^3$, and KB, MB, GB, TB, PB are 10^3 , 10^6 , 10^9 , 10^{12} , 10^{15} bytes.
- One day $\approx 10^5$ seconds, so daily counts $\div 10^5 \approx$ per-second rates.
- Back-of-the-envelope estimates aim for the right order of magnitude with clearly written assumptions.
- Average QPS = DAU $\times$ actions per user $\div 86,400$; peak QPS = average $\times$ peak factor (often 2 to 3, more for spiky events).
- Storage = items/day $\times$ size $\times$ days $\times$ retention $\times$ replication; estimate media separately.
- Bandwidth = QPS $\times$ size; convert bytes to bits for network capacity; egress usually dominates.
- The 80/20 rule suggests caching about 20% of daily read data.
- Number of servers = peak QPS $\div$ per-server capacity, with headroom (50 to 70% utilisation) and N+1 redundancy.
- Different systems have different bottlenecks: timelines for Twitter, connections for WhatsApp, bandwidth and storage for YouTube, key generation for URL shorteners.
- Every estimate should end with a design conclusion.

Key terms: latency, nanosecond, microsecond, millisecond, L1/L2 cache, RAM, SSD, HDD seek, sequential read, random read, round-trip time (RTT), bit, byte, kilobyte, megabyte, gigabyte, terabyte, petabyte, exabyte, kibibyte, order of magnitude, back-of-the-envelope estimation, DAU, MAU, QPS, read:write ratio, peak factor, storage estimation, replication factor, retention, bandwidth, ingress, egress, Mbps, Gbps, 80/20 rule, Pareto principle, cache sizing, utilisation, headroom, N+1 redundancy, Base62.

Exercises

A. Multiple Choice Questions

1. Which is the fastest?
 - a. SSD random read
 - b. RAM reference
 - c. L1 cache reference
 - d. Round trip within a data centre
2. An HDD seek takes approximately:
 - a. 10 ns
 - b. 10 μ s
 - c. 10 ms
 - d. 10 s
3. For estimation, the number of seconds in a day is taken as about:
 - a. 10^3
 - b. 10^4
 - c. 10^5
 - d. 10^7
4. A 100 Mbps connection can download at most about:
 - a. 100 MB/s
 - b. 12.5 MB/s
 - c. 800 MB/s
 - d. 1 GB/s
5. 10 million DAU each making 50 requests per day gives an average QPS of about:
 - a. 500
 - b. 5,000
 - c. 50,000
 - d. 500,000
6. The 80/20 rule in caching suggests caching about:
 - a. 80% of all data
 - b. 20% of the daily read data
 - c. 100% of writes
 - d. 2% of data
7. 1 PB is approximately:
 - a. 10^9 bytes
 - b. 10^{12} bytes
 - c. 10^{15} bytes
 - d. 10^{18} bytes
8. Which most likely dominates the bandwidth of a video platform?
 - a. Ingress of comments
 - b. Egress of video streams
 - c. Login requests
 - d. DNS lookups
9. 62^7 is approximately:
 - a. 3.5 million
 - b. 3.5 billion
 - c. 3.5 trillion
 - d. 57 billion

10. Why do we plan servers for about 50 to 70% utilisation at peak?

- a. To save electricity
- b. To keep latency low and survive spikes and failures
- c. Because servers cannot exceed 70% CPU
- d. Because cloud providers require it

B. Fill in the Blanks

1. Network speeds are usually measured in _____ per second, while file sizes are measured in _____.
2. A main memory (RAM) reference takes about _____ nanoseconds.
3. Peak QPS = average QPS × _____.
4. Data flowing out of your system to users is called _____.
5. 1 crore equals _____ million.
6. A cross-continent packet round trip takes roughly _____ ms.

C. True or False

1. Reading sequentially from a disk is generally faster than reading randomly.
2. You should size your system for average traffic only.
3. Replication increases the total raw storage needed.
4. In estimation, an answer of 4,000 QPS when the true value is 4,630 is a bad estimate.
5. A text messaging system's biggest challenge is usually text storage size.

D. Short Answer Questions

1. Why does the speed of light matter in system design?
2. Explain the difference between DAU and MAU with an example.
3. Write the formula for storage estimation and list three things people often forget to include.
4. How do you estimate the number of servers needed? Include headroom in your answer.
5. Why should read and write traffic be estimated separately?
6. Convert 2 TB per day into an average MB/s rate.

E. Long Answer Questions

1. Explain the back-of-the-envelope estimation method step by step, using a photo-sharing app with 50 million DAU as an example.
2. Do a full estimate (traffic, storage, bandwidth, cache, servers) for a food delivery app with 20 million DAU. State all assumptions.
3. Discuss the lessons system designers learn from the latency numbers table.

F. Design Challenge

Estimate the scale for a "Pastebin" service where users paste text and get a shareable link. Assume 1 million new pastes per day, average paste 10 KB, read:write ratio 10:1, pastes kept for 3 years.

Compute write QPS, read QPS, peak QPS, total storage with replication, egress bandwidth, cache size and a rough number of app servers. State one design conclusion from your numbers.

Answer Key: Chapter 2

A. MCQ

1. (c) L1 cache ≈ 0.5 ns.
2. (c) About 10 ms.
3. (c) $86,400 \approx 10^5$.
4. (b) $100 \div 8 = 12.5$ MB/s.
5. (b) $5 \times 10^8 \div 10^5 = 5,000$.
6. (b) About 20% of daily read data.
7. (c) Peta = 10^{15} .
8. (b) Video egress dwarfs everything else.
9. (c) About 3.5 trillion.
10. (b) Latency rises near full load; spare room absorbs spikes and failures.

B. Fill in the Blanks

1. bits; bytes
2. 100
3. peak factor
4. egress
5. 10
6. 150

C. True or False

1. True. Sequential access avoids repeated seeks and is faster on both HDDs and SSDs.
2. False. You must design for peak traffic, with headroom.
3. True. Each replica is a full copy, so 3 replicas need about 3 times the disk.
4. False. It is within the same order of magnitude, which is the goal.
5. False. Text is small; persistent connections and message routing are the bigger challenges.

D. Short Answers

1. Signals travel through fibre at about 200,000 km/s, so distance sets a minimum latency that no hardware can beat. A round trip between continents takes over 100 ms. To serve distant users quickly, designers must place servers, caches or CDNs physically closer to them.
2. DAU counts unique users active in a day; MAU counts unique users active in a month. A train booking app might have 30 million MAU but only 5 million DAU, because people book only when they travel. DAU is used for daily traffic estimates.
3. Storage = new items per day $\times$ size per item $\times 365 \times$ years $\times$ replication factor. Commonly forgotten: metadata and indexes, media (photos and video), replication and backups, derived copies like thumbnails, and growth over time.
4. Estimate peak QPS, estimate how many QPS one server can handle (from benchmarks or CPU time per request), and divide. Then plan for 50 to 70% utilisation instead of 100%, and add spare servers (N+1, or enough to survive losing a zone).
5. Because their volumes often differ greatly (social apps read about 100 times more than they write), and they are scaled with different techniques: caches and replicas for reads, sharding,

batching and queues for writes.

6. $2 \text{ TB/day} = 2 \times 10^{12} \text{ bytes} \div 86,400 \text{ s} \approx 2.3 \times 10^7 \text{ bytes/s} \approx 23 \text{ MB/s}$ (about 20 MB/s using 10^5 seconds), which is about 185 Mbps.

E. Long Answers

1. Model answer points:

- State assumptions: 50 million DAU, 10% upload 1 photo per day (5 million uploads), each photo 2 MB, each user views 50 photos per day, keep forever, replication 3, peak factor 3.
- Write QPS: $5 \times 10^6 \div 10^5 = 50/\text{s}$; peak $\approx 150/\text{s}$.
- Read QPS: $2.5 \times 10^9 \div 10^5 = 25,000/\text{s}$; peak $\approx 75,000/\text{s}$.
- Storage: $5 \times 10^6 \times 2 \text{ MB} = 10 \text{ TB/day}$; $\approx 3.65 \text{ PB/year}$; $\times 3 \approx 11 \text{ PB/year}$.
- Egress: assume feed images are compressed to 200 KB: $25,000 \times 200 \text{ KB} = 5 \text{ GB/s} \approx 40 \text{ Gbps}$ average, so use a CDN.
- Cache: metadata for hot photos fits in RAM; image bytes are cached by the CDN.
- Servers: at about 1,000 QPS per server for metadata APIs, about 75 at peak plus headroom, so about 100 to 150.
- Conclusion: storage and image egress dominate; use object storage, CDN and a sharded metadata database.

2. Model answer points:

- Assumptions: 20 million DAU; 10% order once a day (2 million orders/day); each user makes 30 API calls per day (browsing, search, tracking); order record 2 KB; location updates from 300,000 active delivery partners every 5 seconds at peak; peak factor 4 (lunch and dinner).
- API QPS: $6 \times 10^8 \div 10^5 = 6,000/\text{s}$ average; peak $\approx 24,000/\text{s}$.
- Order writes: $2 \times 10^6 \div 10^5 = 20/\text{s}$ average; peak $\approx 80/\text{s}$; low but must be strongly consistent and durable.
- Location updates: $300,000 \div 5 = 60,000 \text{ writes/s}$ at peak: this is the real write hotspot, stored in memory, not in the main database.
- Storage: orders $2 \times 10^6 \times 2 \text{ KB} = 4 \text{ GB/day} \approx 1.5 \text{ TB/year}$; $\times 3 \approx 4.5 \text{ TB/year}$. Menus and images are separate, served by CDN.
- Cache: restaurant menus and popular searches by city; tens of GB.
- Servers: at 500 QPS per server, about 48 servers at peak plus headroom, so about 70 to 80.
- Conclusion: order data is small but critical; location tracking needs a fast in-memory store; menus need caching and a CDN.

3. Model answer points:

- Memory is about 100,000 times faster than an HDD seek, so caching hot data in RAM is very effective.
- Disk seeks are expensive, so designs favour sequential writes (logs, LSM trees).
- SSDs are much faster than HDDs for random reads; choose storage based on access pattern.
- Network calls inside a data centre cost about 0.5 ms; chains of many calls add up, so limit sequential service calls.

- Cross-continent latency (about 150 ms) is limited by physics; use CDNs and regional deployments.
- Compression is cheap compared with sending bytes over slow links.
- Numbers change with hardware, but ratios guide design decisions.

F. Design Challenge: Model Solution

```

Write QPS    = 10^6 ÷ 10^5 = 10/s    (exact ≈ 12/s)
Read QPS     = 10 × 10 = 100/s      (exact ≈ 120/s)
Peak (× 3)   ≈ 30 writes/s, 300 to 360 reads/s
Storage/day  = 10^6 × 10 KB = 10 GB/day
3 years      = 10 GB × 365 × 3 ≈ 11 TB
× 3 copies   ≈ 33 TB
Egress       = 100/s × 10 KB = 1 MB/s ≈ 8 Mbps (peak ≈ 30 Mbps)
Cache        = 20% × (100 × 86,400 × 10 KB) ≈ 20% × 86 GB ≈ 17 GB
Servers      = 360 peak QPS ÷ ~500 per server → 1, but run 2 to 3 for redundancy

```

Conclusion: traffic is small; a few app servers with a cache are enough. The main concern is storage growth (about 33 TB over 3 years with replication), so paste contents should go to cheap object storage, with only metadata (key, owner, expiry, location of the content) in the database. Expired pastes should be deleted by a background job to control cost.