DESIGN SCALABLE SYSTEMS • THINK LIKE A SENIOR ENGINEER • ACE YOUR INTERVIEWS

CRACKING THE MOBILE SYSTEM DESIGN INTERVIEW

A PRACTICAL GUIDE TO DESIGNING **SCALABLE, PERFORMANT AND RELIABLE MOBILE APPLICATIONS** AND ACE YOUR SYSTEM DESIGN INTERVIEWS WITH CONFIDENCE

- INTERVIEW-TESTED APPROACH
- REAL-WORLD CASE STUDIES
- HLD & LLD DEEP DIVE
- TRADE-OFFS & BEST PRACTICES
- PERFORMANCE, SCALABILITY & SECURITY

ANAND GAUR

Cracking the Mobile System Design Interview

Preview Edition

A Practical Guide to Designing Scalable Mobile Applications

Author: Anand Gaur

About This Preview Edition

Thank you for exploring the preview edition of **Cracking the Mobile System Design Interview**.

This book is designed for Android, iOS, Flutter, React Native, and cross-platform developers who want to master Mobile System Design, scalable architectures, offline-first systems, real-world engineering decisions, and interview-focused design thinking.

The complete book contains 650+ pages of practical system design concepts, architecture patterns, scalability strategies, interview frameworks, and real-world mobile case studies specifically designed for mobile engineers.

This preview edition contains selected chapters and sample content to help readers understand the depth, structure, and practical approach used throughout the book.

What You'll Explore in This Preview

Inside this preview edition, you'll get an overview of:

- Introduction to Mobile System Design
- HLD vs LLD Fundamentals
- Client-Server Architecture
- APIs, WebSocket & Real-Time Communication
- Offline-First Mobile Design
- Networking & Caching Strategies
- Mobile Performance Optimization
- Scalable Mobile Architecture Patterns
- Authentication & Security Basics
- Real-World Mobile System Design Case Studies

Table of Contents

PART I · SYSTEM DESIGN BASICS

1. What is System Design? ……..……..……..…….9
• System Design in Simple
• Why Companies Ask System Design Questions
• Backend System Design vs. Mobile System Design
• HLD vs LLD — Two Levels of System Design
• What Mobile Interviewers Actually Care About

2. Core Concepts Every Mobile Developer Must Know …… 30
• Client-Server Model
• APIs (REST, GraphQL, WebSocket — when to use what)
• Latency, Throughput, Bandwidth
• Synchronous vs. Asynchronous
• Scalability — what it means on the client side

3. Mobile-Specific Constraints ……..……..……..…….. 49
• Limited Battery, Memory, CPU, Storage
• Unreliable Networks (2G, 3G, 4G, 5G, Wi-Fi, Offline)
• App Lifecycle (foreground, background, killed)
• Why Mobile System Design is Different from Backend

PART II · MOBILE SYSTEM DESIGN BUILDING BLOCKS

4. App Architecture Patterns ……..……..……..……..……69
• MVC, MVP, MVVM, MVI, Clean Architecture (quick comparison)
• Choosing the Right Pattern in an Interview
• Layered Thinking: Presentation → Domain → Data
• Dependency Injection

5. Networking Layer Design ……..……..……....101
• Designing a Robust API Client
• Error Handling, Retries, Timeouts
• Pagination (Offset, Cursor, Keyset)
• Image and File Upload/Download
• Content Delivery Networks (CDN)

6. Storage and Caching ……..……..……....133
• Memory Cache vs. Disk Cache
• SQLite, Room, Core Data, Realm — when to pick which

• Cache Eviction (LRU, TTL)
• Cache Invalidation Strategies

7. **Offline-First Design** ……153
• Why Every Modern App Needs Offline Support
• Optimistic Updates
• Sync Queues and Replay Logic
• Conflict Resolution Basics

8. **Real-Time Updates** ……175
• Polling vs. Long Polling vs. WebSocket vs. SSE
• Push Notifications (FCM, APNs)
• When to Use What — A Decision Guide

9. **Performance Essentials** ……192
• App Launch Optimization
• Smooth Scrolling and 60 FPS
• Image Loading and Memory Management
• Battery and Network Efficiency

10. Security Basics ……212
• HTTPS and Certificate Pinning
• Token Storage (Keychain, Keystore)
• Data Encryption at Rest
• Common Mobile Security Pitfalls
• Authentication Flows

PART III · THE INTERVIEW FRAMEWORK

11. **How to Approach a Mobile System Design Interview** ……245
• The 5-Step Framework: **CRDDS**
  • **C**larify Requirements
  • **R**ough High-Level Design
  • **D**eep Dive into Components
  • **D**iscuss Trade-offs
  • **S**ummarize and Handle Follow-ups
• Knowing the Round: HLD-Focused vs LLD-Focused Interview

12. **Asking the Right Clarifying Questions** ……265
• Functional vs. Non-Functional Requirements
• Scale (users, data, geography)
• Platform (iOS, Android, both?)

• Online/Offline Expectations
• A Reusable Question Checklist

13. **Drawing the High-Level Design (HLD)** ……..……..……..……283
• The Standard Mobile Architecture Diagram
• UI Layer, Business Layer, Data Layer, Network Layer
• How to Draw on a Whiteboard or Virtual Tool
• What HLD Covers — and What It Leaves for LLD

14. **Going Deep — Component-Level Design (LLD)** ……..…….……303
• Picking the Right Component to Deep Dive Into
• Class Diagrams, Interfaces, and Design Patterns in Action
• SOLID Principles Applied Live
• Showing Trade-offs Like a Senior Engineer
• Discussing Edge Cases

15. **Common Mistakes to Avoid** ……..…….……327
• Jumping to Solution Without Clarifying
• Ignoring Mobile Constraints
• Over-Engineering
• Not Discussing Trade-offs
• Mixing Up HLD and LLD Levels

16 · CASE STUDIES (Real Interview Questions)

16.1. Designing a Chat App (WhatsApp / Telegram)……..……346

16.2. Designing a News Feed (Twitter / Facebook)……..……..387

16.3. Designing a Photo Sharing App (Instagram)……..……...428

16.4. Designing a Ride-Sharing App (Uber / Ola)……..……464

16.5. Designing a Video Streaming App (Netflix / YouTube)……..……496

16.6. Designing a Food Delivery App (Swiggy / Zomato)……..……527

16.7. Designing an E-Commerce App (Amazon / Flipkart)……..……558

16.8. Designing a Stock Trading App(Groww / Zerodha)……..……588

16.9. Designing a Hotel Reservation App (Airbnb / OYO)……..…..622

Appendix B: Glossary ……..……..……..654

Chapter 1: What is System Design?

Welcome to your journey into mobile system design. Before we jump into complex topics like designing Instagram or Uber for mobile, we must understand the basics. This chapter answers the most important question: **what is system design, and why does it matter for a mobile engineer?**

1.1 System Design in Simple Way

Imagine you want to build a house. Before you put one brick on top of another, you need a plan. You must decide:

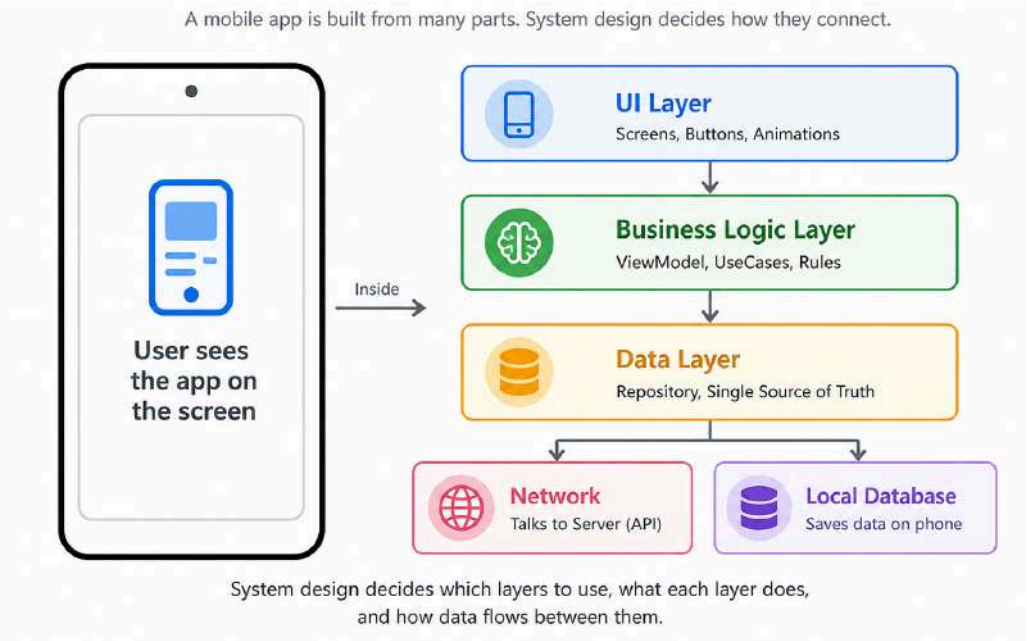
- How many rooms?
- Where will the kitchen go?
- Where will the bathroom be?
- How will water flow through the pipes?
- Where will electricity come from?

This planning step is called **design**.

System design works exactly the same way, but for software. Before you write a single line of code for an app, you must decide:

- What screens does the app have?
- Where does the data come from?
- How does the app talk to the server?
- What happens when the internet is slow or off?
- How and where do we save data on the phone?

System design is the process of planning how all parts of an app or software work together to solve a problem.



In a real, complex app like WhatsApp, Instagram, or Uber, there are many parts working together:

- The screens (UI).
- The buttons, gestures, and animations.
- The code that handles user actions.
- The code that talks to servers over the internet.
- The local database that saves messages.
- The network layer that handles slow internet.
- The cache that makes the app fast.
- The push notification system.
- The image-loading pipeline.
- And many more.

System design is about **deciding how all these parts fit together** so that the app works smoothly, scales to many users, and can be improved by your team for years to come.

A Real Example

Let's say you want to build a chat app like WhatsApp. Some questions you must answer during system design include:

1. When a user sends a message, where does it go first — to the server, or to the local database?
2. If the internet is off, can the user still see old messages and type new ones?
3. How do we deliver a message to another user instantly?
4. How do we show "online" or "typing…" status without draining the battery?
5. Where do we save photos and videos? On the device? On the server? Both?

Each question above leads to a design choice. These choices, taken together, form the **system design**.

System Design is Decision-Making

The most important thing to understand is that system design is **not about writing code**. It is about making **decisions** before code is written. A good designer thinks through the problem, weighs the options, and picks the best approach for the situation.

For example, a backend developer may decide: "We will use PostgreSQL because we need strong consistency." A mobile developer may decide: "We will use Room because it gives compile-time SQL checks and works well with Coroutines."

Both are design decisions. Both shape the future of the project.

1.2 Why Companies Ask System Design Questions

You may wonder, "I am applying for a mobile developer job. Why do I need to learn system design?"

Great question. Here is the truth:

*Companies don't just want coders. They want **engineers** who can think.*

Anyone can write code that works for 10 users. But what about 10 million users? What if the user is on a 2G network in a small village? What if the phone has only 1 GB of RAM and Chrome is also open? These are the real problems that mobile engineers face every single day.

Chapter 2: Core Concepts Every Mobile Developer

Before you can design a mobile app, you must understand the **basic concepts** that every mobile system uses. These are the building blocks. Once you know these, the bigger topics — caching, offline support, sync, real-time chat — all make much more sense.

In this chapter, we will learn five must-know concepts:

1. The Client-Server Model — how phones talk to servers.
2. APIs — REST, GraphQL, and WebSocket, and when to use which.
3. Latency, Throughput, and Bandwidth — three words that describe network speed.
4. Synchronous vs. Asynchronous — two ways code can run.
5. Scalability on the client side — yes, even mobile apps need to scale.

By the end of this chapter, you will have the full vocabulary needed for the rest of the book.

2.1 The Client-Server Model

Almost every mobile app you use today — WhatsApp, Instagram, Uber, Netflix, Swiggy — works using something called the client-server model. This is the foundation of all internet-based apps.

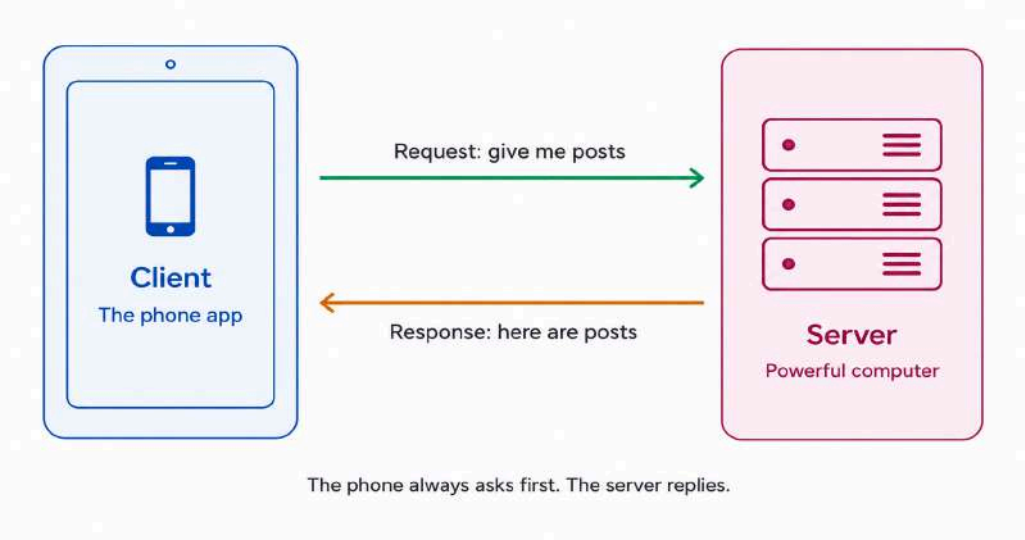
Client = the user's device that asks for something. (Your phone, the app.)

Server = a powerful computer somewhere far away that gives the answer.

When you open Instagram and pull down to refresh:

1. Your phone (the client) sends a message: "Give me the latest posts."
2. A computer in a data center (the server) reads the message, finds the posts, and sends them back.
3. Your phone receives the posts and shows them on the screen.

That's it. Everything else — login, payments, chat, video — is just a more complex version of this same dance.



Why Do We Use This Model?

You may wonder: "Why can't the phone do everything by itself?"

A phone is small and limited. A server is huge and powerful. By splitting the work:

- The server handles heavy work: storing billions of records, running search, matching Uber drivers to riders.
- The client (phone) handles light work: showing the screen, animations, capturing the user's tap.

This split lets a tiny phone feel like it has unlimited power.

Key Properties of the Client-Server Model

There are four properties you must remember:

The client always starts the conversation. A normal server cannot just push messages whenever it wants. The phone must ask first. We will see how WebSocket changes this later in this chapter.

The server is shared. One server may handle requests from millions of phones at the same time. So the server must be fast and stable.

Communication uses a protocol. Both sides must agree on a language. Most apps use HTTP or HTTPS as the language.

Chapter 3: Mobile-Specific Constraints

A mobile app is not just a "small website." It runs on a tiny computer that the user carries in a pocket. That tiny computer has many limits — limits that a server in a data center never has to worry about. If you ignore these limits, your app will be slow, drain the battery, crash often, and get bad reviews on the Play Store and App Store.

In this chapter, you will learn the **four big constraints** that shape every mobile design decision:

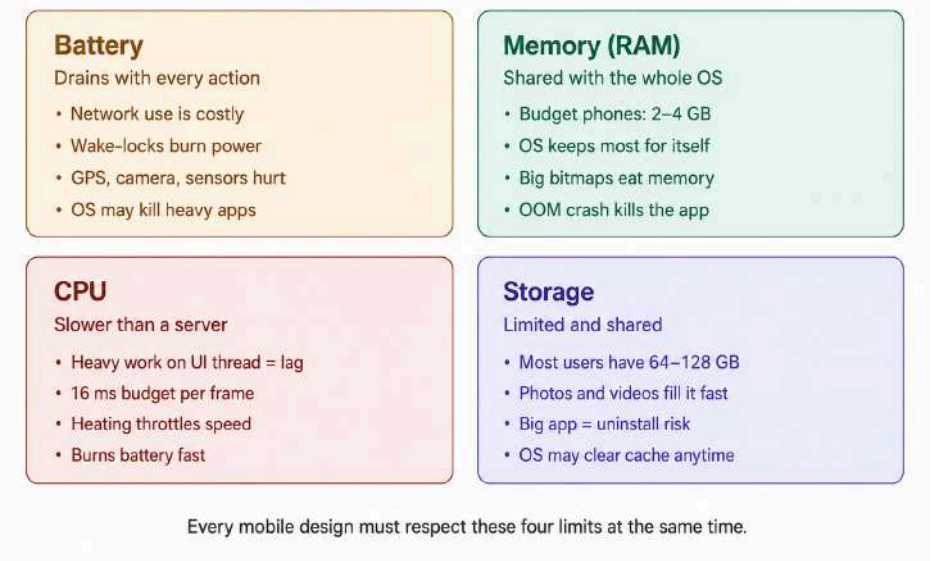
1. Limited resources — battery, memory, CPU, storage.
2. Unreliable networks — 2G, 3G, 4G, 5G, Wi-Fi, offline.
3. App lifecycle — foreground, background, killed.
4. Why mobile system design is fundamentally different from backend.

By the end of this chapter, you will think like a real mobile engineer — someone who designs for the worst phone, the worst network, and the worst moment, not just the perfect lab conditions.

3.1 Limited Battery, Memory, CPU, and Storage

A server in a data center has unlimited power, racks of RAM, dozens of CPU cores, and terabytes of disk. A phone has none of this.

Let's look at each limit one by one.



Battery — The Most Sacred Resource

A user's battery is precious. If your app drains battery fast, the user will uninstall it. They may not even know your app is the culprit, but the OS will tell them: "Your app used 25% of battery yesterday." That number is the kiss of death.

What burns battery on a phone:

The radio (Wi-Fi, mobile data) is the biggest battery drinker. Every network request wakes up the radio. Even after the request is done, the radio stays warm for several seconds (called the "tail energy"). So 10 small network calls drain much more battery than 1 big call with the same total data.

GPS is the second biggest drinker. Continuous GPS for 1 hour can drain 10 to 20 percent of the battery. Use it carefully — turn it off when not needed.

The screen is huge too. Big animations, full brightness, and OLED bright pixels use a lot of power. Dark mode actually helps on OLED screens.

Wake-locks and background work kill battery silently. If your app keeps the CPU awake at night to sync data, the phone heats up and drains while the user sleeps. Then they blame your app.

Sensors like the accelerometer, gyroscope, and step counter cost power if you read them frequently.

Battery-Saving Design Rules

Every mobile engineer should follow these rules:

Batch your network calls. Instead of making 5 calls in a row, combine them into 1 if possible. The radio wakes up once instead of five times.

Use WorkManager (Android) or BGTaskScheduler (iOS) for background sync. These tools batch and schedule tasks together with the OS to save battery. They also wait for good conditions: charging, idle, on Wi-Fi.

Cancel work when the user leaves the screen. If the user opens the profile screen and then immediately goes back, cancel the network call you started. Use coroutine cancellation, RxJava disposables, or Combine's `cancellable`.

Chapter 4: Mobile-Specific Constraints

If Chapter 3 was about the rules of the game (battery, memory, network, lifecycle), Chapter 4 is about how to organize your code to play that game well. Architecture patterns are not just academic theory — they are practical solutions to real problems that every mobile engineer faces.

In this chapter, we will cover:

1. The five main patterns: MVC, MVP, MVVM, MVI, and Clean Architecture.
2. How to choose the right pattern in an interview.
3. The most important mental model in mobile development: layered thinking — Presentation, Domain, and Data.

By the end, you will be able to explain why MVVM beat MVP, why MVI is rising, and why every senior engineer talks about "layers" instead of just folders.

4.1 Why We Need Architecture Patterns at All

Imagine writing a 100,000-line app where every Activity has all the code:

- UI rendering.
- API calls.
- Database queries.
- Business logic.
- Validation.
- Navigation.

This is called a "God Activity" or "Massive View Controller." It has many problems:

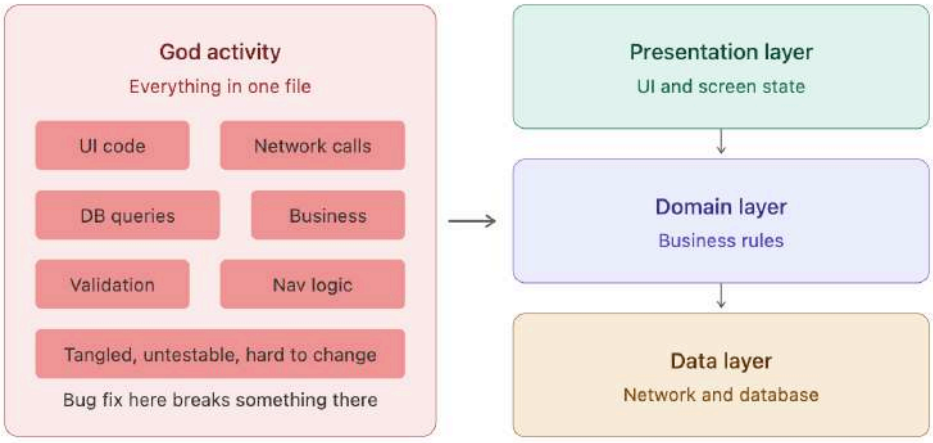
The code is impossible to test. You cannot test the business rules without spinning up the whole UI.

It is impossible to reuse. The login logic is tied to LoginActivity, so you cannot reuse it on a tablet, web, or smart TV.

Bugs spread fast. Changing one piece breaks five others.

Multiple developers cannot work on the same screen at once. Merge conflicts everywhere.

Architecture patterns solve these problems by **separating concerns**. Each part of the code has one clear job. Different parts talk to each other through clean contracts (interfaces). When done right, code becomes testable, reusable, and maintainable.



Each layer has one job. Clean and testable.

Now let's go through the five most common patterns, in the order they evolved.

4.2 MVC — Model-View-Controller

MVC is the oldest and most famous architecture pattern. It came from web frameworks in the 1980s and was adopted in early iOS development.

The three parts:

Model holds the data and business rules. For example, a `User` class with name, email, and a method `validateEmail()`.

View displays the UI. For example, an XML layout with a TextView and Button. It is "dumb" — just shows what it is told.

Controller handles user actions and updates the model and view. For example, a UIViewController on iOS or an Activity on Android (in classic Android).

Chapter 5: Networking Layer Design

The networking layer is the heart of every mobile app. It connects the phone to the rest of the world. If your network code is sloppy, your app will be slow, drain battery, drop data, and frustrate users. If it is well-designed, your app feels fast and reliable — even on a weak 3G signal in a crowded train.

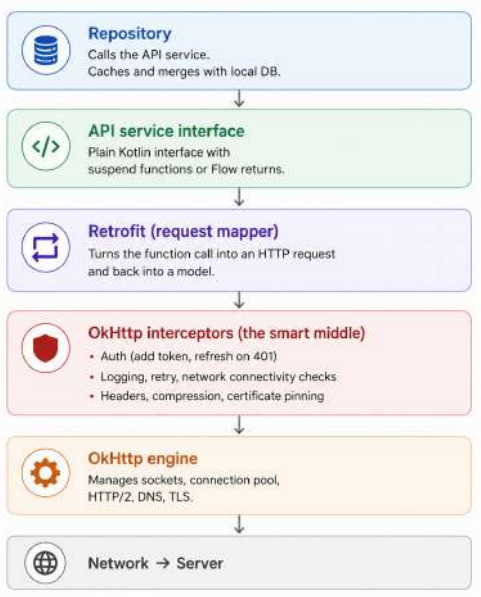
In this chapter, we will go deep into the four main parts of a production-grade networking layer:

1. Designing a robust API client.
2. Error handling, retries, and timeouts.
3. Pagination — offset, cursor, and keyset strategies.
4. Image and file upload and download.

By the end, you will know how to design and discuss a networking layer that survives real-world conditions, and you will sound like an engineer who has shipped real apps.

5.1 Designing a Robust API Client

The API client is the **front door** of your data layer. Every network call goes through it. A good API client is built in clear, separate parts — each with one job. This is what an interview-level design looks like.



The Six Pieces of a Good API Client

A production-grade API client should have these six pieces, each clearly separated:

1. The HTTP engine. This is the low-level library that actually opens sockets and sends bytes. On Android, it is OkHttp. On iOS, it is `URLSession` (or AFNetworking / Alamofire built on top). You don't usually write engine code — you configure it.

2. The serializer. This turns Kotlin/Swift objects into JSON and back. On Android, Moshi or kotlinx.serialization (faster) or Gson (older). On iOS, `Codable`. You define data classes and let the serializer do the work.

3. The API service interface. This is a plain interface that lists what your app can ask the server. With Retrofit (Android), you write something like:

```
interface NewsApiService {
    @GET("posts")
    suspend fun getPosts(@Query("page") page: Int):
Response<List<PostDto>>

    @POST("posts")
    suspend fun createPost(@Body post: NewPostRequest):
Response<PostDto>
}
```

The interviewer wants to see this — a clean, testable interface that does not leak implementation details.

4. Interceptors. These are the smart middle layer that runs before and after every request. They handle cross-cutting concerns: adding the auth token, logging, retries, error handling. Each interceptor has one job. We will cover them in detail below.

5. The data classes (DTOs). These match the JSON shape coming from the server. They live in the data layer. Never let DTOs leak into the UI — convert them to domain models first.

6. The repository. The repository wraps the API service and adds caching, error handling, and combining with local data. The rest of the app talks to the repository, never to the API service directly.

Chapter 6: Storage and Caching

In Chapter 5, we focused on talking to the server. But every great mobile app does something equally important: it **remembers** things. Photos you have already loaded. Messages you have already received. Profiles you have already opened. The faster your app remembers, the faster it feels.

This is the world of storage and caching. It is also one of the deepest topics in mobile system design — and one of the favorites of interviewers. Most candidates know caching exists. Few can explain it well.

In this chapter, we will cover:

1. Memory cache vs. disk cache — the two layers and why you need both.
2. Local databases — SQLite, Room, Core Data, and Realm — when to pick which.
3. Cache eviction — how to throw out old data without losing what matters.
4. Cache invalidation — the famously hard problem of knowing when cached data is stale.

By the end, you will be able to design a storage layer that is fast, correct, and respectful of the user's storage and battery.

6.1 Why Caching Matters at All

Before diving into types, let's understand why caching is such a big deal on mobile.

Imagine an app that fetches every screen from the server every single time. Open the home feed → 2 seconds wait. Tap a profile → 1 second wait. Go back → another 1 second wait, even though you just had that data. This app feels broken even on a fast connection.

Now imagine the same app with smart caching. Open the home feed → instant from cache, fresh data appears 1 second later. Tap a profile → instant. Go back → instant. The app feels fast because the user almost never waits.

Caching gives you four wins:

Speed. Reading from RAM is millions of times faster than a network call. Reading from disk is thousands of times faster.

Battery savings. Every avoided network call saves the radio from waking up. We saw in Chapter 3 how expensive the radio is.

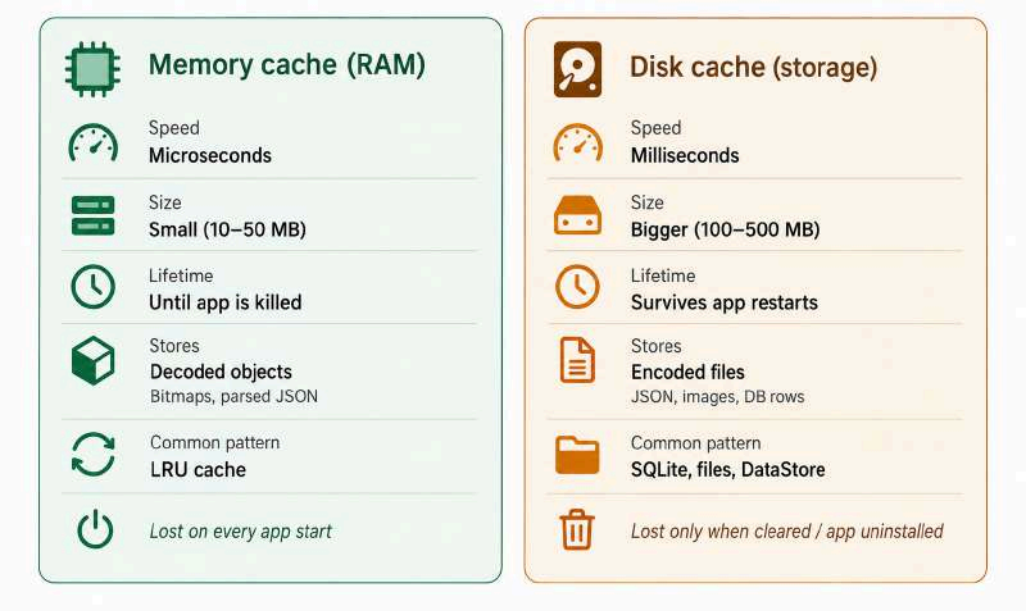
Data savings. Users on metered plans pay for every megabyte. A cached image costs them nothing.

Offline support. When the user has no network, cached data is the only thing that works. This is the foundation of offline-first apps.

The cost of caching is **complexity**. You must decide what to cache, where to cache it, when to evict it, and when to consider it stale. Get any of these wrong and the app shows old data, runs out of memory, or fails to update when the user expects.

6.2 Memory Cache vs. Disk Cache

There are two main caching layers on a mobile device. Think of them as two different "memories" with different speeds, sizes, and lifetimes.



Chapter 7: Offline-First Design

In Chapter 6, we learned how to cache data so the app feels fast. In this chapter, we go further: how to build an app that **works fully even when there is no network at all**. This is the offline-first philosophy, and it is the gold standard for modern mobile apps.

WhatsApp lets you read messages, type replies, and view photos without the internet. Notion lets you write a page on a flight. Google Maps lets you navigate a downloaded area in a tunnel. Gmail lets you compose mail offline and sends it when you land.

These are not lucky features — they are the result of careful design. In this chapter, we will cover:

1. Why every modern app needs offline support.
2. Optimistic updates — how to make actions feel instant.
3. Sync queues and replay logic — how to deliver actions later.
4. Conflict resolution basics — what to do when the user and server disagree.

By the end, you will be able to design an offline-first app and explain it confidently in any system design interview.

7.1 Why Every Modern App Needs Offline Support

Many engineers think offline support is a "nice-to-have." It is not. For any app used in the real world, offline support is essential. Here is why.

The User's Reality

Imagine a user's day:

The morning commute on a crowded metro — signal drops 5 times in 30 minutes. A long flight with no Wi-Fi. The basement of a mall. A village with patchy 3G. An office Wi-Fi that says "connected" but the captive portal blocks everything.

In all these situations, the user still wants to use your app. They want to read the news they downloaded earlier. They want to type a message that will go out the moment the signal returns. They want to see their saved bookmarks.

If your app shows a spinning wheel or "No internet" page in these moments, the user closes the app and forgets about it. If your app keeps working, the user trusts it. Trust is what builds five-star reviews and long-term retention.

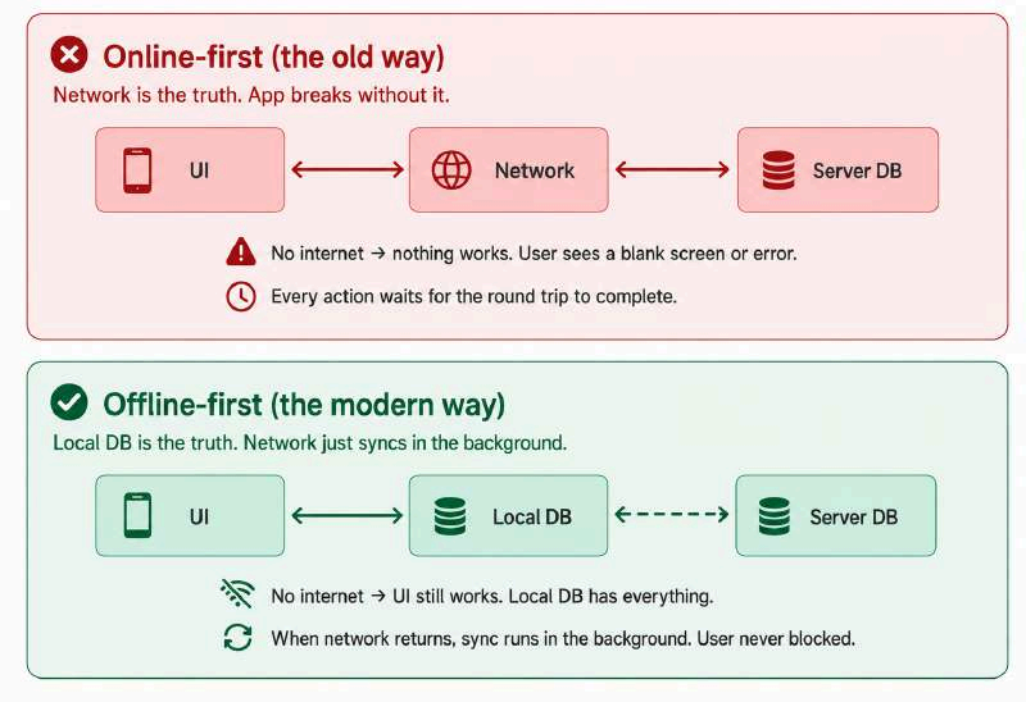
What "Offline-First" Actually Means

Offline-first is **not** "we added an offline mode at the end." It is a **philosophy** that flips the usual approach:

In the old "online-first" model: the network is the **truth**. The app fetches data from the server and shows it. If the network is gone, the app is gone.

In the "offline-first" model: the **local database is the truth**. The app reads from local storage. The network is just a way to keep the local DB in sync with the server. If the network is gone, the app keeps working with whatever is local.

This single shift — making the local DB the source of truth — changes everything about how you design data flow, networking, sync, and UI.



Chapter 8: Real-Time Updates

So far we have built apps that fetch data when the user asks for it. But many modern apps need something more: data that **updates by itself**, in real time. Chat messages arriving without a refresh. Live cricket scores changing every ball. An Uber map showing the driver's car move. A trading app where prices flicker every second.

These features all need **real-time updates** — a way for the server to push new data to the phone without the user having to ask. There are several ways to do this, each with strengths and weaknesses. Choosing the right one is a classic mobile system design interview question.

In this chapter, we will cover:

1. The four real-time techniques: polling, long polling, WebSocket, and Server-Sent Events (SSE).
2. Push notifications — how FCM (Android) and APNs (iOS) work and when to use them.
3. A decision guide — when to use what, with realistic examples.

By the end, you will know exactly which tool to reach for in any real-time feature, and how to defend your choice in an interview.

8.1 The Real Problem — Why is "Real-Time" Hard?

In Chapter 2, we saw that the client always starts the conversation in HTTP. The phone asks; the server answers. This works for "give me the news headlines." It does not work for "tell me the moment a new message arrives."

There are two ways to solve this:

The phone keeps asking. "Anything new? Anything new? Anything new?" Simple but wasteful.

The phone and server stay connected, and the server can speak whenever it wants.

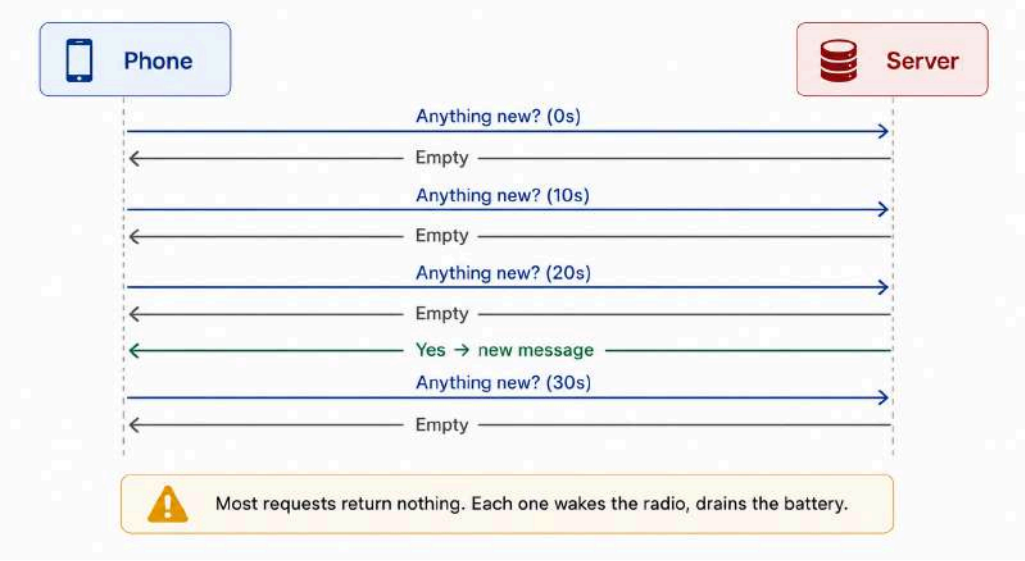
Different real-time techniques are different points on this spectrum. Each makes a trade-off between **latency**, **battery use**, **server cost**, and **complexity**. There is no single best choice — only the best choice for your app.

Let's explore each one.

8.2 Polling — The Simplest (and Worst) Real-Time

Polling is the most basic technique. The app makes a normal HTTP request every few seconds: "Anything new?" If yes, the server returns the new data. If no, it returns "nothing new."

```
Every 10 seconds:
  GET /messages?since=lastMessageId
  → { "messages": [] }   // most of the time, nothing new
  → { "messages": [...] } // sometimes, new messages
```



Why Polling Hurts on Mobile

Polling has serious problems on mobile devices:

Wasted requests. Most of the time the answer is "nothing new." Out of 100 requests, maybe 95 come back empty. Each one still cost data, battery, and server resources.

Bad latency. If the polling interval is 10 seconds, a new message can take up to 10 seconds to appear. For chat or live scores, that is too slow.

Battery drain. Every request wakes the radio. As we saw in Chapter 3, the radio is the biggest battery drinker. Polling every 10 seconds means waking the radio constantly.

Chapter 9: Performance Essentials

Performance is what separates a great mobile app from a forgettable one. Two apps can have the same features, the same design, the same backend — and one feels like silk while the other feels like sandpaper. The difference is performance.

Performance is also one of the most popular topics in mobile system design interviews. Interviewers love asking "How would you make the app faster?" or "The launch is slow — how would you fix it?" If you have a clear, structured answer, you stand out instantly.

In this chapter, we will go deep into the four performance areas that matter most:

1. App launch optimization — how to make the app open fast.
2. Smooth scrolling and 60 FPS — how to make every gesture feel buttery.
3. Image loading and memory management — how to handle media without crashing.
4. Battery and network efficiency — how to be a good citizen of the device.

By the end, you will speak the same language as senior performance engineers and know exactly where to look when something feels slow.

9.1 Why Performance Matters More Than You Think

A small reminder before we start. Mobile users have very little patience:

- If the app takes more than **2 seconds** to launch, many users abandon and never come back.
- If scrolling drops below **60 frames per second**, users feel "lag" even if they cannot name it.
- If the app drains **5% of battery per hour**, the OS will warn the user and many will uninstall.
- If a single screen costs **20 MB of data** to load, users on metered plans will quietly stop opening the app.

Performance is not a vanity metric. It is the foundation of trust. A fast app feels well-built; a slow app feels broken. Users do not need to read your code to know which one yours is.

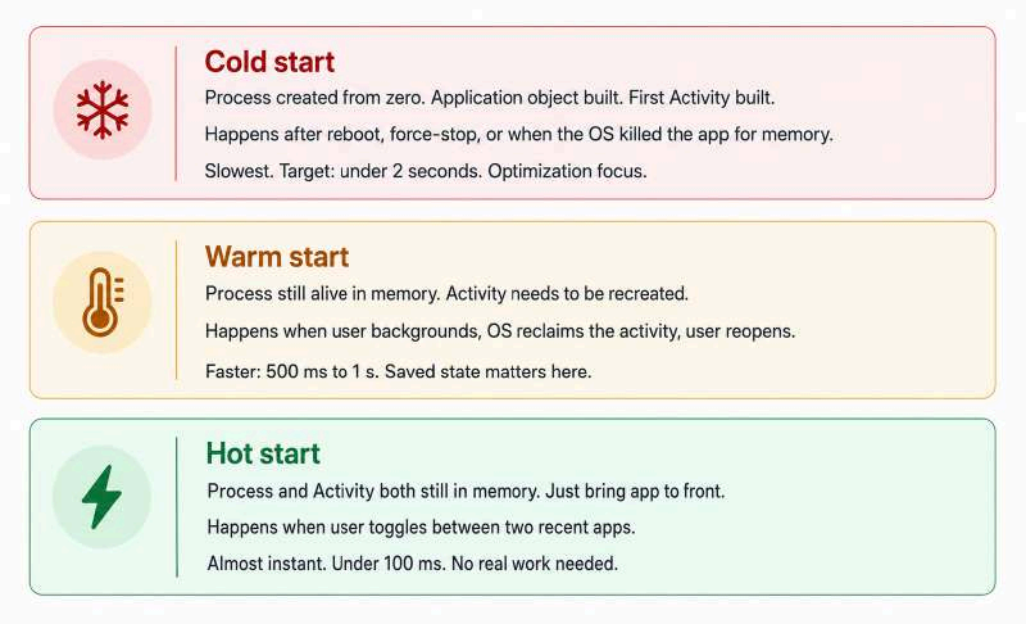
Now let's look at each area.

9.2 App Launch Optimization

The launch is the first impression of your app, every single day. If the user taps the icon and waits 4 seconds for a blank screen, they already feel disappointed. If the app appears in 700 milliseconds, they feel happy.

Three Kinds of Launch

Before optimizing, know what kind of launch you are dealing with. They have very different characteristics.



Cold start is the one we optimize hardest. Warm start mostly takes care of itself if cold start is good. Hot start is essentially free.

What Happens During a Cold Start

Let's see what the OS and your app actually do during a cold start. This understanding is exactly what an interviewer wants to hear.

Chapter 10: Security Basics

A mobile app lives on a device the user fully controls. The user can root or jailbreak the phone, decompile your APK or IPA, sniff your network traffic, and read your app's local files. From a security view, **the device itself can be hostile**.

This is very different from a server, where you control the hardware, the OS, the network, and the access. On mobile, you must assume the worst and design defensively.

Security is a deep topic — entire books are written on it. In this chapter, we will cover the four areas a mobile system design interviewer cares about most:

1. HTTPS and certificate pinning — securing the network.
2. Token storage — keeping credentials safe on the device.
3. Data encryption at rest — protecting local data.
4. Common mobile security pitfalls — the mistakes engineers actually make.

By the end, you will know what to do, what not to do, and how to talk about it with confidence in interviews.

10.1 The Mobile Security Mindset

Before any specific technique, you need the right mental model. Here are three rules that govern everything in mobile security.

Rule 1: Never trust the client. The phone is in the user's hand. They can change anything in your app. Validation in the app is for UX only; the real validation must happen on the server. If the server says "user X can withdraw $100," the app must call the server to confirm — never decide it locally.

Rule 2: Assume the device can be compromised. Some users root or jailbreak. Some install malware. Some give their phone to a friend. Your app must protect data so that even on a compromised device, the damage is limited.

Rule 3: Defense in depth. Don't rely on one safeguard. Use HTTPS *and* certificate pinning. Encrypt tokens *and* require biometric unlock for sensitive actions. If one layer fails, the others still help.

With these in mind, let's go layer by layer.

10.2 HTTPS and Certificate Pinning

The first attack surface is the **network**. Every byte that leaves the phone goes through Wi-Fi, the cell tower, ISPs, and the internet before reaching your server. Anyone on the path can try to read or modify your traffic.

HTTPS — The Baseline

HTTPS is HTTP over TLS. It does three things:

Encrypts the traffic so eavesdroppers see only random bytes.

Authenticates the server using a TLS certificate signed by a trusted authority. The phone verifies the certificate before sending any data.

Protects integrity so the data cannot be modified in transit without being detected.

This sounds great. But there is a hidden weakness, and it leads us to certificate pinning.

The Problem with Plain HTTPS

When the phone connects to `api.yourapp.com`, it expects a certificate signed by a **trusted Certificate Authority (CA)**. The phone has a list of about 100 trusted root CAs built into the OS. Any of these can sign a certificate for your domain — even ones you don't know about.

This creates risks:

Compromised or malicious CAs. A few have been hacked or have signed certificates they shouldn't have over the years. Once a CA mistakenly signs a certificate for `api.yourapp.com`, anyone with that certificate can impersonate your server.

User-installed CAs. A corporate device may have a corporate CA installed. A government may install a national CA. A malicious app may trick the user into installing a fake CA. Any of these allows a "man-in-the-middle" (MITM) to intercept HTTPS traffic.

Chapter 11: How to Approach a Mobile System Design Interview

You have spent the first ten chapters learning **what** to design — networking layers, caches, real-time updates, security. This chapter is about something different: **how to actually run an interview**. The technique. The structure. The way you talk and think for 45 to 60 minutes while a senior engineer judges you.

Two engineers can have the same knowledge and get very different outcomes from the same interview. The difference is **process**. A clear, repeatable approach helps you stay calm, cover the right things, avoid blind spots, and finish with confidence.

This chapter gives you exactly that. We will cover:

1. The **CRDDS framework** — five steps to navigate any mobile system design question.
2. How to recognize the kind of round you are in: **HLD-focused vs LLD-focused** — and how to adapt.

Once you internalize this, every system design question becomes a familiar shape, not a scary blank canvas.

11.1 Why Most Candidates Fail (Even Strong Ones)

Before learning the right way, let's understand the wrong ways. After watching many real interviews, the most common failure patterns are:

Jumping to a solution. The interviewer says "design Instagram." The candidate immediately starts drawing classes and choosing libraries. Five minutes later, they realize they did not even ask whether stories are in scope. They have to throw away half their work.

Drowning in detail. The candidate goes deep into one corner — say, the exact JSON shape of the feed API — and spends 25 minutes there. They never get to the cache, the offline behavior, or the architecture. The interviewer wants breadth and they showed only one narrow trench.

Treating it as a memory test. The candidate tries to recite every library and pattern they know, regardless of whether it fits. "I'd use Hilt and Coil and Coroutines and StateFlow and Retrofit and OkHttp and Gson and ProGuard

and Firebase and..." The interviewer learns nothing about how the candidate **thinks**.

Going silent. The candidate stops talking and starts thinking quietly. Two minutes pass. The interviewer has no idea what is going on. Even if the answer in their head is great, the interviewer cannot see it.

No structure. The candidate jumps from topic to topic randomly. They mention pagination, then security, then offline, then back to pagination. Hard to follow. The interviewer ends up confused.

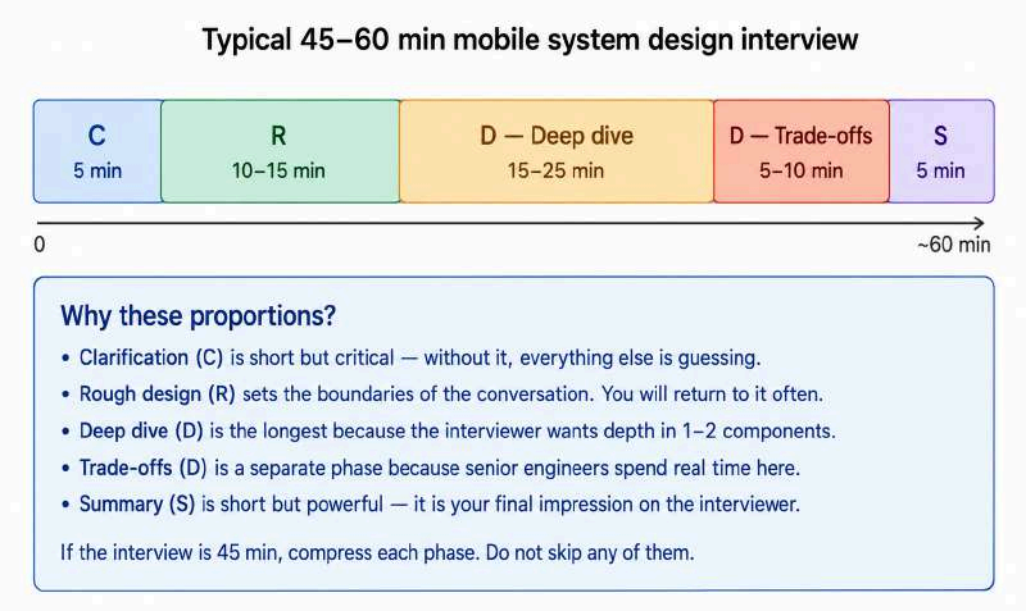
A good framework solves all of this. Let's build one.

11.2 The CRDDS Framework

CRDDS is a five-step process that fits a typical 45–60 minute mobile system design interview. The steps are:

C — Clarify Requirements R — Rough High-Level Design D — Deep Dive into Components D — Discuss Trade-offs S — Summarize and Handle Follow-ups

You will spend roughly the time below in each step:



Chapter 12: Asking the Right Clarifying Questions

In Chapter 11, you learned the CRDDS framework. The very first step — **C: Clarify Requirements** — is the most underrated step in mobile system design interviews. Most candidates rush past it in 60 seconds. Strong candidates spend 4 to 6 minutes here, and that single difference often decides the outcome.

Why? Because **everything you design after this depends on what you clarified**. Get this step right and the rest of the interview flows naturally. Get it wrong and you will design the wrong system beautifully — and fail.

In this chapter, we will go deep into the art of asking clarifying questions:

1. The difference between **functional** and **non-functional** requirements.
2. How to think about **scale** — users, data, geography.
3. How to handle **platform** questions — iOS, Android, or both.
4. How to clarify **online/offline expectations** — perhaps the most important mobile-specific question.
5. A **reusable question checklist** you can apply to any prompt.

By the end, you will be able to take any vague prompt — "design Instagram," "design Uber's mobile app," "design a podcast app" — and within 5 minutes, have a sharp, agreed-upon problem statement.

12.1 Why Clarification Is Where Most Candidates Lose Points

Before tactics, let's understand the stakes. Imagine the interviewer says "design an e-commerce app." Two candidates respond:

Candidate A says: "OK, e-commerce app. I'll start with the home screen, product list, cart, checkout. I'll use MVVM and Room and Retrofit." They start drawing. Five minutes in, they have the architecture for a generic e-commerce app.

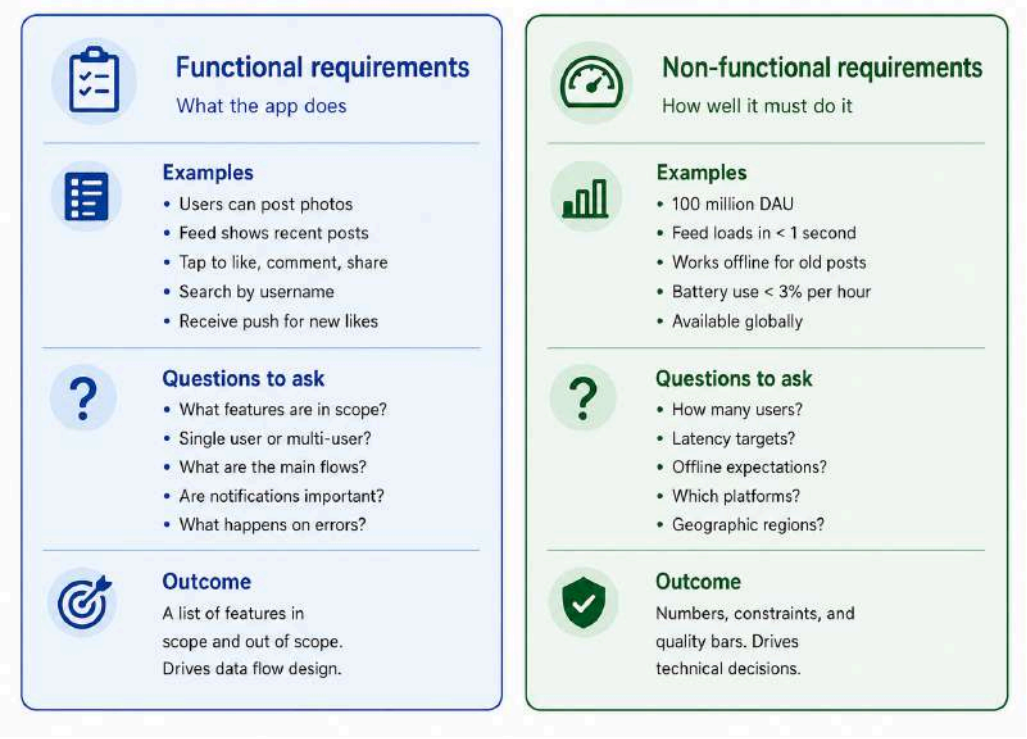
Candidate B says: "A few questions first. Are we focusing on the buyer's app or the seller's app? Do we support multiple languages and currencies? Is offline browsing important — like browsing wishlists on a flight? How important is real-time inventory? And roughly how big is this — 1 million DAU, 100 million? Indian market or global?"

In 90 seconds, Candidate B has uncovered five important constraints. These will shape every later decision. Candidate A is solving the wrong problem.

The interviewer is listening for **scoping skill** — the ability to take a vague request and turn it into a sharp, designable problem. This is the most senior skill in real engineering, and your interview is the proof you have it.

12.2 Functional vs. Non-Functional Requirements

The first lens to bring is the difference between two kinds of requirements. Once you can sort questions into these buckets, your clarification becomes structured and easy.



User Scale — DAU, MAU, and Peaks

Daily Active Users (DAU) is the headline number — how many unique people open the app on a typical day. Most product discussions use DAU.

Monthly Active Users (MAU) captures occasional users. The DAU/MAU ratio is a measure of engagement: 0.5 means half your monthly users come in daily, which is very high.

Chapter 13: Drawing the High-Level Design (HLD)

In Chapter 12, you mastered clarification — the first step of CRDDS. Now we move to the next step: **drawing the high-level design**. This is where the interviewer first sees your structured thinking made visual.

A good HLD diagram does three things at once:

1. It shows the interviewer that you understand the **shape** of a real mobile app.
2. It anchors the conversation. Every later question — "what about offline?" "how do you handle errors?" — points back to a specific box on your diagram.
3. It communicates faster than words alone. A clean diagram with arrows is worth a thousand sentences.

In this chapter, we will cover:

1. The **standard mobile architecture diagram** — a template you can adapt to almost any prompt.
2. The four core layers — **UI, Business, Data, Network** — and what lives in each.
3. How to actually **draw** on a whiteboard or virtual tool: order, labels, narration.
4. What HLD covers and what it leaves for **LLD** later.

By the end, you will be able to walk into any interview, listen to the prompt, clarify, and within 10 minutes draw a clear, professional HLD that sets up the rest of the discussion.

13.1 What HLD Is — and What It Is Not

Before drawing anything, get clear on what HLD is for.

HLD is a bird's-eye view. It shows the major components of the app and how they connect. Think city map, not street map.

HLD is for the architecture conversation. It is the picture the interviewer points at while asking questions. It is the picture you point at while answering.

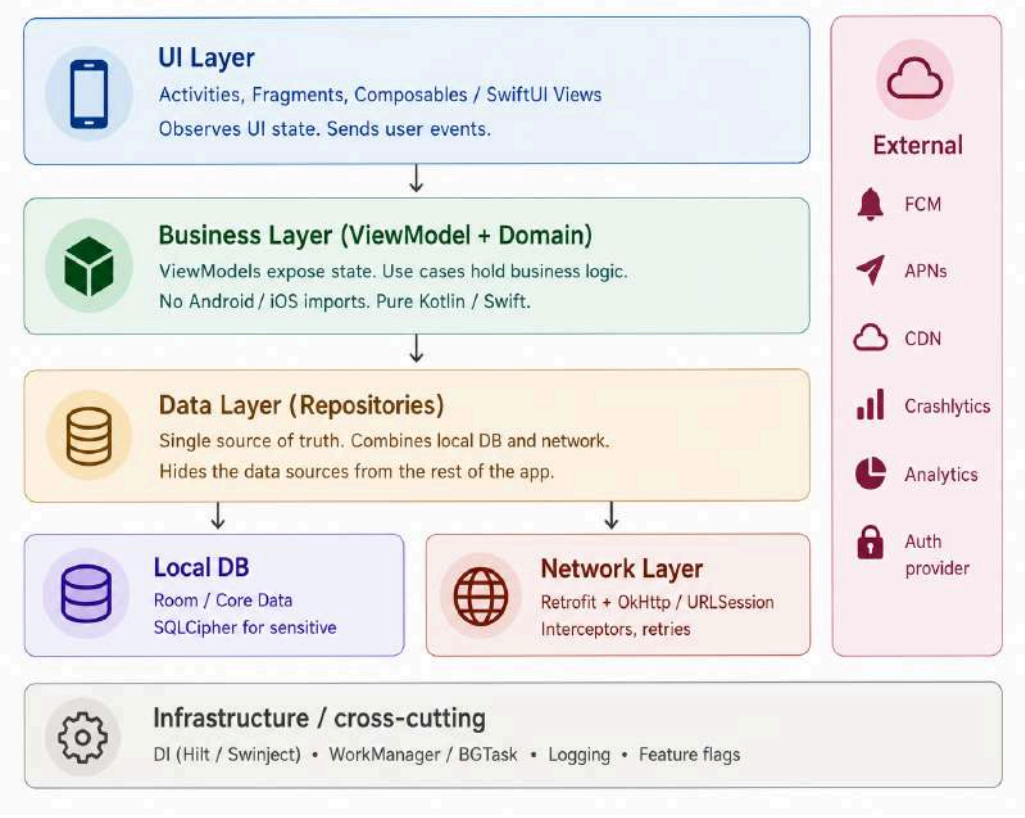
HLD is not code. No method names. No exact class signatures. No JSON shapes. Those come in LLD (Chapter 14) if the interview goes that deep.

HLD is not a finished product. It is a starting frame for a discussion. It will evolve as the interviewer asks "what about X?" and you add boxes or refine them.

Think of HLD as the **table of contents** of your design. Once it is on the board, every later topic finds its place.

13.2 The Standard Mobile Architecture Diagram

Almost every modern mobile app — Instagram, Uber, WhatsApp, Spotify, Notion — has the same basic shape under the hood. Different apps add their own components on top, but the skeleton is shared. Memorize this skeleton.



This is your mental template. Let's break each layer down.

13.3 The UI Layer

The UI layer is what the user actually sees and touches. It is the topmost layer, closest to the screen.

Chapter 14: Component-Level Design (LLD)

In Chapter 13, you built a clean HLD diagram. The interviewer is now staring at your boxes. After a moment, they pick one and say:

"Let's go deeper into the [Repository / Player / Download Manager / Image Loader]."

This is the deep-dive phase. The boxes from your HLD are zoomed-out shapes. Now you must zoom in and design the **inside** of one box: the classes, the interfaces, the methods, the patterns, and the edge cases.

Many candidates know the architecture but freeze here. They wave their hands, mention a few class names, and never reach the depth interviewers are looking for. This chapter teaches you exactly how to handle the deep-dive phase like a senior engineer.

We will cover:

1. How to **pick the right component** to deep-dive into.
2. How to use **class diagrams, interfaces, and design patterns** to communicate design.
3. How to apply **SOLID principles** in the moment without sounding academic.
4. How to **discuss trade-offs** the way senior engineers actually do.
5. How to **bring up edge cases** without being asked.

By the end, you will be able to take any component from your HLD and confidently design its internals in 15 to 25 minutes.

14.1 What "Deep Dive" Really Means

Before we start picking and drawing, let's get clear on what the deep-dive is.

In HLD, you said "I'll have a `FeedRepository` that combines the API and Room." That was one box.

In LLD, you say:

- "Here's the `FeedRepository` interface with these three methods."
- "Here's the implementation, which depends on `FeedApi`, `FeedDao`, and `MemoryCache`."

- "When `getFeed()` is called, here's the order of operations."
- "I'd use the **Repository pattern** plus **stale-while-revalidate** plus **single-flight refresh**."
- "Edge case: what if two coroutines call `getFeed()` at the same time? Here's how I handle it."

The deep dive is where **theory meets reality**. You move from "I'll use a repository" to "here's exactly how the repository works." The interviewer is checking whether you can actually build the box you drew, or whether you only know the buzzwords.

A good deep dive feels like a real design review at work. The interviewer follows your reasoning, asks pointed questions, and walks away thinking "yes, this person could ship this."

14.2 Picking the Right Component

The interviewer will often choose a component for you. But sometimes they say "what would you like to dive deeper into?" Knowing how to pick well is itself a skill.

The Three Things to Look For

Pick a component that has all three of these:

1. Genuine complexity. A box where there is real design work to do. Lots of moving parts, decisions, edge cases. Not "ButtonComponent."

2. Relevance to the prompt. A box that captures the heart of the app. For a chat app, the Message Repository or the WebSocket service is at the heart. For a feed app, the Repository and pagination. For a media app, the Player service.

3. Personal strength. A box you have shipped or studied deeply. Don't pick something you only vaguely know about. The interviewer will probe, and shallow knowledge collapses fast.

The intersection of those three is your safest pick.

Chapter 15: Common Mistakes to Avoid

You now have everything you need to perform well in a mobile system design interview. The framework. The technical knowledge. The patterns. The vocabulary. The instincts.

But here is a hard truth: **most candidates fail not because they lack knowledge, but because they make small, avoidable mistakes**. The same mistakes, over and over, across thousands of interviews. Strong technical candidates lose offers because they jump to the solution. Calm candidates over-engineer. Senior engineers misjudge the level of the interview.

This chapter is about those mistakes. We will go through the five most common patterns, why each one hurts you, and how to avoid them. Reading this chapter is not enough — you must internalize each pattern until your reflex changes.

We will cover:

1. Jumping to the solution without clarifying.
2. Ignoring mobile constraints.
3. Over-engineering.
4. Not discussing trade-offs.
5. Mixing up HLD and LLD levels.

By the end, you will know exactly what to **not** do — and that is sometimes more valuable than knowing what to do.

15.1 Why Mistakes Matter More Than Strengths

Before we dive in, a moment of perspective.

Interviewers are pattern-matching. They have seen hundreds of candidates. They have built mental categories: "the candidate who jumps to the solution," "the candidate who knows libraries but no trade-offs," "the candidate who over-engineers everything." When you make a mistake, the interviewer does not just note it down — they pattern-match you into a category and start watching for confirmation.

Conversely, when a candidate **avoids** the common mistakes, the interviewer's brain registers something rare: "this one is structured and self-aware." That single impression often counts more than any specific technical answer.

So this chapter is not just a list of don'ts. It is a list of **first impressions to avoid creating**. Each mistake below is a category you do not want to land in.

15.2 Mistake 1: Jumping to the Solution Without Clarifying

This is the most common mistake — by far. The interviewer says "design Instagram." The candidate says "OK, I'll use MVVM with Compose, Room for the local DB..." and starts drawing.

What happened? The candidate skipped the entire **C** of CRDDS. They are now solving a problem they did not define.

Why It Feels Right (But Is Wrong)

Two reasons candidates jump:

Excitement. You finally have a real prompt. Your brain wants to start producing answers. The longer you "just talk," the more anxious you feel. Diving in feels like making progress.

Misreading the cue. "Design Instagram" sounds clear. You assume the interviewer wants a generic Instagram, so you start sketching.

Both feelings are wrong. The interviewer wants to see whether you can take a vague prompt and **scope it like a senior engineer**. Diving in tells them you cannot.

Chapter 16: CASE STUDIES (Real Interview Questions)

1. Designing a Chat App

This is the chapter every mobile engineer waits for. **"Design WhatsApp"** is the single most common mobile system design interview prompt in the world. Variants include "Design Telegram," "Design Messenger," "Design iMessage," "Design Slack DMs," and "Design Discord chat." All of these have the same skeleton, with small variations.

If you can confidently design a chat app end-to-end, you can pass almost any mobile system design interview. Why? Because chat pulls together **every** core mobile concept: real-time networking, offline-first storage, optimistic UI, sync queues, conflict resolution, push notifications, media handling, security, and scale.

This chapter is longer and deeper than the others on purpose. We will walk through a **complete 60-minute interview** simulation using the CRDDS framework — clarification, rough HLD, deep dive, trade-offs, summary. By the end, you will have a mental movie of exactly what to say, what to draw, and when to pause.

We will cover:

1. Step 1 — Understanding the problem and establishing design scope.
2. Step 2 — Choosing the right communication protocol.
3. Step 3 — High-level client architecture.
4. Step 4 — Sending and receiving messages (the heart of the design).
5. Step 5 — Message storage and sync.
6. Step 6 — Read receipts, typing indicators, and presence.
7. Step 7 — Media sharing.
8. Step 8 — Offline support and conflict resolution.
9. Step 9 — Push notifications.
10. Step 10 — Wrap-up and follow-up topics.

Let's begin.

1.1 Step 1: Understanding the Problem and Establishing Design Scope

Apply Chapter 11 and Chapter 12. The interviewer says **"Design WhatsApp"** or **"Design a chat app."** Your first move is always the same: clarify.

Why This Step Decides Everything

Chat apps look similar from the outside, but the internals vary dramatically based on requirements:

- A 1-to-1 chat for 1 million users is very different from a 100-million-user group chat platform.
- An offline-first chat is architecturally different from an online-only chat.
- A chat with end-to-end encryption (Signal) is very different from one without (older Messenger).
- A chat with voice/video calls needs a completely different stack from text-only.

Without clarifying these, you risk designing the wrong thing.

The Clarifying Conversation

Imagine this dialogue. Notice how the candidate proposes assumptions instead of waiting for everything to be spelled out.

> ***You:*** *"Before I start, a few questions to scope this properly.*
>
> *First, scope of features — are we focusing on 1-to-1 text chat, or also group chat, voice/video calls, stories, and channels?"*
>
> ***Interviewer:*** *"Let's focus on 1-to-1 text chat. Skip groups, calls, and stories."*
>
> ***You:*** *"Got it. Are media attachments — photos, videos, voice notes — in scope?"*
>
> ***Interviewer:*** *"Yes, photos and short videos. Skip voice notes for now."*
>
> ***You:*** *"Offline support — should the user be able to read old messages and send new ones with no internet?"*

2: Designing a News Feed App

After designing a chat app, the news feed is the next canonical mobile system design problem. **"Design Twitter's mobile app," "Design Instagram's home feed," "Design Facebook's news feed,"** and **"Design the LinkedIn feed"** all reduce to the same core challenge: an infinite, personalized stream of mixed content that users scroll through with engagement actions on every post.

The news feed is loved by interviewers for one reason — it touches almost every mobile concern at once:

- **Pagination at massive scale** — billions of posts, you load 20 at a time.
- **Smooth 60 FPS scrolling** — with photos, videos, and rich text.
- **Caching strategy** — what to keep, what to evict, when to refresh.
- **Optimistic engagement** — likes, comments, shares must feel instant.
- **Media handling** — photos at the right size, videos with adaptive quality.
- **Offline reads + offline writes** — users browse on planes, like posts in tunnels.
- **Real-time elements** — new posts banner, live like counts.
- **Rich post content** — bold text, mentions, hashtags, links.

By the end of this chapter, you will have a complete, interview-ready design and know exactly how to defend every decision under interviewer pressure.

We will follow the **CRDDS framework** and cover ten steps:

1. Understanding the problem and establishing design scope.
2. API design — endpoints, pagination, data models.
3. High-level client architecture.
4. The feed rendering pipeline.
5. Cursor-based pagination in depth.
6. Local storage and offline-mode strategy.
7. Optimistic engagement — likes, comments, bookmarks.
8. Rich post content — HTML vs Markdown vs native.
9. Smooth scrolling at scale — view recycling, image/video tuning.
10. Wrap-up and follow-up topics.

Let's begin.

2.1 Step 1: Understanding the Problem and Establishing Design Scope

The interviewer says **"Design a news feed app"** or **"Design Twitter's mobile app."** The first 5 minutes are for clarification — apply Chapter 11 and Chapter 12 here.

The Clarifying Conversation

Imagine the dialogue. Notice how a senior candidate proposes assumptions instead of waiting for everything to be spelled out.

> ***You:*** *"Before I dive in, a few questions. First — what platforms are we targeting? Mobile only, or web too?"*
>
> ***Interviewer:*** *"Mobile only. Both Android and iOS, but center on Android."*
>
> ***You:*** *"Got it. What are the post content types — text only, or also photos and videos?"*
>
> ***Interviewer:*** *"Mixed. Text, photos, short videos, and links with previews."*
>
> ***You:*** *"Are we building the entire app or just the home feed?"*
>
> ***Interviewer:*** *"Focus on the home feed, post detail view, and create-post flow. Skip profiles, search, DMs, and notifications."*
>
> ***You:*** *"What about engagement — likes, comments, shares, bookmarks?"*
>
> ***Interviewer:*** *"All four — likes, comments, shares, bookmarks."*
>
> ***You:*** *"Is the feed algorithmic or chronological?"*
>
> ***Interviewer:*** *"Server-ranked algorithmic. The client just gets a pre-ordered list."*
>
> ***You:*** *"Real-time updates? Should new posts arrive without a refresh, or is pull-to-refresh enough?"*

3: Designing a Photo Sharing App

After mastering chat apps and news feeds, the next canonical mobile system design problem is **"Design Instagram"** — a photo-and-video sharing app. Variants include **"Design TikTok," "Design Snapchat," "Design Pinterest,"** and **"Design YouTube Shorts."**

While Instagram shares the feed pattern from Chapter 18, it pushes mobile engineering much harder in three areas:

- **Media creation tools** — the camera, photo editor, and filter pipeline are first-class features, not afterthoughts.
- **Heavy media uploads** — every post carries one or more photos or videos through unreliable mobile networks.
- **Stories — ephemeral content** — 24-hour disappearing posts with their own data model, viewer surface, and progress UX.

In this chapter, we will design Instagram end-to-end using CRDDS, covering ten steps:

1. Understanding the problem and establishing design scope.
2. API design — endpoints for posts, stories, and media uploads.
3. High-level client architecture with three major surfaces.
4. The camera and media capture pipeline.
5. The photo editor — filters, crop, and adjustments.
6. Reliable media upload with chunks and resume.
7. Stories — ephemeral content design.
8. The feed and explore surfaces.
9. Offline behavior and sync queue.
10. Wrap-up and follow-up topics.

This chapter is the most camera-heavy and media-heavy of the book. By the end, you will know how to defend every decision in a 60-minute Instagram-style interview.

3.1 Step 1: Understanding the Problem and Establishing Design Scope

The interviewer says **"Design Instagram"** or **"Design a photo sharing app."** Apply Chapter 11 and Chapter 12 here.

The Clarifying Conversation

You: "Before diving in, a few questions. First — what surfaces are we designing? Just the photo feed, or also stories, reels, DMs, search?"

Interviewer: "Focus on three: the home feed with posts, stories at the top, and the create flow with camera and basic editing. Skip reels, DMs, search, and shopping."

You: "Got it. For the create flow — are we capturing photos only, or also videos?"

Interviewer: "Both. Single photos, photo carousels (multiple photos in one post), and short videos up to 60 seconds."

You: "What about editing — basic filters only, or full adjustments like brightness, contrast, crop?"

Interviewer: "Filters, crop, brightness, contrast. Skip stickers, drawing tools, and AR filters."

You: "Engagement on posts — likes, comments, shares, saves?"

Interviewer: "Likes and comments. You don't need to design shares or saves from scratch — assume they follow the same pattern as likes."

You: "For stories — what's the lifecycle and what kinds of stories?"

Interviewer: "Stories live for 24 hours, then disappear. Photos and videos, with simple text overlay support. Skip story replies and reactions for now."

You: "Scale and platforms?"

Interviewer: "2 billion MAU. Both Android and iOS, center on Android. Global, with a heavy presence in regions with patchy networks."

You: "Offline support?"

4: Designing a Ride-Sharing App

After mastering chat, news feed, and Instagram, the next canonical mobile system design problem is **"Design Uber."** Variants include **"Design Lyft," "Design Ola," "Design DoorDash's driver app,"** and **"Design any real-time location-based service."**

Ride-sharing apps are the most operationally intense mobile applications ever built. While Instagram pushes mobile engineering hard on **media**, ride-sharing pushes it hard on **real-time, location, maps, and reliability under stakes**. A bug in Instagram delays a photo. A bug in Uber strands a rider in the rain at 11 PM or sends a driver to the wrong address.

This chapter is the most demanding in the book on purpose. We will explore engineering challenges that don't appear elsewhere:

- **Two distinct apps** — rider and driver — that share a backend but have very different mobile architectures.
- **Continuous location streaming** — the driver app reports location every few seconds, around the clock.
- **Real-time matching** — finding the right driver in under 10 seconds among thousands of candidates.
- **Maps and routing** — the visual centerpiece of both apps, with offline tile caching and turn-by-turn navigation.
- **The trip state machine** — a long-lived workflow with explicit states and recovery for every failure.
- **Safety and reliability** — driver safety features, emergency SOS, trip recording, and graceful degradation when networks fail.

We will follow CRDDS and cover ten steps:

1. Understanding the problem and establishing design scope.
2. The two-app architecture — rider and driver.
3. API design and the two-protocol strategy.
4. High-level rider app architecture.
5. High-level driver app architecture.
6. Real-time location streaming.
7. The trip state machine.
8. Maps, routing, and offline tiles.
9. Reliability, safety, and edge cases.
10. Wrap-up and follow-up topics.

By the end, you will know how to defend every decision in a 60-minute Uber-style interview.

4.1 Step 1: Understanding the Problem and Establishing Design Scope

The interviewer says **"Design Uber"** or **"Design a ride-sharing app."** Apply Chapter 11 and Chapter 12 here.

The Clarifying Conversation

> **You:** "Big problem space. Let me clarify the scope. Are we designing both the rider app and the driver app, or just one?"
>
> **Interviewer:** "Both. They share a backend so let's see how the two apps cooperate."
>
> **You:** "Got it. Are we covering the full ride lifecycle — request, match, pickup, in-trip, dropoff, payment — or focusing on a slice?"
>
> **Interviewer:** "The full lifecycle. Skip ride scheduling for later, focus on on-demand rides only."
>
> **You:** "Ride types? UberX only, or also pool, premium, deliveries, etc.?"
>
> **Interviewer:** "Just standard solo rides. Skip pool, skip deliveries."
>
> **You:** "Are we designing the maps and turn-by-turn navigation ourselves, or assuming we plug into Google Maps / Mapbox?"
>
> **Interviewer:** "Plug into a third-party maps SDK for rendering, routing, and navigation. Focus on how we integrate, not how we build maps."
>
> **You:** "Payment flow — design it, or assume we plug into Stripe or similar?"
>
> **Interviewer:** "Plug into a payment provider. Focus on when payment is triggered, error handling, not the PCI surface."
>
> **You:** "Scale and geography?"

5: Designing a Video Streaming App

After mastering ride-sharing, the next canonical mobile system design problem is **"Design Netflix"** or **"Design YouTube."** Variants include **"Design Hotstar," "Design Disney+," "Design Amazon Prime Video,"** and **"Design Twitch."**

Video streaming apps push mobile engineering hard in directions we have not yet explored:

- **Adaptive bitrate (ABR) streaming** — playback quality changes mid-video based on network speed.
- **Content protection (DRM)** — premium content must be encrypted, both in transit and on disk.
- **Smart prefetching** — start playback in under 2 seconds, never buffer mid-playback.
- **Offline downloads** — watch downloaded content on planes, with license expiry.
- **Long sessions** — users watch 30-90 minute videos, sometimes multiple in a row.
- **Cross-device sync** — resume from where you left off, across phone, tablet, TV.
- **Background audio playback** — listening to a music video while using another app.
- **Picture-in-picture** — watching while messaging.

In this chapter, we will design a Netflix-style video streaming app end-to-end using CRDDS. We will cover ten steps:

1. Understanding the problem and establishing design scope.
2. Streaming protocols and DRM strategy.
3. High-level client architecture.
4. The home screen and content discovery.
5. The player surface and adaptive streaming.
6. Smart prefetching for instant playback.
7. Offline downloads and license management.
8. Cross-device sync and resume points.
9. Background audio and picture-in-picture.
10. Wrap-up and follow-up topics.

By the end, you will know how to defend every decision in a 60-minute Netflix-style interview.

5.1 Step 1: Understanding the Problem and Establishing Design Scope

The interviewer says **"Design Netflix"** or **"Design a video streaming app."** Apply Chapter 11 and Chapter 12 here.

The Clarifying Conversation

> **You:** "Big problem space. Let me clarify the scope. Are we designing the full Netflix experience or focusing on specific surfaces?"
>
> **Interviewer:** "Focus on three surfaces: the home screen with content rows, the video player itself, and the downloads feature. Skip live TV, kids profiles, and the social features."
>
> **You:** "Got it. What kinds of content — short videos like YouTube, long-form like Netflix, or both?"
>
> **Interviewer:** "Long-form. Movies and TV episodes, typically 20 to 120 minutes."
>
> **You:** "Premium content with DRM, free with ads, or both?"
>
> **Interviewer:** "Premium content only, DRM-protected. Subscription model. No ads."
>
> **You:** "Resolution levels?"
>
> **Interviewer:** "Up to 1080p for this design. Skip 4K and HDR — those are extensions."
>
> **You:** "Offline downloads?"
>
> **Interviewer:** "Yes. Users can download episodes for offline viewing. Downloads must respect license expiry."
>
> **You:** "Cross-device resume?"
>
> **Interviewer:** "Yes. If a user pauses on their phone and opens on tablet, they should resume from where they left off."
>
> **You:** "Scale?"

What Makes This Book Different?

Most System Design books focus heavily on backend systems and distributed infrastructure.

This book focuses specifically on:

✅ Mobile-first system design
✅ Real-world mobile engineering challenges
✅ Interview-focused explanations
✅ Practical architecture discussions
✅ Offline-first and real-time systems
✅ Performance & scalability optimization
✅ Production-ready mobile architectures
✅ HLD & LLD interview preparation
✅ Real app architecture breakdowns

Real-World Case Studies Included in the Full Book

The complete edition includes detailed Mobile System Design case studies such as:

- Chat Application (WhatsApp / Telegram)
- News Feed System (Twitter / Facebook)
- Photo Sharing App (Instagram)
- Ride-Sharing App (Uber / Ola)
- Video Streaming App (Netflix / YouTube)
- Food Delivery App (Swiggy / Zomato)
- E-Commerce App (Amazon / Flipkart)
- Stock Trading App (Groww / Zerodha)
- Hotel Reservation App (Airbnb / OYO)

Why Mobile System Design Matters

Modern mobile developers are expected to do much more than build UI screens.

Today's engineers must understand:

- Scalability
- Architecture
- Performance Optimization

- Offline-first systems
- Real-time communication
- Security & reliability
- Production-ready engineering practices

This book helps bridge the gap between traditional app development and modern scalable mobile engineering.

👨‍💻 About the Author

Anand Gaur is a Tech Lead with 9+ years of experience in Mobile Application Development, specializing in Android, Kotlin,Swift, Flutter, React Native and scalable mobile architectures. He has built production-ready applications and worked extensively on performance optimization, modern Android development, and real-world mobile system design.

Full Book Information

Book Title: Cracking the Mobile System Design Interview
Pages: 670+ Pages
Level: Beginner to Advanced
Focus: Practical & Interview-Focused Mobile System Design

Get the Full Book Access

The complete edition of **Cracking the Mobile System Design Interview** contains 650+ pages of practical architecture discussions, scalable mobile systems, interview frameworks, and real-world case studies.

Purchase the Full Book:

- Direct Purchase from the Author
- Topmate
- Gumroad
- Google Play Books

🙌 Thank You

Thank you for reading this preview edition.

Keep learning. Keep designing. Keep building scalable systems.

CRACKING THE MOBILE SYSTEM DESIGN INTERVIEW

A practical, hands-on guide to help Android developers master mobile system design and ace high-level interviews with confidence.

This book goes beyond theory. It teaches you how to think like an engineer, design scalable systems, make the right trade-offs, and communicate your ideas clearly in interviews.

INTERVIEW-FOCUSED APPROACH
Real interview questions with complete system design solutions and explanations.

REAL-WORLD CASE STUDIES
Design popular apps and systems used in real-world production.

SCALABLE & PRODUCTION-READY DESIGNS
Learn to design systems that are scalable, reliable, and performant.

PRACTICAL & EASY TO FOLLOW
Clear explanations, diagrams, and structured approach for better understanding.

ABOUT THE AUTHOR

ANAND GAUR

Anand Gaur is a Tech Lead with 9+ years of experience in Mobile Application Development, specializing in Android, Kotlin, and scalable mobile architectures. He has built production-ready applications and worked extensively on performance optimization, modern Android development, and real-world mobile system design.

OTHER BOOKS BY ANAND GAUR

Scan to connect with me!

PRACTICAL REAL-WORLD PROJECTS

PERFORMANCE & COST OPTIMIZATION

SECURITY & PRIVACY BEST PRACTICES

ISBN : 978-93-5913-811-4

9 789359 138114

MRP: XXX