48 · System Design

Flutter Practice Notes

Contents

48 · System Design

Introduction

This module covers **system design from the mobile/Flutter perspective**: how to design an app-plus-backend system end-to-end — clarifying **requirements** (functional + non-functional), defining **client-server contracts & data flow** (APIs, pagination, real-time, sync), choosing a **caching/offline strategy**, and reasoning about **scalability + trade-offs** — plus how to **approach a mobile system-design interview** ("design the Instagram feed," "design a chat app"). It synthesizes the whole handbook (architecture, networking, offline-first, storage, performance, security) into a design discipline, capped by a capstone.

Why this module exists

Senior/architect roles require designing systems, not just implementing screens — and **mobile system design is its own discipline** distinct from backend system design: the client is constrained (battery, memory, flaky network, offline), the contract with the server is the crux, and caching/offline/sync dominate the design. Interviews and real projects both demand a structured approach: clarify requirements, sketch the client-server data flow, pick storage/caching/offline strategies, and justify trade-offs. This module teaches that structured, mobile-centric design thinking.

Module map

#	File	Topic	Level
1	01_system_design_fundamentals.md	What mobile system design is; requirements; the process	🔴
2	02_client_server_and_data_flow.md	API contracts, pagination, real-time, data flow & sync	🔴
3	03_caching_offline_and_scale.md	Caching strategy, offline-first, scalability & trade-offs	🔴
4	04_mobile_system_design_interview.md	Interview approach: frameworks, common prompts, communication	🟡
5	05_system_design_integration.md	Capstone: design a feed/chat app end-to-end	🔴

> **Cross-references:** Architecture band: 40–47. Networking: Module 16. Offline-first: Module 19. Database/caching: Module 20/Module 34. Performance: Module 21. Security: Module 37. Notifications/real-time: Module 32.

Prerequisites

The architecture band (40–47), 16 Networking, 19 Offline First, 20 Database, 21 Performance.

What you'll be able to do after this module

- Run a structured mobile system-design process from requirements to trade-offs.
- Define client-server contracts + data flow (pagination, real-time, sync) deliberately.
- Choose caching/offline strategies and justify them against constraints.
- Reason about mobile-specific scalability and trade-offs (battery/memory/network).
- Confidently approach a mobile system-design interview end-to-end.

Capstone

End-to-end design: Design a feed or chat app — clarify functional/non-functional requirements, define the client-server API contract + data flow (pagination/real-time), choose storage + caching + offline-sync strategies, address mobile constraints (battery/memory/flaky network), map it onto the app architecture (Clean/feature-first/MVVM), and present the trade-offs — as a written design doc + a whiteboard-style walkthrough.

Summary

Mobile system design is a structured discipline: clarify requirements, design the client-server contract + data flow, choose caching/offline/sync strategies, and justify trade-offs under mobile constraints (battery/memory/network/offline). It synthesizes the handbook into designing app systems end-to-end — for real projects and interviews alike.

System Design Fundamentals (Mobile Perspective)

> Mobile system design is **its own discipline**: unlike backend system design (scaling servers/DBs/load balancers), the mobile designer's focus is the **client and its contract with the server** under harsh constraints — **flaky/offline network, battery, limited memory, small screen, app-store rules, and an untrusted client**. The process is structured: **clarify requirements** (functional + **non-functional** — offline? real-time? scale? latency?), then design **data flow, storage/caching, sync, and client architecture**, justifying **trade-offs** at each step. The recurring theme: **the network is unreliable and the device is constrained**, so caching/offline/sync usually dominate the design.

Introduction

This file establishes what mobile system design *is*, how it differs from backend system design, the constraints that drive it, and the structured process to run — the frame for the contract/caching/interview files.

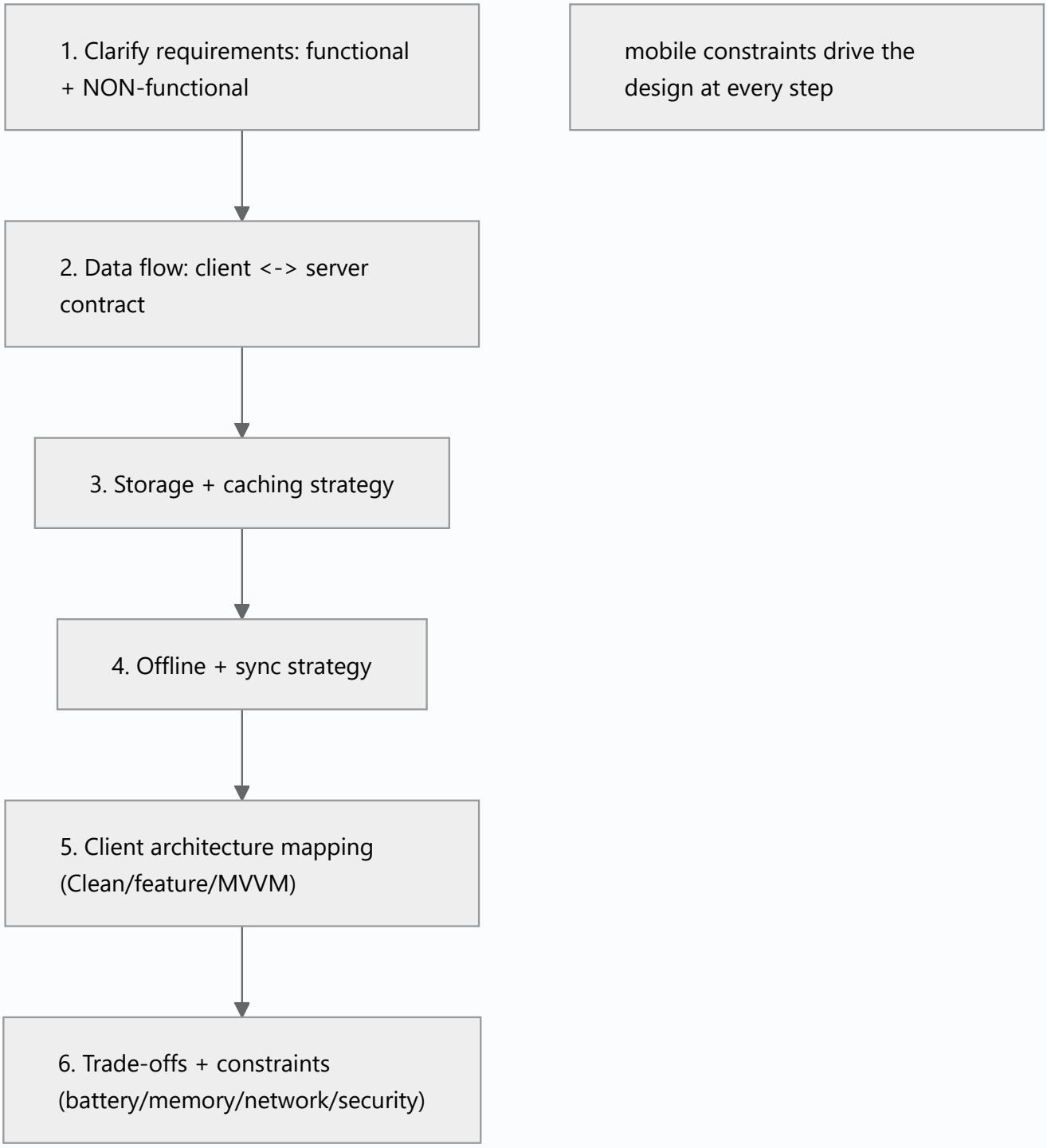
Why this concept exists

Engineers often approach "design X app" like a backend problem (servers, sharding) and miss the mobile essence: the hard parts are the **client-server contract, offline behavior, caching, sync, and device constraints**. A mobile-specific frame + a repeatable process ensures you design what actually matters for an app — and communicate it clearly in interviews and docs.

Real-world analogy

Backend system design is planning a **city's central infrastructure** (power plants, water mains — scale/throughput). Mobile system design is designing a **well-equipped expedition vehicle** that must operate **far from that infrastructure**: it needs onboard supplies (cache), the ability to keep going when disconnected (offline), fuel efficiency (battery), limited cargo (memory), and a reliable **radio protocol** to sync with base when in range (client-server contract). You design for the harsh field, not the data center.

Internal Working



- **Clarify requirements first (never skip)**:
  - **Functional**: what does it do? (feed of posts, send messages, like/comment, search).
  - **Non-functional (the differentiators)**: **offline** support? **real-time**? expected **scale** (users/data volume)? **latency/perf** targets? **consistency** needs? **security/privacy** (PII)? **platforms**? These shape the whole design far more than the feature list.
  - **Scope/assumptions**: state them; ask clarifying questions (in interviews, this is scored heavily).

- **Mobile constraints that drive design**:
  - **Unreliable network** (offline, flaky, slow, metered) → caching + offline-first + sync are usually central (Module 19).
  - **Battery + data** → minimize/batched network, push over poll, deferral (Module 33/Module 47).
  - **Limited memory/CPU** → pagination, list virtualization, bounded caches, isolates (Module 21).
  - **Untrusted client** → server-as-source-of-truth, no secrets on device (Module 37).
  - **App-store/platform** → size, permissions, background limits, review.
- **Backend vs mobile system design (know the difference)**:
  - **Backend**: horizontal scaling, load balancers, DB sharding/replication, caching layers, queues, CAP theorem — server throughput/availability.
  - **Mobile**: **client-server contract**, on-device storage/caching, offline/sync/conflict resolution, pagination/real-time protocols, client architecture, device constraints. You **treat the backend as a given API** and design the **client + contract** (unless asked to design both).
- **The process (repeatable)**: (1) requirements → (2) client-server **data flow + contract** (02_client_server_and_data_flow.md) → (3) **storage + caching** → (4) **offline + sync** (03_caching_offline_and_scale.md) → (5) **client architecture** mapping (Clean/feature-first/MVVM — Module 40/Module 44/Module 43) → (6) **trade-offs**. Iterate; there's **no single right answer** — justify choices against requirements/constraints.
- **Trade-offs are the point**: every choice (cache TTL, real-time vs poll, optimistic vs pessimistic, consistency vs availability) has costs; **articulating and justifying** them (not memorizing "the answer") is what senior design demonstrates.

Memory Representation

Not runtime — a **design artifact**: a requirements list (functional + non-functional), a data-flow/contract diagram, storage/caching/sync decisions, an architecture mapping, and a trade-off analysis. The artifact evolves as you iterate.

Compiler / Runtime / Engine / VM Behavior

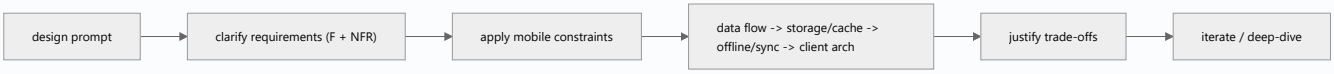
Not applicable — system design is a **conceptual/architectural** activity (the resulting code lives across the handbook's modules). Its "behavior" is the reasoning process + documented decisions.

Examples

```
Requirements clarification (design "a news feed app"):
  Functional:      list posts, infinite scroll, like, view detail, pull-to-refresh
  Non-functional:  offline read? YES (cache last feed) | real-time? partial (new-post badge)
                   scale? 1M users | latency? feed < 1s perceived | consistency? eventual
                   security? auth + no PII in logs | platforms? iOS+Android
  Assumptions:     backend REST API exists; images via CDN; ~20 posts/page
  -> these NFRs drive: pagination, cache-first read, offline snapshot, push for new-post signal

The mobile-design process (say it out loud in an interview):
  requirements -> data flow/contract -> storage+caching -> offline+sync -> client arch -> trade-offs
```

```
Mobile vs backend focus:
  Backend Q "design Twitter": fan-out, timeline service, sharding, caches, queues.
  Mobile  Q "design Twitter feed (client)": API contract, pagination, cache-first + offline snapshot,
           real-time new-tweet signal, image loading/memory, optimistic likes, client architecture.
```

Diagrams



Common Mistakes

Mistake	Why it's wrong	Fix
Jumping to a solution without requirements	Designs the wrong thing	Clarify functional + NFRs first
Ignoring non-functional requirements	NFRs drive the design	Ask offline/real-time/scale/latency/security
Treating it like backend system design	Misses the mobile essence	Focus on client + contract + offline/caching
Ignoring device constraints	Battery/memory/network bite	Design under network/battery/memory limits
Presenting one "answer" without trade-offs	Misses the point of design	Justify choices; state trade-offs
Designing the backend when asked for the client	Off-scope	Treat backend as a given API (unless asked)
No structure (rambling)	Poor communication	Follow the repeatable process

Best Practices

- **Clarify requirements first** — functional **and** non-functional (offline/real-time/scale/latency/consistency/security/platforms) + assumptions; ask questions.
- Focus on the **mobile essence**: **client-server contract, storage/caching, offline/sync, client architecture** under **device constraints** (network/battery/memory/untrusted client).
- Follow a **repeatable process** (requirements → data flow/contract → storage/caching → offline/sync → client arch → trade-offs) and **iterate**.
- **Justify trade-offs** at every step (no single right answer); treat the backend as a **given API** unless asked to design it too.

Performance

Not a runtime topic, but performance is a **first-class design input**: latency targets, memory/battery budgets, and network efficiency shape caching/pagination/sync choices (Module 21/Module 47). Good design bakes performance in from requirements, not after.

Advantages / Disadvantages

- **+** (Structured mobile design) designs what matters, communicates clearly, justifies trade-offs, produces implementable + resilient systems.
- **−** Requires broad synthesis (whole handbook), judgment (no formula), and disciplined communication; easy to over/under-scope.

Interview Questions

1. 🟢 **How does mobile system design differ from backend system design?** — Mobile focuses on the client + its contract with the server under device constraints (offline/battery/memory/untrusted), treating the backend as a given API; backend focuses on server scaling (LB/sharding/queues).
2. 🟢 **What's the first step in any system design?** — Clarify requirements — functional and especially non-functional (offline/real-time/scale/latency/security) — plus assumptions.
3. 🟡 **Why do non-functional requirements matter so much?** — They (offline, real-time, scale, consistency) drive the whole design (caching/sync/protocols) far more than the feature list.
4. 🟡 **What mobile constraints shape the design?** — Unreliable network (offline/flaky), battery/data, limited memory/CPU, untrusted client, and app-store/platform limits.
5. 🟡 **What's the repeatable process?** — Requirements → client-server data flow/contract → storage/caching → offline/sync → client architecture → trade-offs (iterate).
6. 🔴 **Why is there "no single right answer"?** — Every choice trades off (consistency vs availability, real-time vs battery, cache freshness vs offline) — design is about justified trade-

offs, not one answer.

7. 🔴 **When would you design the backend too?** — Only if the prompt asks; otherwise treat the backend as a given API and design the client + contract.

Senior Engineer Tips

- Spend real time clarifying requirements (especially offline/real-time/scale) before drawing anything; interviewers and projects both reward this, and it prevents designing the wrong system.
- Frame the problem as "client + contract under constraints," not "scale a backend"; the mobile signal is caching/offline/sync/device-limits reasoning, not sharding.
- Narrate the process and the trade-offs out loud; senior design is judged on structured thinking and justified choices, not on reciting a canonical architecture.

Architect Perspective

Mobile system design is the synthesis discipline of the whole handbook: it takes requirements + device constraints and composes networking, storage, caching, offline-first, performance, security, and the architecture band into a coherent, justified system centered on the client-server contract. Its distinguishing marks — network-unreliability and device-constraint reasoning, caching/offline/sync at the core, and trade-off articulation — are exactly what separate a senior mobile architect from an implementer. The process is repeatable; the value is judgment (02_client_server_and_data_flow.md, 03_caching_offline_and_scale.md, Module 19).

Summary

- Mobile system design = designing the client + its server contract under device constraints (network/battery/memory/untrusted) — distinct from backend (server scaling).
- Process: clarify requirements (functional + NFRs) → data flow/contract → storage/caching → offline/sync → client architecture → trade-offs; iterate.
- Caching/offline/sync usually dominate; treat the backend as a given API (unless asked); justify trade-offs (no single answer).

Revision Notes

- Mobile ≠ backend design: focus = client + server contract + storage/caching + offline/sync + client arch, under constraints (unreliable network, battery/data, memory/CPU, untrusted client, store limits). Backend = given API unless asked.
- Process: requirements (functional + NFR: offline/real-time/scale/latency/consistency/security/platforms) → data flow/contract → storage/caching →

offline/sync → client architecture (Clean/feature/MVVM) → trade-offs; iterate.

- No single answer → justify trade-offs; NFRs + constraints drive the design; performance is a design input.

Practice Questions

1. What questions do you ask to clarify a "design app X" prompt?
2. Why is mobile system design different from backend system design?
3. Which mobile constraints most shape the design, and how?

Coding Questions

1. Write a requirements list (functional + NFR + assumptions) for a given app.
2. Outline the mobile-design process for that app.
3. Identify the constraint-driven decisions (caching/offline/sync) it implies.

Mini Project

Requirements + process (Flutter/design): For a chosen app (e.g., a news feed or notes app), write a clarified requirements list (functional + non-functional + assumptions), identify the mobile constraints that shape it, and outline the design process (data flow → storage/caching → offline/sync → client arch → trade-offs) at a high level. Acceptance: functional + NFRs (offline/real-time/scale/latency/security) + assumptions stated; mobile constraints identified + linked to design decisions; repeatable process outlined; framed as client+contract (backend given); trade-off mindset evident.

Client-Server Contract & Data Flow

> The heart of mobile system design is the **client-server contract** — the API shape, protocols, and data flow between app and backend. Key decisions: the **API style** (REST vs GraphQL vs gRPC/WebSocket), **pagination** (cursor vs offset — cursor wins for feeds), **real-time** delivery (push/WebSocket/SSE vs polling), the **data shape** (tailored to the screen vs generic + client assembly), and how data **flows in and out** (fetch → cache → render; write → optimistic → sync). Design the contract to be **efficient over flaky networks** (minimal round-trips, right-sized payloads), **paginated**, and **evolvable** (versioned, additive).

Introduction

This file covers designing the contract + data flow: API style, pagination, real-time vs polling, payload shaping, and the read/write flows. It's the "how app and backend talk" layer of the design process (01_system_design_fundamentals.md), building on networking (Module 16).

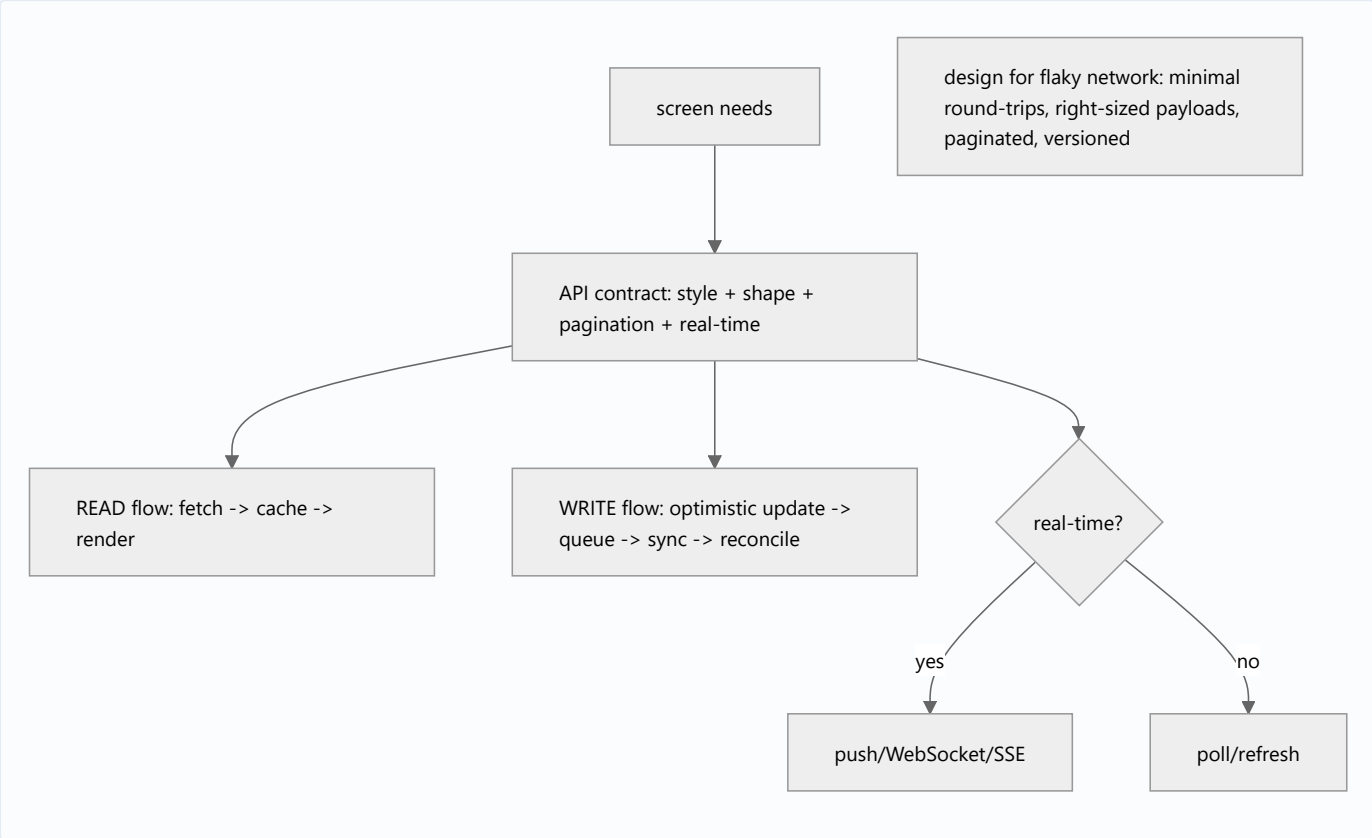
Why this concept exists

Over a mobile network, the contract determines perceived performance, battery/data use, and resilience. Poor choices (chatty APIs, offset pagination on a live feed, polling for real-time) cause jank, drain, and bugs. Designing the contract deliberately — for the screen's needs, minimal round-trips, and correct pagination/real-time — is where mobile system design earns its keep.

Real-world analogy

The contract is the **radio protocol between an expedition and base camp**: you choose the **channel type** (a chatty voice line = REST, a precise data burst = GraphQL/gRPC, an always-open line = WebSocket), send **minimal, well-formed messages** (right-sized payloads — bandwidth is precious), request **in manageable batches** (pagination), and decide whether base **calls you when news breaks** (push) or you **check in periodically** (polling). A bad protocol wastes battery and drops messages in bad weather (flaky network).

Internal Working



- **API style** (choose per needs):
  - **REST**: simple, cacheable, ubiquitous; can be **chatty** (multiple calls per screen) or **over/under-fetch**.
  - **GraphQL**: **client requests exactly the fields it needs** (great for varied mobile screens, reduces over-fetch/round-trips); more server/tooling complexity.
  - **gRPC**: efficient binary, typed contracts, streaming; great for performance/typed contracts, less browser-friendly.
  - **WebSocket/SSE**: for **real-time** streams (chat, live updates).
  - Often a **mix** (REST/GraphQL for CRUD + WebSocket for real-time).
- **Pagination** (essential for lists/feeds):
  - **Offset/page** (`?page=2&size=20`): simple but **breaks on live data** (items shift → duplicates/skips) and is slow deep in.
  - **Cursor/keyset** (`?after=<cursor>`): **stable for live feeds**, efficient — **preferred** for infinite-scroll feeds (Module 21).
  - Return `nextCursor` / `hasMore` ; the client requests pages on scroll; cache pages.
- **Real-time vs polling**:
  - **Push (FCM/APNs)** for out-of-app events (Module 32); **WebSocket/SSE** for in-app live streams (chat, presence, live scores); **polling** only when real-time isn't needed (battery cost — use long intervals/conditional GETs).

    - Design **which events are real-time** vs **pull-on-demand** (e.g., new-post *badge* via push, full content on open).
- **Payload/data shape**:
    - **Tailor to the screen** (BFF/aggregated endpoints or GraphQL) to minimize round-trips, **or** generic endpoints + client assembly (more calls). Right-size payloads (no over-fetching huge objects for a list row).
    - Use **DTOs** mapped to domain entities at the boundary (Module 40); images via **CDN** with sized variants.
- **Read flow**: request → (cache-first check) → network → **cache** → map DTO→entity → render; show cached data immediately, refresh in background (stale-while-revalidate — 03_caching_offline_and_scale.md).
- **Write flow**: **optimistic update** (update UI + local store) → **queue** the mutation (outbox) → **sync** to server → **reconcile** (confirm/rollback on failure); use **idempotency keys** so retries don't duplicate (Module 19/Module 31).
- **Efficiency + resilience** (mobile-critical): **minimize round-trips** (batch/aggregate), **right-size payloads**, **compress** (gzip), support **conditional GET/ETag** (skip unchanged — Module 34), **retry with backoff** on failure (Module 38), and **time out** hung requests.
- **Evolvability**: **version** the API (or additive-only changes), tolerate unknown fields, and keep the contract stable for shipped clients (old app versions linger) — treat it as a **published contract** (Module 45).
- **Security**: TLS + pinning, auth tokens, no secrets client-side, server-authoritative (Module 37).

Memory Representation

The contract is a spec (endpoints/messages/schemas). At runtime: DTOs on the wire → entities in the app; pages cached; the write outbox holds queued mutations; real-time streams push events into app state.

Compiler / Runtime / Engine / VM Behavior

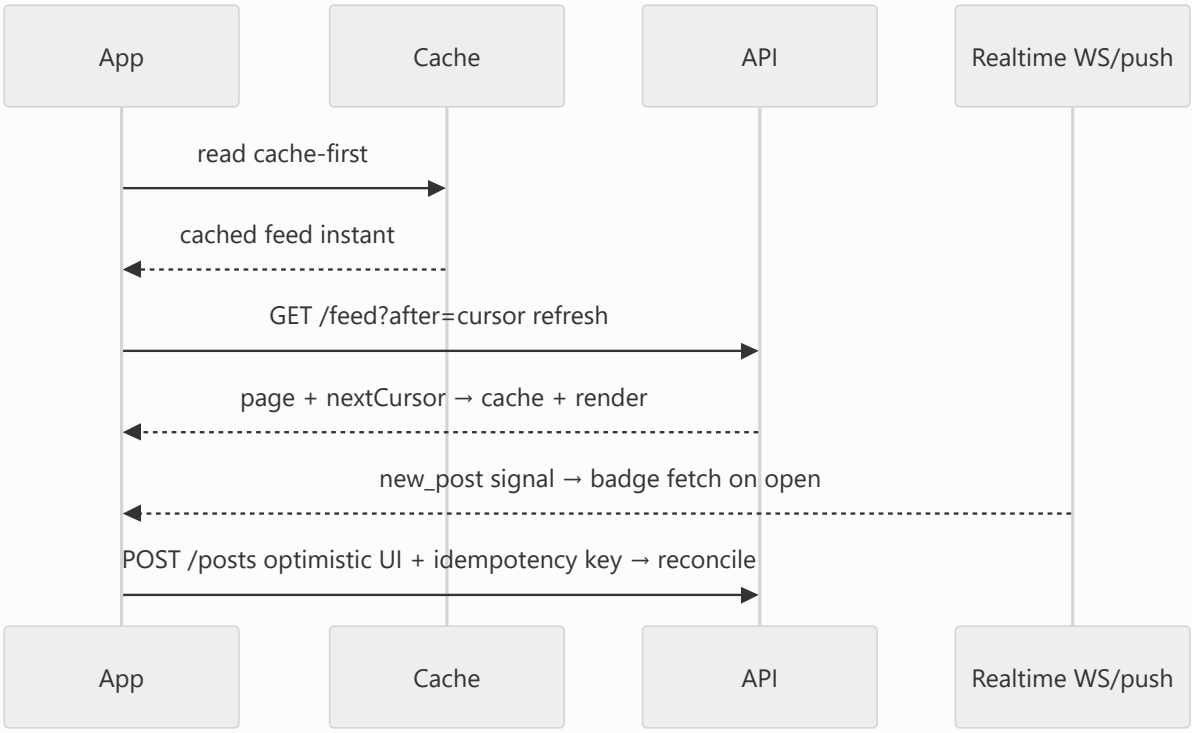
Design-level (the implementation lives in networking/offline modules). Runtime characteristics you *design for*: round-trip latency, payload size, real-time event delivery, and sync/reconcile behavior — all shaped by the contract choices above.

Examples

```
Feed contract (design):
  GET /feed?after=<cursor>&limit=20  -> { items:[PostDto...], nextCursor, hasMore }   [cursor pagination]
  POST /posts (Idempotency-Key: <uuid>) { text, mediaId } -> PostDto                  [optimistic +
idempotent write]
  WS /events  -> {type:'new_post', postId} (badge signal; fetch content on open)      [real-time signal,
not full payload]
  Images: https://cdn.example.com/media/<id>?w=640  (sized variants)                  [CDN, right-sized]
  ETag/If-None-Match on /feed for unchanged pages (304)                               [conditional GET]
```

```
Style decision matrix:
  varied screens, avoid over/under-fetch, many field combos -> GraphQL
  simple CRUD, cacheable, ubiquitous                        -> REST
  high-perf typed contracts / streaming                     -> gRPC
  in-app live updates (chat/presence/live)                  -> WebSocket/SSE
  out-of-app events                                         -> push (FCM/APNs)
  -> often REST/GraphQL + WebSocket + push together
```

Diagrams



Common Mistakes

Mistake	Why it's wrong	Fix
Offset pagination on a live feed	Duplicates/skips as data shifts	Cursor/keyset pagination
Polling for real-time	Battery/data drain, lag	Push/WebSocket/SSE; poll only if acceptable
Chatty API (many calls/screen)	Latency/battery over flaky net	Aggregate/BFF/GraphQL; minimize round-trips
Over-fetching huge payloads for lists	Bandwidth/memory	Right-size payloads (list vs detail shapes)
No idempotency on writes	Duplicate mutations on retry	Idempotency keys
Breaking the API for shipped clients	Old apps break	Version/additive; treat as published contract
No conditional GET/compression	Wasted bandwidth	ETag/If-None-Match + gzip
No retry/backoff/timeout	Hangs/failures on flaky net	Backoff retries + timeouts

Best Practices

- Choose the **API style per needs** (REST/GraphQL/gRPC + WebSocket/push, often mixed); design the **payload shape to the screen** (minimize over/under-fetch + round-trips).
- Use **cursor pagination** for feeds; **push/WebSocket** for real-time (not polling); design **which events are real-time vs pull-on-demand**.
- Design **read (cache-first + refresh)** and **write (optimistic + queue + sync + reconcile, idempotent)** flows; make the contract **efficient over flaky networks** (compression, conditional GET, retry/backoff, timeouts).
- Keep the contract **evolvable** (versioned/additive, tolerate unknown fields — shipped clients linger) and **secure** (TLS/pinning/auth, server-authoritative).

Performance

The contract *is* perceived performance: fewer round-trips + right-sized payloads + cursor pagination + cache-first reads = fast, resilient UX on flaky networks; conditional GET + compression save bandwidth; push beats polling on battery. Poor contracts are the top cause of sluggish, battery-hungry apps (Module 21).

Advantages / Disadvantages

- **+** (Well-designed contract) fast/resilient over flaky networks, battery/data-efficient, correct pagination/real-time, evolvable, secure.
- **−** Requires deliberate design + backend cooperation (BFF/GraphQL), real-time infra (WebSocket) adds complexity, versioning discipline, optimistic/sync complexity.

Interview Questions

1. 🟢 **Which pagination for a live feed, and why?** — Cursor/keyset — it's stable as data shifts (offset causes duplicates/skips) and efficient deep in the list.
2. 🟢 **Real-time: push/WebSocket vs polling?** — Push (out-of-app) / WebSocket/SSE (in-app streams) for real-time; polling only when real-time isn't needed (battery/data cost).
3. 🟡 **REST vs GraphQL vs gRPC for mobile?** — REST (simple/cacheable/chatty), GraphQL (client picks fields → fewer round-trips/over-fetch), gRPC (efficient typed/streaming); often mixed with WebSocket for real-time.
4. 🟡 **Design the write flow for offline resilience.** — Optimistic UI + local write → queue (outbox) → sync with idempotency key → reconcile (confirm/rollback) on reconnect.
5. 🟡 **How do you make the contract efficient over flaky networks?** — Minimize round-trips (aggregate/BFF/GraphQL), right-size payloads, compress, conditional GET (ETag), retry+backoff, timeouts.
6. 🔴 **How do you keep the API evolvable for shipped clients?** — Version or additive-only changes, tolerate unknown fields, keep it stable (old app versions persist) — treat it as a published contract.
7. 🔴 **How do you decide which events are real-time vs pull-on-demand?** — By UX need + cost: send lightweight real-time *signals* (e.g., new-post badge) and fetch full content on demand, reserving heavy real-time for genuinely live features.

Senior Engineer Tips

- Default feeds to cursor pagination + cache-first reads + a lightweight real-time signal (fetch-on-open); it's the resilient, battery-friendly pattern for the most common prompt.
- Design writes optimistic + queued + idempotent from the start; naive "POST and wait" breaks on flaky networks and duplicates on retry.
- Shape payloads to the screen (list vs detail), minimize round-trips, and treat the API as a versioned published contract — shipped clients live for months.

Architect Perspective

The client-server contract is the pivotal design surface in mobile system design: it dictates perceived performance, battery/data cost, resilience, and evolvability. Designing it deliberately — right API style, cursor pagination, push/WebSocket for real-time, screen-shaped payloads, and robust read/write (cache-first + optimistic/queued/idempotent) flows over flaky networks — is what makes an app feel fast and reliable. It feeds directly into the caching/offline/sync strategy and maps onto the client's Clean/repository architecture (03_caching_offline_and_scale.md, Module 16, Module 40).

Summary

- Design the client-server contract: API style (REST/GraphQL/gRPC + WebSocket/push), cursor pagination for feeds, real-time via push/WebSocket (not polling), screen-shaped payloads.
- Read flow = cache-first + background refresh; write flow = optimistic + queued + synced + reconciled (idempotent).
- Make it efficient over flaky networks (few round-trips, right-sized, compressed, conditional GET, retry/backoff), evolvable (versioned/additive), and secure.

Revision Notes

- API style: REST (simple/cacheable/chatty), GraphQL (field-precise, fewer round-trips), gRPC (typed/streaming), WebSocket/SSE (in-app real-time), push (out-of-app) — often mixed.
- Pagination: cursor/keyset for feeds (stable/efficient) > offset (duplicates/skips). Real-time: push/WS > polling; design real-time-signal vs pull-on-demand.
- Read: cache-first + refresh (SWR). Write: optimistic + outbox queue + sync + reconcile + idempotency key. Efficiency: minimize round-trips, right-size payloads, gzip, ETag/conditional GET, retry/backoff, timeouts. Evolvable (versioned/additive), secure (TLS/pinning/auth, server-authoritative).

Practice Questions

1. Why cursor over offset pagination for a live feed?
2. When do you use WebSocket vs push vs polling?
3. How do you design a resilient, idempotent write flow?

Coding Questions

1. Specify a feed API contract (cursor pagination + real-time signal + write).
2. Choose an API style for a given app and justify it.
3. Design the read + write data flows (cache-first read; optimistic/queued write).

Mini Project

Contract & data flow design (Flutter/design): For a feed app, design the client-server contract — API style choice (justified), cursor-paginated feed endpoint (`nextCursor` / `hasMore`), a real-time new-post signal (push/WebSocket), screen-shaped payloads (list vs detail) + CDN images, and the read (cache-first + refresh) and write (optimistic + queued + idempotent + reconcile) flows — plus efficiency (conditional GET/compression/backoff) and evolvability (versioning). Acceptance:

API style justified; cursor pagination; real-time via push/WS (signal + fetch-on-open); read/write flows designed (cache-first; optimistic/queued/idempotent); flaky-network efficiency + versioning + security addressed.

Caching, Offline & Scale Trade-offs

> The mobile-defining part of the design: choose **where and how to cache** (memory/disk/DB, per data type), a **freshness strategy** (cache-first / network-first / **stale-while-revalidate**, with TTL + validators), an **offline model** (read-only cached snapshot vs full offline-first with an outbox + sync + **conflict resolution**), and reason about **scale + trade-offs** on the client (large lists, image memory, storage growth, battery). The unifying tension is **consistency vs availability vs cost**: fresher data means more network/battery; offline means sync/conflict complexity. Senior design is **picking and justifying** the right point on these spectra for the requirements.

Introduction

This file covers the caching/offline/scale layer of the design — the part most distinctive to mobile. It ties together caching strategy, offline models, sync/conflict resolution, and client-side scale trade-offs, drawing on offline-first (Module 19), storage (Module 20/Module 34), and performance (Module 21).

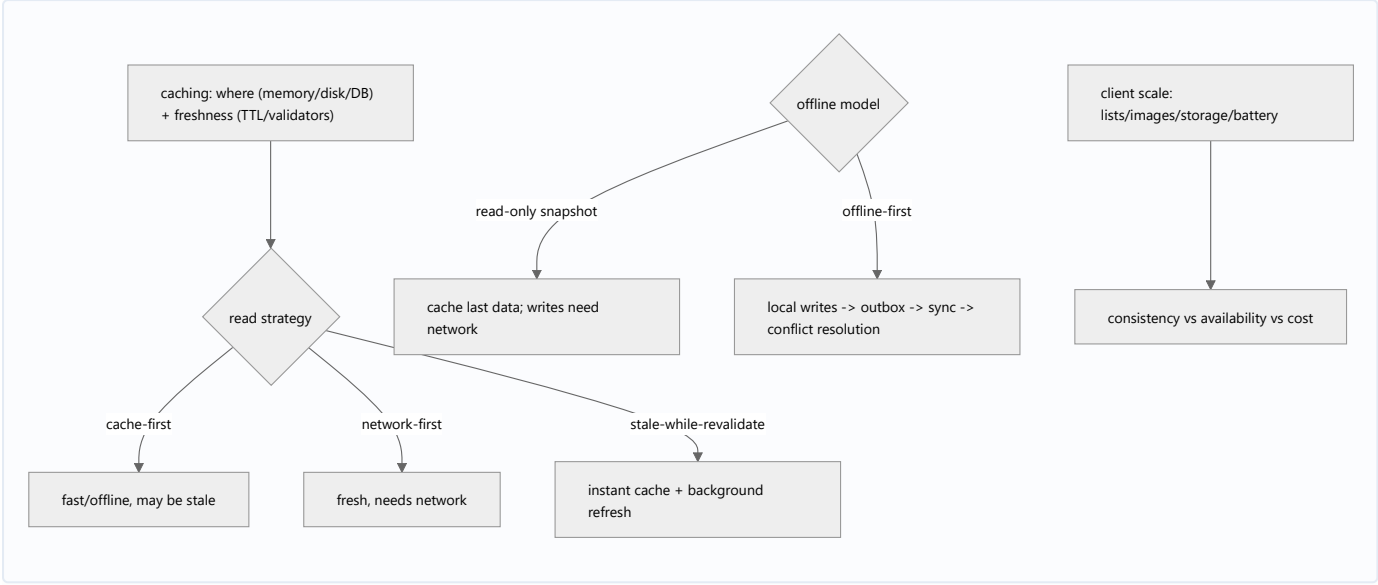
Why this concept exists

On mobile, the network is the bottleneck and offline is the norm at times, so **caching and offline strategy dominate perceived performance and reliability** — and they're where the hard trade-offs live (freshness vs battery, availability vs consistency, storage vs re-fetch). Designing them deliberately (not "just cache everything") is the crux of a good mobile design.

Real-world analogy

It's **stocking the expedition vehicle**: you decide **what to carry** (cache) vs **fetch from base** (network), **how fresh supplies must be** (TTL/freshness), and whether you can **operate fully off-grid** (offline-first with your own log to reconcile later) or only **read your last resupply** (read-only snapshot). Carrying more means heavier load (storage/memory); insisting on fresh means frequent trips (battery/data). You **stock for the mission's actual needs** — over-stocking and under-stocking both cost you.

Internal Working



- **Where to cache** (by data type):
  - **In-memory** (fast, volatile) for hot/session data; **disk/files** for blobs/images (bounded — Module 34); **local DB** (sqflite/Drift/Isar/Hive) for structured, queryable, persistent data (Module 20). **secure storage** for tokens (Module 37).
  - Match store to access pattern (query? blob? ephemeral?).
- **Freshness strategy** (per data type):
  - **Cache-first**: serve cache, maybe refresh in background — **fast + offline-capable**, may be **stale**. Great for feeds/read-heavy.
  - **Network-first**: fetch, fall back to cache — **fresh**, needs network. For must-be-current data (balances).
  - **Stale-while-revalidate (SWR)**: show cache **instantly** + refresh in background + update UI — the **best-of-both** default for most reads.
  - **TTL + validators**: age-based freshness (TTL) + **ETag/Last-Modified** conditional GET to skip unchanged (Module 34). Cache is **re-fetchable** (OS may clear).
- **Offline model** (choose per requirements):
  - **Read-only offline snapshot**: cache last-fetched data for offline **reading**; **writes require network**. Simple; enough for many apps.
  - **Full offline-first**: local writes succeed offline → **outbox queue** → **sync** on reconnect → **conflict resolution** (last-write-wins / server-wins / merge / CRDTs) → optimistic UI + reconcile. Powerful but **complex** (Module 19). Justify the complexity against the requirement.
  - **Conflict resolution** is the hard part: decide the policy (LWW is simple but loses data; merge/CRDT is correct but complex); design idempotent sync (Module 31).
- **Client-side scale trade-offs** (the "scale" that matters on-device):

    - **Large lists** → cursor pagination + virtualization + paged cache (don't hold everything).
    - **Images/blobs** → bounded disk cache + resized variants + memory limits (top memory offender).
    - **Storage growth** → eviction (TTL/LRU/size budget), don't cache unbounded; cache is re-fetchable.
    - **Battery/data** → SWR/push over aggressive polling; batch sync; sync on wifi/charging where possible.
- **The core tensions (articulate them)**:
    - **Consistency vs availability**: offline/cache-first = available but possibly stale; network-first = fresh but unavailable offline.
    - **Freshness vs cost**: fresher = more network/battery.
    - **Simplicity vs capability**: read-only snapshot (simple) vs full offline-first (capable, complex).
    - Senior design = **choosing a justified point** per data type, not maximal everything.
- **Consistency model**: mobile is usually **eventually consistent** (cache + eventual sync); state this explicitly and design reconciliation for it.

Memory Representation

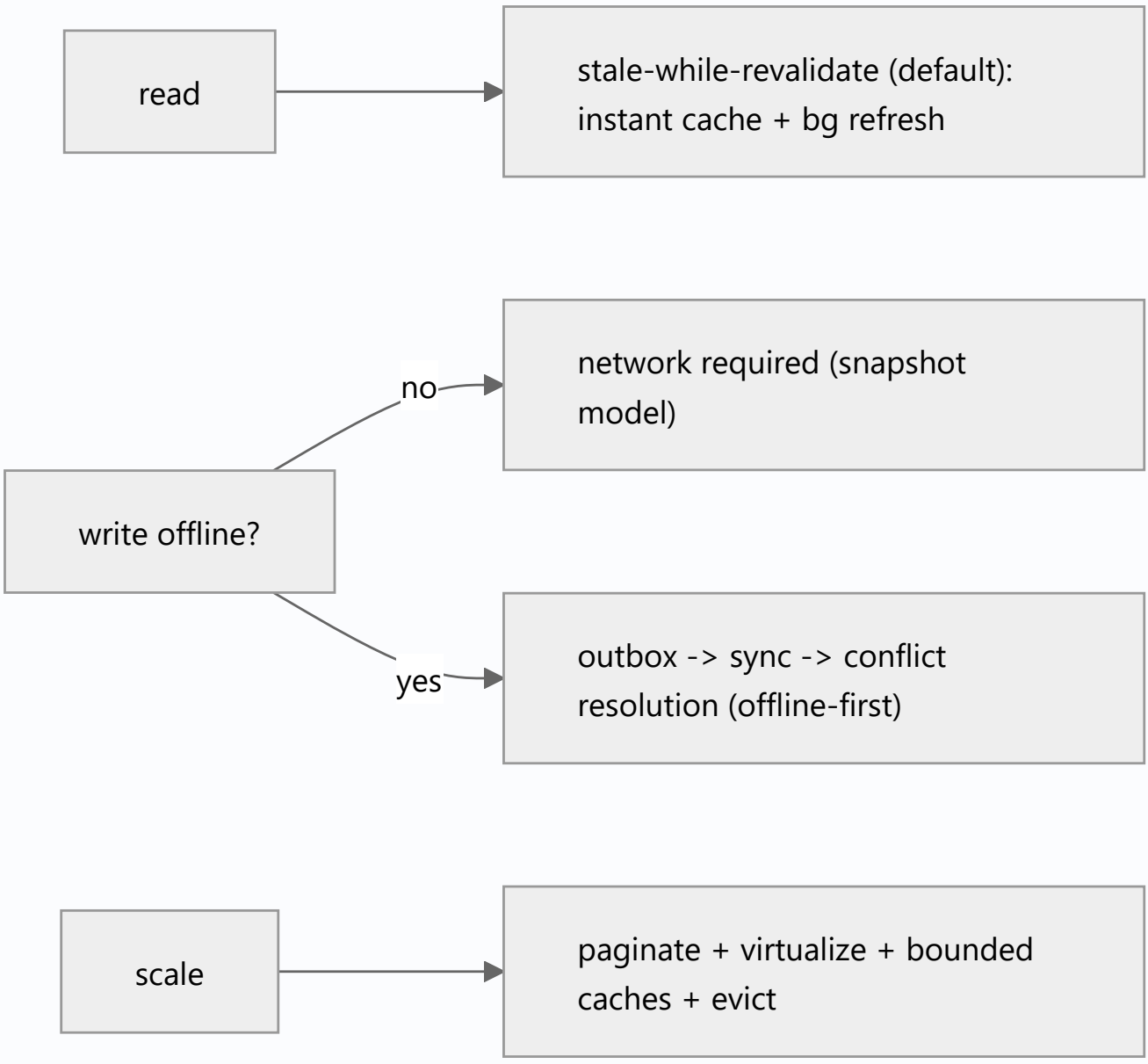
Design artifact: a **per-data-type table** (where cached, freshness strategy, TTL/validators, offline model). Runtime: memory/disk/DB caches + an outbox (for offline-first) + eviction policies. The invariant: bounded, re-fetchable caches; eventual consistency via sync.

Compiler / Runtime / Engine / VM Behavior

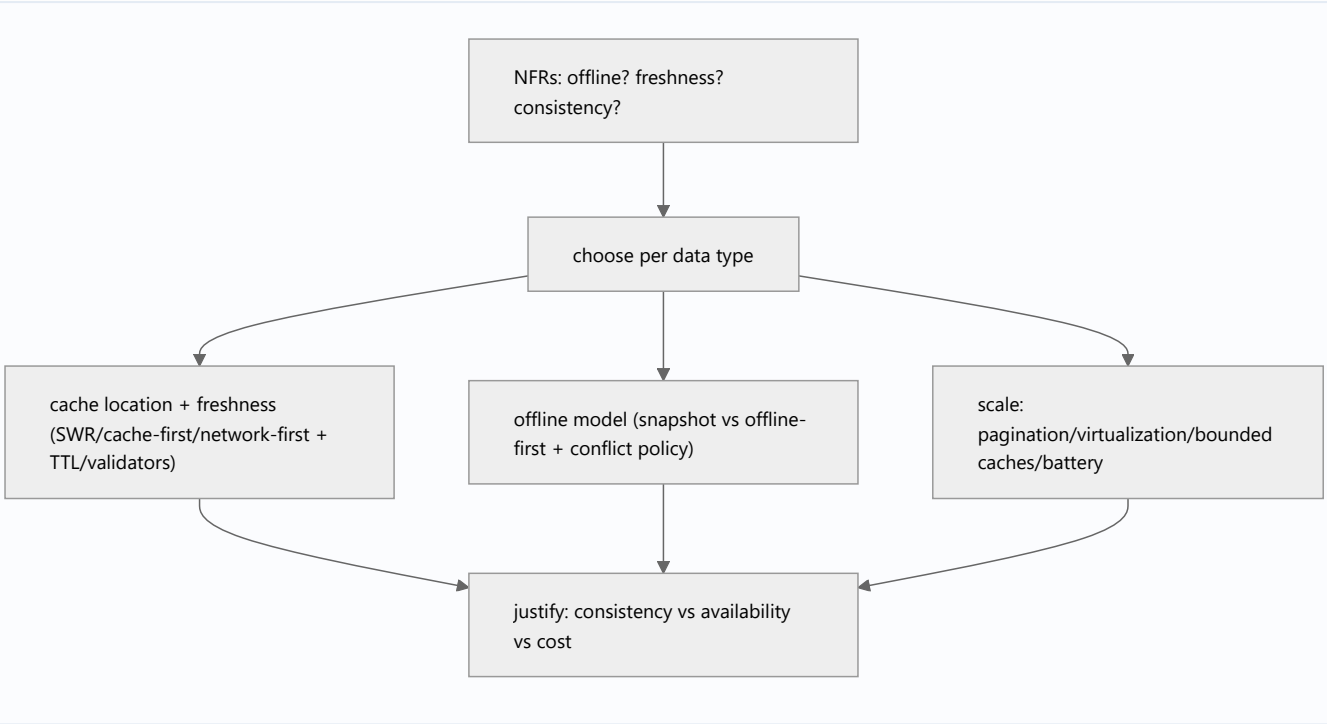
Design-level (implemented via storage/networking/offline modules). Runtime characteristics you design for: cache-hit latency (instant/offline), sync timing (eventual), memory/storage bounds, and battery cost of freshness/sync.

Examples

```
Per-data-type caching/offline design (feed app):

  Data           Where          Freshness            Offline model

  ----           -----          ---------            -------------

  Feed posts     local DB       stale-while-revalid. read-only snapshot (offline read)

  Post images    disk cache     TTL + LRU + size cap re-fetchable (OS may clear)

  Account balance (not cached)  network-first        no offline (must be fresh)

  Draft/compose  local DB       n/a                  FULL offline-first (outbox + sync + LWW)

  Auth token     secure storage n/a                  persisted (short-lived + refresh)


Trade-off statements to say out loud:

  "Feed = cache-first/SWR -> available offline, eventually consistent (acceptable for a feed)."

  "Balance = network-first -> must be fresh; show 'unavailable offline' rather than stale money."

  "Drafts = offline-first with LWW -> simple conflict policy; acceptable since single-user edits."
```


read
stale-while-revalidate (default): instant cache + bg refresh
write offline?
no
network required (snapshot model)
yes
outbox -> sync -> conflict resolution (offline-first)
scale
paginate + virtualize + bounded caches + evict

Diagrams



Common Mistakes

Mistake	Why it's wrong	Fix
"Cache everything" (one strategy)	Different data needs different freshness	Choose strategy per data type
Full offline-first when not needed	Huge complexity (sync/conflicts)	Read-only snapshot if writes-offline aren't required
Ignoring conflict resolution	Data loss/corruption on sync	Explicit policy (LWW/server-wins/merge/CRDT)
Unbounded caches	Storage/memory bloat	TTL/LRU/size eviction; re-fetchable
Showing stale money/critical data	Wrong/unsafe	Network-first for must-be-fresh data
Aggressive polling for freshness	Battery/data drain	SWR + push; batch/conditional sync
Not stating the consistency model	Ambiguous design	Declare eventual consistency + reconciliation
Holding entire list/images in memory	OOM/jank	Paginate + virtualize + bounded image cache

Best Practices

- Choose **caching per data type** (where + freshness): **SWR** as the default read strategy, **network-first** for must-be-fresh data, **cache-first** for read-heavy; use **TTL + validators**; keep caches **bounded + re-fetchable**.
- Pick the **offline model per requirement**: **read-only snapshot** (simple) unless offline **writes** are required, then **full offline-first** (outbox + sync + explicit **conflict resolution**) — justify the

complexity.

- Design **client-side scale**: cursor pagination + virtualization + paged/bounded caches + image memory limits + battery-aware sync (SWR/push, batch, wifi/charging).
- **State the consistency model** (usually eventual) and **articulate the trade-offs** (consistency vs availability vs cost) for each choice.

Performance

Caching/offline strategy *is* mobile performance: SWR gives instant reads + freshness; bounded caches keep memory/storage sane; pagination/virtualization keep lists smooth; battery-aware sync avoids drain. The trade-offs (fresh vs battery, available vs consistent) are performance/UX decisions made explicit (Module 21).

Advantages / Disadvantages

- **+** (Deliberate per-data strategy) fast + resilient + battery-friendly UX, correct freshness where it matters, bounded resource use, justified trade-offs.
- **−** More design effort (per-data decisions), offline-first complexity (sync/conflicts), cache-invalidation subtlety, eventual-consistency UX handling.

Interview Questions

1. 🟢 **What caching read strategies are there, and a good default?** — Cache-first, network-first, and stale-while-revalidate; SWR (instant cache + background refresh) is a strong default for most reads.
2. 🟢 **Read-only snapshot vs full offline-first — how to choose?** — Snapshot (cache for offline reads; writes need network) unless the requirement demands offline *writes*, which needs full offline-first (outbox + sync + conflict resolution).
3. 🟡 **What's the hard part of offline-first, and the options?** — Conflict resolution: last-write-wins (simple, lossy), server-wins, merge, or CRDTs (correct, complex) — plus idempotent sync.
4. 🟡 **Which data should NOT be cached/served stale?** — Must-be-fresh/critical data (balances, live prices) — use network-first and show "unavailable offline" rather than stale.
5. 🟡 **How do you handle client-side scale (big lists/images)?** — Cursor pagination + list virtualization + paged/bounded caches + resized images + memory limits; battery-aware sync.
6. 🔴 **What are the core trade-offs to articulate?** — Consistency vs availability (offline/cache = available but stale; network-first = fresh but offline-unavailable) and freshness vs cost (fresher = more network/battery).
7. 🔴 **What consistency model do mobile apps usually have, and why state it?** — Eventual consistency (cache + eventual sync); stating it clarifies reconciliation design and sets UX

expectations.

Senior Engineer Tips

- Make caching/offline decisions **per data type** in a small table (where/freshness/offline model); "one strategy for everything" is the tell of a shallow design.
- Default reads to SWR and reserve network-first for genuinely must-be-fresh data; never show stale money/critical values to save a request.
- Only reach for full offline-first when offline *writes* are a real requirement — its sync/conflict complexity is large, and a read-only snapshot satisfies most apps.

Architect Perspective

Caching, offline, and client-side scale are where mobile system design is won or lost, because the network is unreliable and the device is constrained. The senior move is a **per-data-type strategy** (location + freshness + offline model) that lands each datum at a justified point on the consistency/availability/cost spectra, with bounded caches, pagination, and battery-aware sync — and an explicit consistency model. This synthesizes offline-first, storage, and performance into the resilience layer of the design, feeding directly into the interview approach and the end-to-end capstone (Module 19, Module 20, Module 21, 04_mobile_system_design_interview.md).

Summary

- Cache per data type (memory/disk/DB) with a freshness strategy (SWR default; network-first for fresh-critical; cache-first for read-heavy) + TTL/validators; keep caches bounded + re-fetchable.
- Choose the offline model per requirement (read-only snapshot vs full offline-first with outbox + sync + conflict resolution); justify complexity.
- Design client scale (pagination/virtualization/bounded caches/battery-aware sync); state the (usually eventual) consistency model and articulate consistency-vs-availability-vs-cost trade-offs.

Revision Notes

- Cache location by type: memory (hot), disk (blobs, bounded), local DB (structured/queryable), secure storage (tokens). Freshness: cache-first / network-first / **SWR** (default) + TTL + ETag/Last-Modified; bounded + re-fetchable.
- Offline model: read-only snapshot (simple; writes need net) vs full offline-first (outbox + sync + conflict resolution: LWW/server-wins/merge/CRDT + idempotent); justify.

- Scale: cursor pagination + virtualization + paged/bounded caches + image memory limits + battery-aware sync (SWR/push/batch/wifi-charging). Consistency usually eventual — state it; articulate consistency vs availability vs cost.

Practice Questions

1. When do you choose cache-first vs network-first vs SWR?
2. Read-only snapshot vs full offline-first — how do you decide, and what's the hard part?
3. What client-side scale issues arise for big lists/images, and their fixes?

Coding Questions

1. Build a per-data-type caching/offline table for a given app (where/freshness/offline model).
2. Design an SWR read flow + a bounded image cache.
3. Design an offline-first write path (outbox + sync + a chosen conflict policy).

Mini Project

Caching/offline/scale design (Flutter/design): For a feed + compose app, produce a per-data-type caching/offline table (feed=SWR/snapshot, images=disk TTL/LRU, balance=network-first/no-offline, drafts=offline-first LWW, token=secure), design the SWR read + offline-first write (outbox/sync/conflict) flows, address client-side scale (pagination/virtualization/bounded caches/battery-aware sync), and articulate the consistency-vs-availability-vs-cost trade-offs + the (eventual) consistency model. Acceptance: per-data-type strategy (not one-size); SWR default + network-first for fresh-critical; justified offline model + conflict policy; bounded caches + pagination/virtualization + battery-aware sync; consistency model + trade-offs stated.

The Mobile System-Design Interview

> Mobile system-design interviews score **structured thinking, communication, and trade-off reasoning** — not a memorized "correct" architecture. Run a **repeatable framework**: (1) **clarify requirements** (functional + non-functional + scope) by asking questions, (2) sketch **high-level architecture + data flow** (client-server contract), (3) **deep-dive** the interesting parts (pagination, real-time, caching, offline/sync, the client architecture), (4) **address mobile constraints** (network/battery/memory/offline), (5) **discuss trade-offs + bottlenecks**. Common prompts ("design the Instagram feed / a chat app / a file-sync app") all reduce to **contract + caching + offline + real-time + client architecture** — communicated out loud, driven by the requirements you clarified.

Introduction

This file gives a concrete framework + communication approach for mobile system-design interviews, maps common prompts to the design levers, and lists the signals interviewers look for. It operationalizes the fundamentals (01_system_design_fundamentals.md) for the interview setting.

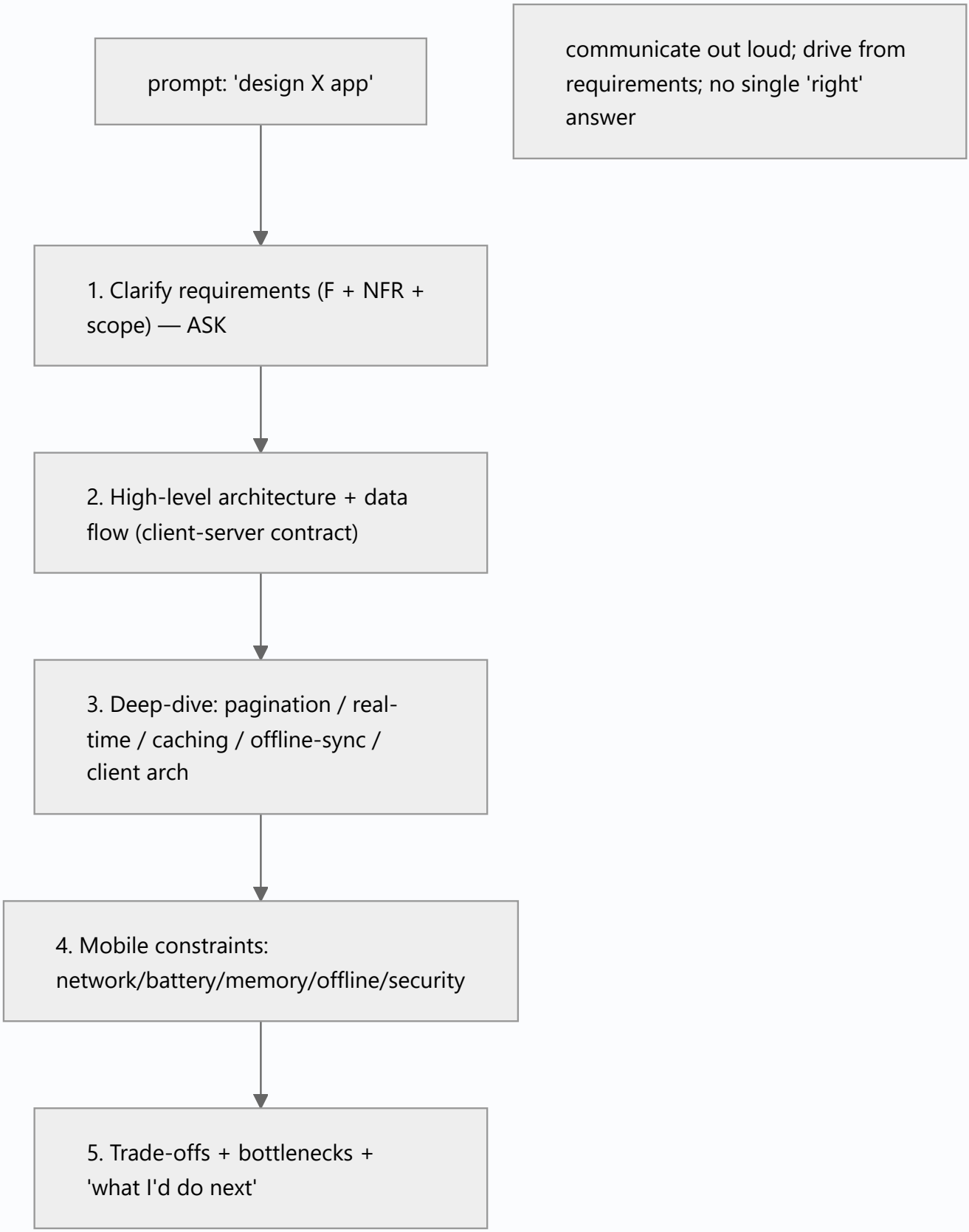
Why this concept exists

Candidates often freeze or ramble in system-design rounds, jumping to a solution or reciting a diagram. A **framework + communication discipline** turns an open-ended prompt into a structured, senior-sounding conversation — and mobile prompts have a **recognizable shape** (contract/caching/offline/real-time/architecture) you can navigate confidently once you know it.

Real-world analogy

It's like a **structured consultation**: a good consultant doesn't blurt a solution — they **ask about your situation and constraints** (requirements), **sketch an approach** (high-level design), **dive into the tricky parts** (deep-dives), **flag risks/limits** (constraints/bottlenecks), and **explain the trade-offs** so *you* can decide. The interviewer is evaluating your consulting process, not a single "right" prescription.

Internal Working



- **The framework (drive it explicitly, out loud)**:

1. **Clarify requirements** (spend real time here): functional (features to support) + **non-functional** (offline? real-time? scale/users? latency? consistency? platforms? security?) + **scope** (what's in/out). **Ask questions** — interviewers reward it and it prevents designing the wrong thing.
2. **High-level design + data flow**: sketch client ↔ server (the **contract**), major components (API, cache/DB, real-time channel), and the **read/write data flow**. Keep it high-level first, then zoom in.
3. **Deep-dives** (pick the interesting ones for the prompt): **pagination** (cursor), **real-time** (push/WebSocket), **caching + freshness** (SWR/etc.), **offline + sync + conflicts**, **client architecture** (Clean/feature-first/MVVM + repository/cache), image/media handling. Go deep where the prompt is hard.
4. **Mobile constraints**: explicitly address **flaky/offline network, battery, memory, small screen, untrusted client, app-store/background limits** — this is the mobile signal.
5. **Trade-offs + bottlenecks**: state the tensions (consistency vs availability, freshness vs battery, real-time vs cost, optimistic vs pessimistic), identify bottlenecks (network, image memory), and say **what you'd measure/do next**.

- **Common prompts → the same levers**:
  - **Instagram/Twitter feed**: cursor pagination, cache-first/SWR feed, offline snapshot, real-time new-post *signal* (push), image loading/memory, optimistic likes, repository/cache client arch.
  - **Chat/messaging (WhatsApp)**: **WebSocket** real-time, **offline-first** message queue (outbox) + sync + ordering, delivery/read receipts, local DB of messages, pagination of history, conflict/ordering handling.
  - **File sync (Dropbox/Drive)**: **offline-first** with sync engine + **conflict resolution** (versioning), chunked upload/download, background sync, bounded local cache.
  - **Maps/ride-hailing**: real-time location (stream), map tiles/markers + clustering, battery-aware location, offline caching of tiles.
  - All reduce to **contract + caching + offline/sync + real-time + client architecture** under **constraints**.
- **Communication (heavily scored)**: think out loud, **state assumptions**, structure the conversation (announce which step you're in), **draw diagrams** (boxes + arrows for data flow), invite the interviewer to steer, and **manage time** (breadth first, then depth on the key parts).
- **Signals interviewers look for**: requirements-first discipline, mobile-specific reasoning (offline/caching/battery — not backend sharding), clear data-flow diagrams, justified trade-offs, and structured communication. **Red flags**: jumping to a solution, ignoring NFRs/constraints, backend-only focus, one "answer" with no trade-offs, disorganized rambling.

- **No single right answer**: the goal is a **coherent, justified** design for the clarified requirements — defend choices, acknowledge alternatives.

Memory Representation

Not runtime — an **interview playbook**: the 5-step framework, a mapping of prompts→levers, a diagram habit, and a communication checklist. Internalize it so the open-ended prompt becomes a structured walk.

Compiler / Runtime / Engine / VM Behavior

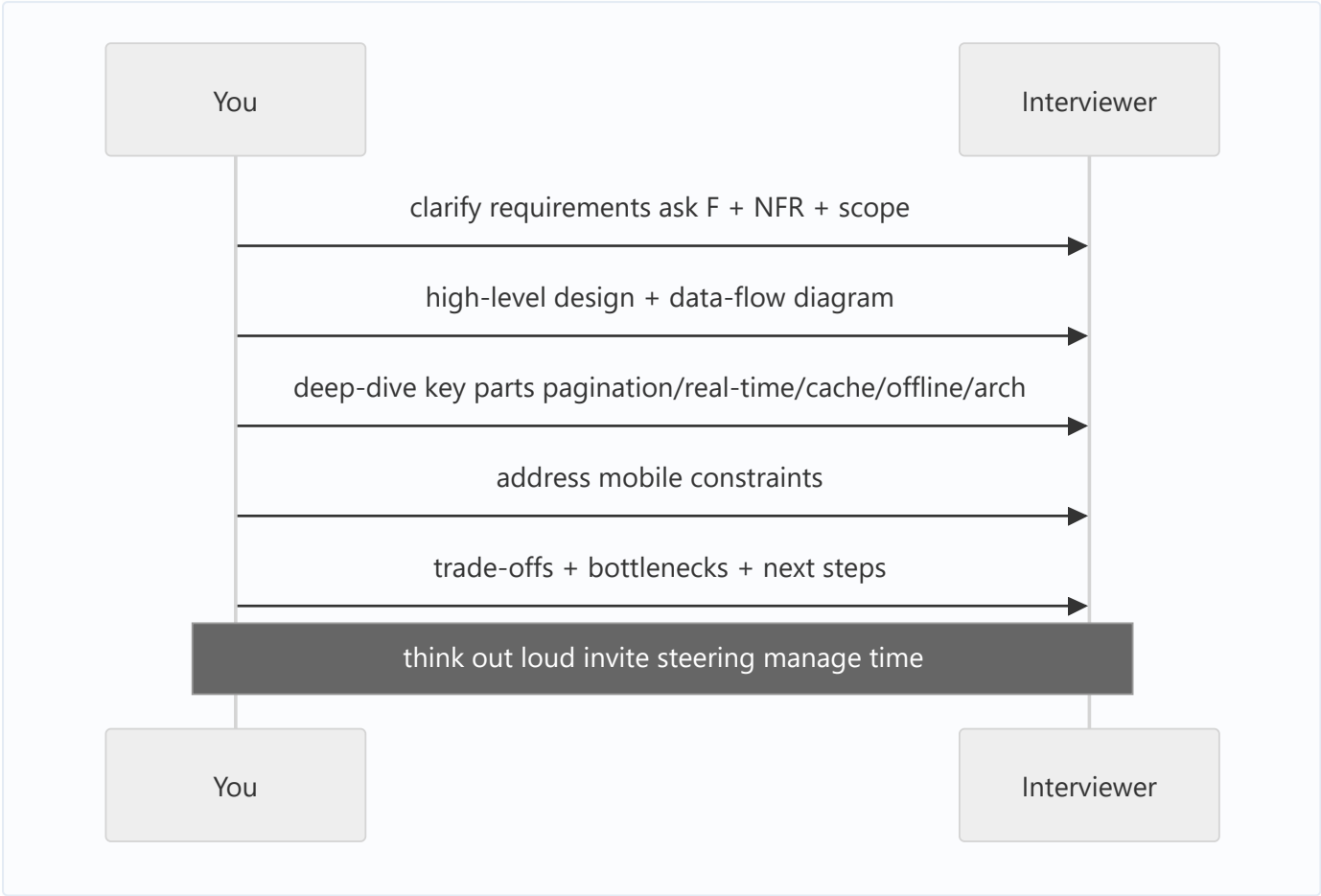
Not applicable — this is a communication/reasoning skill; the underlying tech is the rest of this module + handbook.

Examples

```
5-step walkthrough (prompt: "design the Instagram feed"):
  1. Clarify: offline read? (yes, last feed) real-time? (new-post badge) scale? (100M) media?
(images+video)
              latency? (feed<1s) -> assumptions stated.
  2. High-level: App <-REST/GraphQL-> API; local DB cache; CDN for media; WebSocket/push for new-post
signal.
                  Read: cache-first + refresh. Write (like): optimistic + queued.
  3. Deep-dives: cursor pagination; SWR feed cache; offline snapshot; image memory (resized + bounded
cache);
                  optimistic likes + reconcile; client arch = MVVM + repository + cache.
  4. Constraints: flaky net -> cache-first + retry; battery -> push not poll; memory -> virtualized list +
image cache.
  5. Trade-offs: eventual consistency for feed (acceptable); freshness vs battery; what I'd measure
(frame/startup, cache hit rate).
```

```
Prompt -> dominant levers cheat-sheet:
  feed         -> cursor pagination + SWR cache + offline snapshot + real-time signal + image memory
  chat         -> WebSocket + offline-first outbox + ordering + local DB + receipts + history pagination
  file sync    -> offline-first sync engine + conflict/versioning + chunked transfer + background sync
  maps/ride    -> location stream + tiles/markers/clustering + battery-aware + offline tiles
```

Diagrams



Common Mistakes

Mistake	Why it hurts	Fix
Jumping to a solution/diagram	Designs the wrong thing; poor signal	Clarify requirements first (ask)
Ignoring non-functional requirements	Misses design drivers	Ask offline/real-time/scale/latency/security
Backend-only focus (sharding/LB)	Wrong for a mobile round	Client + contract + caching/offline/battery
One "answer," no trade-offs	Misses the point	State tensions + alternatives + justify
Silent thinking	Interviewer can't follow	Think out loud; announce steps
No diagrams	Hard to communicate data flow	Draw boxes + arrows
Poor time management (deep too early)	Runs out of time	Breadth first, then depth on key parts
Ignoring constraints	Misses mobile signal	Explicitly address network/battery/memory

Best Practices

- Run the **5-step framework out loud**: clarify requirements (ask, F + NFR + scope) → high-level design + **data-flow diagram** → **deep-dive** the hard parts → **mobile constraints** → **trade-offs/bottlenecks/next steps**.

- Recognize that common prompts reduce to **contract + caching + offline/sync + real-time + client architecture**; pick the **dominant levers** for the prompt.
- **Communicate** deliberately: state assumptions, structure the conversation, draw diagrams, invite steering, **manage time** (breadth → depth).
- Emphasize **mobile-specific reasoning** (offline/caching/battery/memory/untrusted client) and **justify trade-offs** — there's **no single right answer**.

Performance

The interview tests design *thinking*, but performance reasoning is a scored sub-signal: mention latency/battery/memory budgets, cache-hit rates, and what you'd measure — showing you design with performance in mind (Module 21/Module 47).

Advantages / Disadvantages

- **+** (Framework) confident, structured, senior-sounding designs; adaptable to any prompt; strong communication/trade-off signals.
- **−** Requires practice + breadth (whole handbook); still needs genuine depth on deep-dives; over-scripting can feel rote if not adapted to the prompt.

Interview Questions

1. 🟢 **What's the first thing you do in a system-design interview?** — Clarify requirements (functional + non-functional + scope) by asking questions — never jump to a solution.
2. 🟢 **What's the 5-step framework?** — Clarify requirements → high-level design/data flow → deep-dive key parts → mobile constraints → trade-offs/bottlenecks/next steps.
3. 🟡 **What do common mobile prompts reduce to?** — Client-server contract + caching + offline/sync + real-time + client architecture, under device constraints.
4. 🟡 **How does a mobile round differ from a backend round?** — Focus on the client + contract + offline/caching/battery/memory, not server sharding/LB — treat the backend as a given API unless asked.
5. 🟡 **What are interviewers actually scoring?** — Structured thinking, requirements-first discipline, mobile-specific reasoning, clear data-flow diagrams, justified trade-offs, and communication.
6. 🔴 **Design a chat app's offline behavior briefly.** — Local DB of messages, an outbox for sends, WebSocket for real-time, sync + ordering on reconnect, delivery/read receipts, history pagination — offline-first with idempotent sync.
7. 🔴 **Why is "no single right answer" important to convey?** — Because senior design is about justified trade-offs for the clarified requirements; defending choices + acknowledging alternatives is the signal, not a canonical diagram.

Senior Engineer Tips

- Spend the first few minutes clarifying requirements and stating assumptions; it's the highest-ROI thing you can do and the most common thing candidates skip.
- Lead with a high-level data-flow diagram, then deep-dive the two or three genuinely hard parts (pagination/real-time/offline) — breadth first, depth where it matters, watch the clock.
- Keep the conversation mobile: reason about offline/caching/battery/memory and justify trade-offs out loud; drifting into backend sharding is the classic mobile-round miss.

Architect Perspective

The mobile system-design interview is a proxy for real architectural work: taking an ambiguous prompt, clarifying requirements, designing the client + contract + caching/offline/real-time under constraints, and justifying trade-offs — communicated clearly. The framework isn't a script to recite but a **thinking structure** that mirrors how a senior actually designs. Mastering it (over the whole handbook's substance) is what lets you turn any "design app X" into a coherent, defensible, mobile-centric design — the culmination this module builds toward (01_system_design_fundamentals.md, 02_client_server_and_data_flow.md, 03_caching_offline_and_scale.md).

Summary

- Run the 5-step framework out loud: clarify requirements → high-level design/data flow → deep-dive → mobile constraints → trade-offs/bottlenecks.
- Common prompts reduce to contract + caching + offline/sync + real-time + client architecture under device constraints; pick the dominant levers.
- Interviewers score structured thinking, mobile-specific reasoning, diagrams, justified trade-offs, and communication — no single right answer.

Revision Notes

- Framework (out loud): 1) clarify requirements (F+NFR+scope, ASK) 2) high-level design + data-flow diagram 3) deep-dive (pagination/real-time/caching/offline-sync/client arch) 4) mobile constraints (network/battery/memory/offline/untrusted) 5) trade-offs/bottlenecks/next-steps.
- Prompts → contract + caching + offline/sync + real-time + client arch (feed/chat/file-sync/maps). Backend = given API unless asked.
- Scored: requirements-first, mobile reasoning, diagrams, justified trade-offs, communication. Red flags: jump-to-solution, ignore NFRs/constraints, backend-only, one-answer-no-tradeoffs, rambling. No single right answer.

Practice Questions

1. Walk the 5-step framework for "design the Instagram feed."
2. What dominant levers does a chat-app prompt require?
3. What signals (and red flags) do interviewers watch for?

Coding Questions

1. Produce a clarifying-questions list for a given prompt (F + NFR + scope).
2. Draw a high-level data-flow diagram for a feed app.
3. Deep-dive one hard part (real-time, offline-sync, or pagination) with trade-offs.

Mini Project

Interview walkthrough (Flutter/design): Pick a prompt (feed, chat, or file-sync) and produce a full 5-step written walkthrough: clarifying questions + assumptions (F + NFR + scope), a high-level data-flow diagram (client-server contract), two deep-dives on the hard parts (e.g., real-time + offline-sync, or pagination + caching), explicit mobile constraints, and a trade-offs/bottlenecks/next-steps section. Acceptance: requirements clarified first (questions + assumptions); high-level diagram; ≥2 substantive deep-dives with trade-offs; mobile constraints addressed; trade-offs/bottlenecks stated; mobile-centric (not backend-sharding); communicates structure clearly.

System Design Integration (Capstone: Design an App End-to-End)

> Put it all together by designing one app **end-to-end**: **clarify requirements** (functional + non-functional), define the **client-server contract + data flow** (API style, cursor pagination, real-time), choose a **per-data-type caching/offline/sync strategy** with **conflict resolution**, address **mobile constraints** (network/battery/memory/untrusted client), **map it onto the client architecture** (Clean/feature-first/MVVM + repositories/cache), and **present the trade-offs + bottlenecks** — delivered as both a **design doc** and a **whiteboard-style walkthrough**. This is the synthesis of the module (and much of the handbook) into one coherent, defensible mobile system design.

Introduction

This module capstone integrates the fundamentals, contract/data-flow, caching/offline/scale, and interview framework into a complete end-to-end design for a realistic app (a feed or chat). It demonstrates the full process producing a coherent artifact — the deliverable a senior mobile engineer/architect is expected to produce.

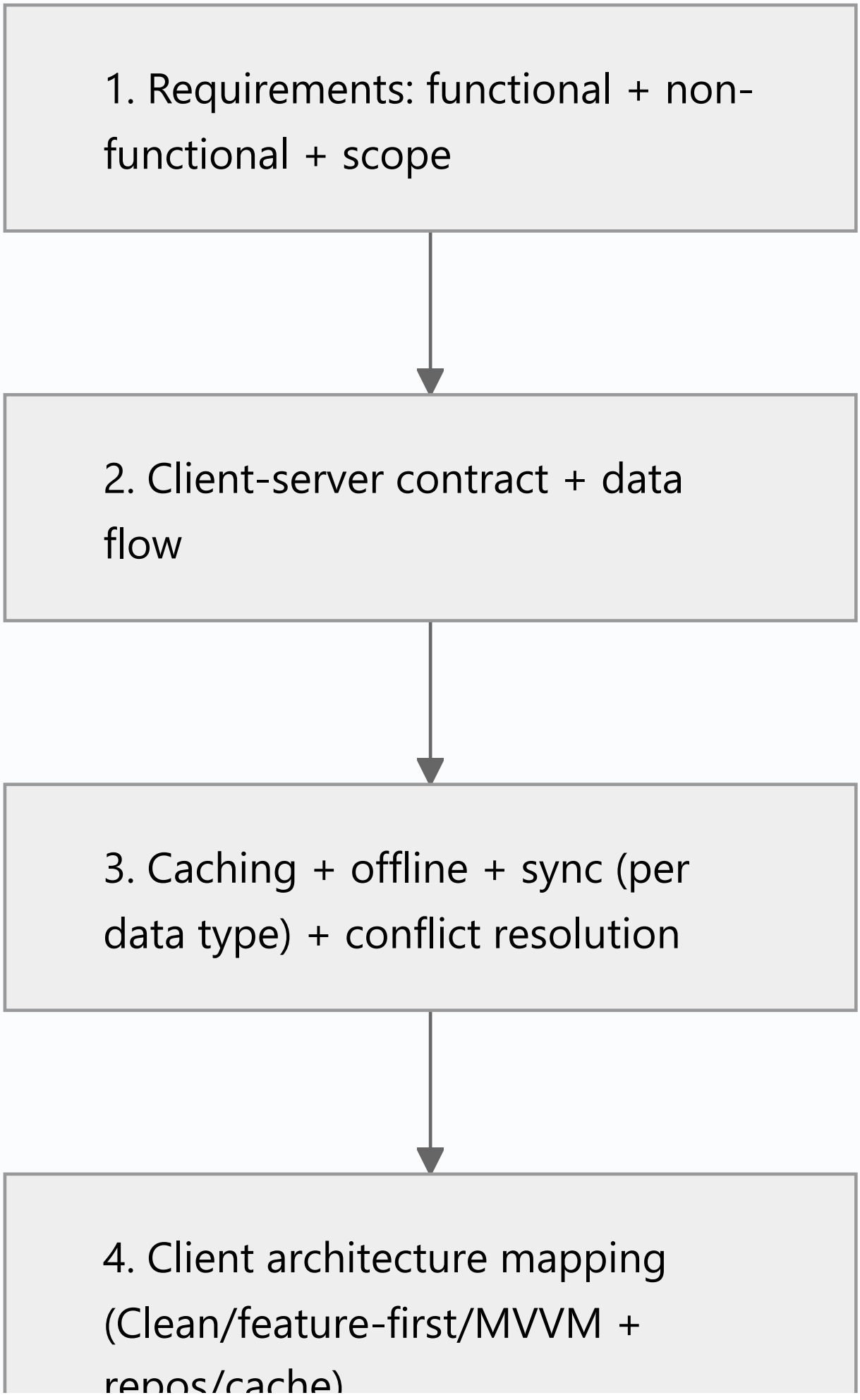
Why this concept exists

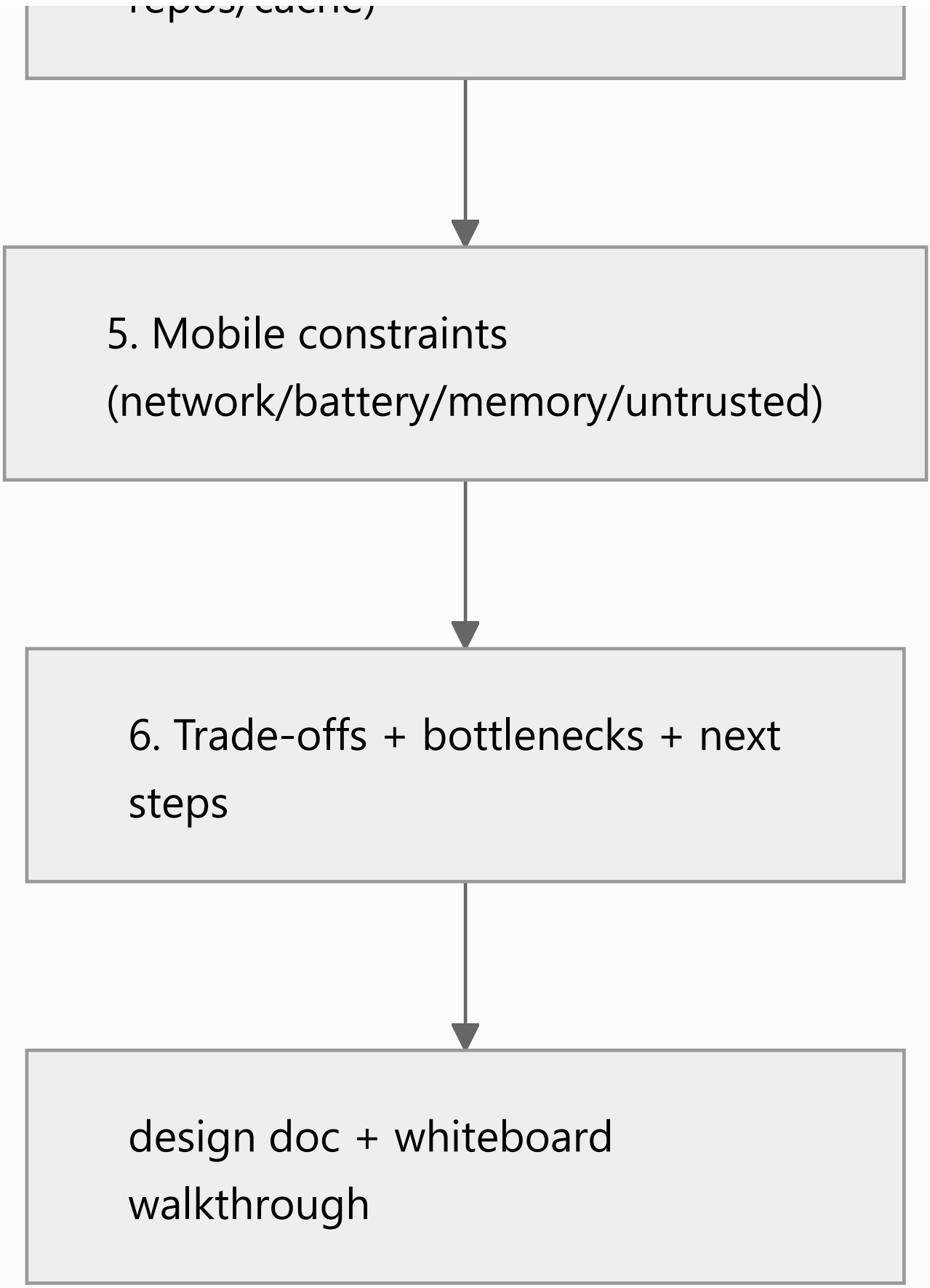
Knowing the pieces isn't enough; the value is composing them into **one coherent design** for a real app, with the contract, caching/offline, architecture, and trade-offs all consistent and justified. This capstone provides that integrated exemplar and cements the repeatable process.

Real-world analogy

It's delivering a **complete expedition plan**: the mission goals + conditions (requirements), the radio protocol with base (contract), the supply/off-grid strategy (caching/offline/sync), the vehicle's internal organization (client architecture), and a briefing that **explains the trade-offs and risks** (bottlenecks) — coherent enough that the team can execute and stakeholders can evaluate.

Internal Working


1. Requirements: functional + non-functional + scope
2. Client-server contract + data flow
3. Caching + offline + sync (per data type) + conflict resolution
4. Client architecture mapping (Clean/feature-first/MVVM + repos/cache)



- **Step 1 — Requirements** (01_system_design_fundamentals.md): functional (feed/chat features) + **non-functional** (offline? real-time? scale? latency? consistency? security?) +

scope/assumptions.

- **Step 2 — Contract + data flow** (02_client_server_and_data_flow.md): API style (REST/GraphQL + WebSocket/push), **cursor pagination**, real-time signal vs pull-on-demand, screen-shaped payloads + CDN media, and **read (cache-first/SWR)** + **write (optimistic/queued/idempotent)** flows.
- **Step 3 — Caching/offline/sync** (03_caching_offline_and_scale.md): a **per-data-type table** (where cached, freshness strategy, offline model), **offline model** (snapshot vs offline-first + **conflict resolution**), bounded caches + eviction.
- **Step 4 — Client architecture**: map onto **feature-first** slices (Module 44) with **Clean layers** (Module 40) + **MVVM** presentation (Module 43); **repositories** own caching/offline/sync (Module 40/Module 19); DI wires it. Show where the contract, cache, DB, real-time channel, and sync live in the architecture.
- **Step 5 — Mobile constraints**: flaky/offline network (cache-first + retry), battery (push not poll, batch sync), memory (pagination + virtualization + bounded image cache), untrusted client (server-authoritative, TLS/pinning, no secrets — Module 37), background/store limits.
- **Step 6 — Trade-offs + bottlenecks**: consistency vs availability (eventual for feed), freshness vs battery, optimistic vs pessimistic, real-time vs cost; bottlenecks (network latency, image memory); **what you'd measure/do next** (frame/startup/memory, cache-hit rate, sync-failure rate — Module 52).
- **Deliverable**: a concise **design doc** (the six sections + diagrams) and a **whiteboard walkthrough** (the interview framework — 04_mobile_system_design_interview.md), communicating structure + trade-offs.
- **Coherence + justification**: every choice traces to a requirement/constraint; alternatives acknowledged; **no single right answer** — the design is defensible for the stated requirements.

Memory Representation

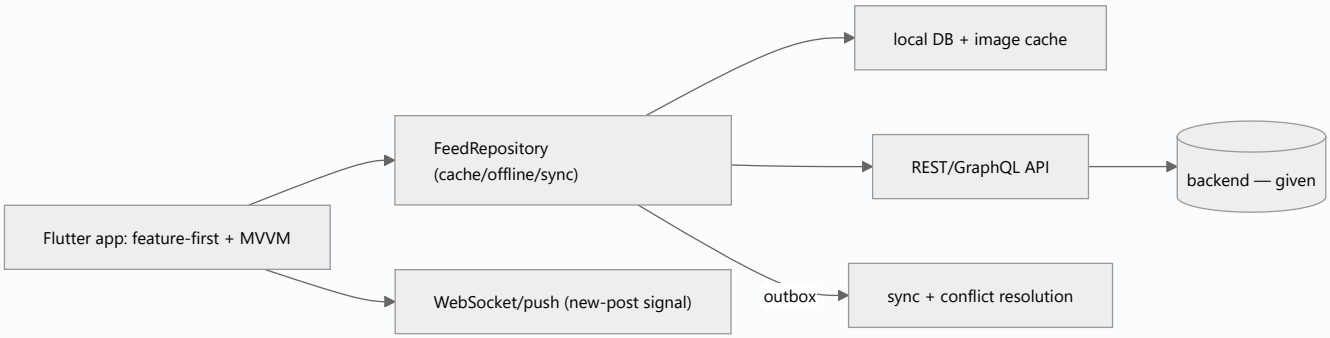
The artifact: a six-section design doc (requirements → contract → caching/offline → architecture → constraints → trade-offs) + diagrams (data flow, architecture, per-data caching table). It's a static, justified plan.

Compiler / Runtime / Engine / VM Behavior

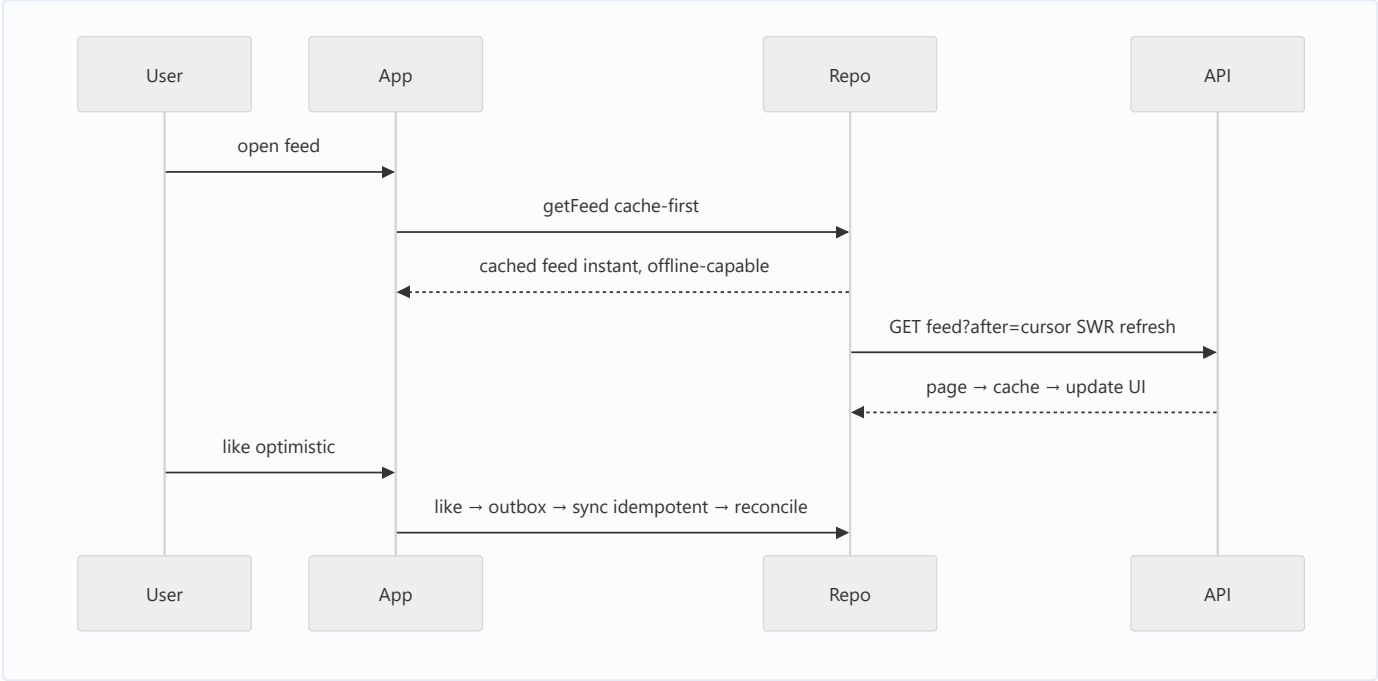
Design-level; the implementation spans the handbook's modules. The design *specifies* runtime characteristics (latency, offline behavior, memory/battery budgets, consistency) the implementation must meet.

Examples

End-to-end design (feed app) — the six sections, condensed:
  1. Requirements: infinite feed, likes, detail; offline read=yes, real-time=new-post badge,
                   scale=10M, feed<1s, eventual consistency, auth + no PII logs. Backend=given REST/GraphQL API.
  2. Contract: GraphQL feed(after,limit)->{items,nextCursor,hasMore}; WS new_post signal; POST like (idempotent);
               CDN sized images; read=cache-first/SWR; write=optimistic+queued.
  3. Caching/offline: feed=DB+SWR+offline snapshot; images=disk TTL/LRU/size cap; likes=offline-first outbox+LWW;
               token=secure storage. Bounded + evict; eventual consistency.
  4. Architecture: feature-first (features/feed) + Clean (domain/data/presentation) + MVVM;
               FeedRepository owns cache/network/sync; DI-wired; WS+push in a service.
  5. Constraints: cache-first+retry (flaky net); push not poll (battery); virtualized list + bounded image cache (memory);
               server-authoritative + TLS/pinning (untrusted client).
  6. Trade-offs: eventual consistency for feed (ok); freshness vs battery; optimistic likes reconcile;
               bottlenecks: image memory, network latency; measure: frame/startup/memory, cache-hit, sync-failure.



Diagrams



Common Mistakes

Mistake	Why	Fix
Skipping requirements/assumptions	Designs the wrong system	Clarify F + NFR + scope first
No data-flow/architecture diagrams	Hard to communicate	Draw contract + architecture + caching table
One caching strategy for all data	Wrong freshness per datum	Per-data-type strategy
Over-engineering (full offline-first everywhere)	Needless complexity	Right-size (snapshot unless writes-offline needed)
Ignoring mobile constraints	Misses the essence	Address network/battery/memory/untrusted
Not mapping to client architecture	Design floats abstractly	Map onto feature-first/Clean/MVVM + repos
Presenting without trade-offs/bottlenecks	Incomplete design	State trade-offs + bottlenecks + next steps

Best Practices

- Deliver the **six-section design** (requirements → contract/data flow → caching/offline/sync → client architecture → constraints → trade-offs) as a **doc + whiteboard walkthrough**, with **diagrams**.
- **Clarify requirements first**; choose a **per-data-type caching/offline strategy**; **right-size** the offline model + conflict resolution; **map onto feature-first/Clean/MVVM + repositories**.
- **Address mobile constraints explicitly** (network/battery/memory/untrusted client) and **justify every trade-off** against requirements; note **bottlenecks + what you'd measure**.

- Ensure **coherence** (every choice traces to a requirement/constraint) and acknowledge **alternatives** — defensible, not canonical.

Performance

The design bakes in performance: cache-first/SWR + cursor pagination + virtualization + bounded image cache + push-over-poll = fast, resilient, battery-friendly UX, with explicit budgets + metrics to verify (Module 21/Module 52). Performance is a design output, traced to requirements.

Advantages / Disadvantages

- **+** Coherent, justified, mobile-centric end-to-end design; implementable; communicates well (doc + whiteboard); synthesizes the handbook.
- **−** Requires broad synthesis + judgment; time-consuming to do fully; no single correct answer (must defend choices).

Interview Questions

1. 🟢 **What are the sections of an end-to-end mobile design?** — Requirements, client-server contract/data flow, caching/offline/sync, client architecture, mobile constraints, trade-offs/bottlenecks.
2. 🟢 **How do you present the design?** — As a concise doc + a whiteboard walkthrough with diagrams (data flow, architecture, per-data caching), communicating structure + trade-offs.
3. 🟡 **How does the design map onto the client architecture?** — Feature-first slices + Clean layers + MVVM; repositories own caching/offline/sync; DI wires it; real-time via a service.
4. 🟡 **How do you decide caching/offline per data type?** — By freshness need + offline requirement: SWR/snapshot for feed, network-first for must-be-fresh, offline-first (outbox + conflict policy) only where offline writes are required.
5. 🟡 **Which mobile constraints must the design address, and how?** — Flaky net (cache-first + retry), battery (push/batch), memory (pagination/virtualization/bounded caches), untrusted client (server-authoritative + TLS/pinning).
6. 🔴 **How do you keep the design coherent and defensible?** — Trace every choice to a requirement/constraint, articulate trade-offs, acknowledge alternatives — coherent for the clarified requirements, not a canonical answer.
7. 🔴 **What bottlenecks/metrics would you call out?** — Network latency + image memory as bottlenecks; measure frame/startup/memory, cache-hit rate, and sync-failure rate to verify/iterate.

Senior Engineer Tips

- Structure the deliverable as the six sections with diagrams; a coherent doc + whiteboard that traces every decision to a requirement is what reads as senior.
- Right-size the caching/offline design per data type and reserve full offline-first for genuine offline-write needs; over-engineering the sync engine is as much a failure as ignoring offline.
- Always close with trade-offs, bottlenecks, and what you'd measure next; a design without acknowledged tensions and metrics looks naive.

Architect Perspective

This capstone is the synthesis the whole handbook builds toward: turning requirements + mobile constraints into a coherent, justified end-to-end system — contract, caching/offline/sync, client architecture, and trade-offs — communicated clearly. It composes networking, offline-first, storage, performance, security, and the architecture band into a single design discipline. The mark of a senior mobile architect is exactly this: given an ambiguous prompt, produce a defensible, mobile-centric design that a team can build and stakeholders can evaluate (01_system_design_fundamentals.md, Module 40, Module 19, Module 47).

Summary

- Design end-to-end in six sections: requirements → contract/data flow → caching/offline/sync → client architecture → constraints → trade-offs; deliver as a doc + whiteboard with diagrams.
- Clarify requirements first; per-data-type caching/offline (right-sized); map onto feature-first/Clean/MVVM + repositories; address mobile constraints; justify trade-offs + bottlenecks + metrics.
- Coherent + defensible for the stated requirements — the synthesis of the module and much of the handbook.

Revision Notes

- Six sections: requirements (F+NFR+scope) → contract/data flow (API style, cursor pagination, real-time, read/write flows) → caching/offline/sync (per-data-type + conflict resolution, bounded) → client architecture (feature-first + Clean + MVVM + repositories/DI) → mobile constraints (network/battery/memory/untrusted) → trade-offs/bottlenecks/metrics.
- Deliver doc + whiteboard with diagrams (data flow, architecture, caching table); every choice traces to a requirement/constraint; right-size; acknowledge alternatives; no single right answer.
- Synthesizes networking/offline/storage/performance/security/architecture band.

Practice Questions

1. What are the six sections, and what goes in each?
2. How do you map the design onto the client architecture?
3. How do you keep the design coherent and defensible?

Coding Questions

1. Produce a six-section end-to-end design doc for a given app.
2. Draw the data-flow + architecture + per-data caching diagrams.
3. Write the trade-offs/bottlenecks/metrics section.

Mini Project

End-to-end design (capstone — Flutter/design): Design a feed or chat app end-to-end as a six-section doc + whiteboard walkthrough: (1) requirements (F + NFR + scope), (2) client-server contract + data flow (API style, cursor pagination, real-time, read/write flows), (3) per-data-type caching/offline/sync + conflict resolution, (4) client architecture mapping (feature-first + Clean + MVVM + repositories/DI), (5) mobile constraints, (6) trade-offs + bottlenecks + metrics — with diagrams. Acceptance: all six sections with diagrams; requirements-first + assumptions; deliberate contract (cursor pagination, real-time); per-data-type caching/offline (right-sized + conflict policy); mapped onto client architecture; mobile constraints addressed; trade-offs/bottlenecks/metrics stated; coherent + defensible; presentable as a whiteboard walkthrough.