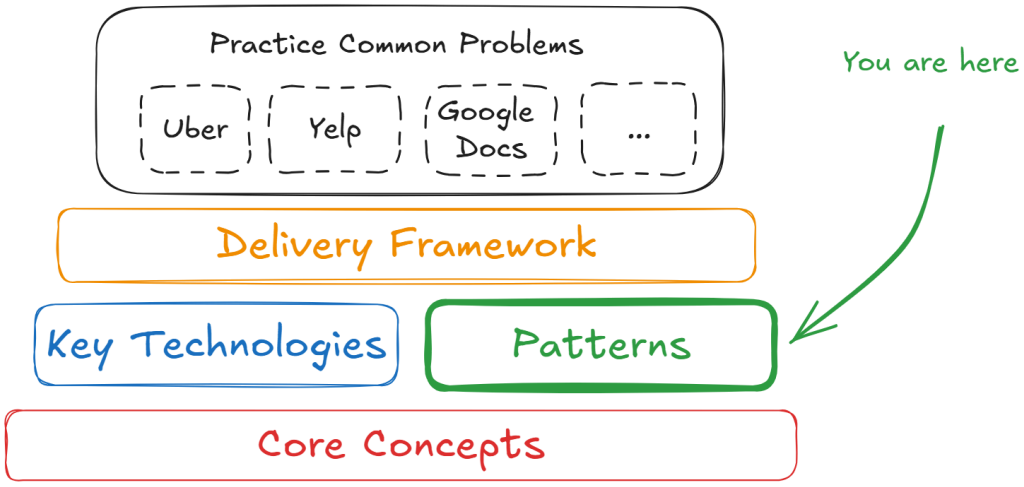
Common Patterns



Overall Structure

⭐ 1. “The most common system design interview patterns, built by FAANG managers and staff engineers”

✔ What it really means

These are **battle-tested architectural templates** that top companies repeatedly use while designing scalable systems.
Think of them as **reusable blueprints**—like MVC in software engineering, but for system design.

✔ Real-life analogy

Just like architects reuse familiar house plans (2BHK layouts, duplex, villas) instead of designing from scratch every time, system designers reuse **patterns** like:

- Load Balancing
- Caching
- Sharding
- Queue-based decoupling

- Event-driven architecture
- Replication

These patterns solve problems that appear **again and again** in scalable systems.

✔ Why it matters in interviews

If you recognize the right pattern early, you can:

- Answer faster
- Answer correctly
- Impress interviewers because you show real-world intuition

⭐ 2. “By taking the key technologies and core concepts we've discussed and combining them, you can build a wide variety of systems.”

✔ What it means

System design isn't about memorizing 200 technologies.
It's about understanding a few fundamental building blocks like:

- Cache (Redis / CDN)
- Message Queue (Kafka / SQS)
- Load Balancer
- Database models
- Replication & Sharding
- Object storage (S3, GCS)

By **mixing and matching** these blocks, you can design systems like:

- Instagram → CDN + object storage + cache + pub/sub + feed ranking
- Uber → real-time messaging + geo-indexed DB + Kafka + caching
- WhatsApp → message queues + database replication + load balancers

✔ Example

If someone asks you to **design YouTube**:

- Videos go to object storage → pattern
- Global delivery via CDN → pattern
- Metadata stored in SQL → pattern
- Comments/likes handled via microservices → pattern
- Queue for processing thumbnails → pattern

Different system → same set of patterns.

⭐ 3. "Success in the time-constrained environment of interviewing is all about patterns."

✔ What it means

Most system design interviews give you **35–45 minutes**.
That is NOT enough time to:

- invent new architecture
- deeply research every component
- explore all failure scenarios

So the interviewer expects you to quickly identify:

👉 “Ah, this is a **read-heavy system** → I should use cache + replication”
👉 “This needs **asynchronous processing** → I should use queues”
👉 “This needs **global users** → use CDN + geo-replicated DB”

✔ Real-life scenario

If asked to build an **analytics system**, and you immediately say:

> “This is a write-heavy system → we need Kafka + batch layer + pre-aggregation.”

That shows **pattern recognition**, which senior engineers have.

⭐ 4. “Patterns allow you to know what's interesting and what's not.”

✔ What this means

During interviews, some details matter; others don’t.

If you understand patterns, you instantly know:

Important points:

- how to scale reads
- how to scale writes
- how to handle failures
- how to guarantee consistency or choose availability

Unimportant points (at interview level):

- exact schema of every table
- library-level API details
- how servers are physically wired
- low-level implementation of TCP

✔ Example

If the problem is **"Design Instagram Feed"**,
then interesting parts are:

- feed fanout pattern (push vs pull)
- caching strategies
- ranking calculation
- handling massive write bursts from celebrities

Not interesting:

- how HTTP POST is parsed
- exact JSON structure

Patterns help you avoid wasting time on the irrelevant parts.

⭐ 5. “They also save you time by helping you to see common failure modes rather than reverse engineering them on the fly.”

✔ What this means

Good engineers know **how systems break** because every pattern has typical failure modes.

Examples of common failures by pattern:

🟦 Caching pattern → common failure

Cache stampede (when cache expires, millions of requests hit the DB).
If you know the pattern, you immediately propose:

- cache lock
- staggered expiration
- background refresh

🟦 **Queue pattern → common failure**

Consumers lag behind producers, causing delays.
Solution patterns:

- autoscaling consumers
- DLQs (Dead Letter Queues)

🟦 **Sharding pattern → common failure**

Hot shard when one key gets too much traffic.
Solutions:

- consistent hashing
- resharding

✔ Interview advantage

You don't need to think from scratch.
Just mention common pitfalls:

> "If the feed cache is missed, we may overload the DB — so we will use write-through caching."

That shows senior-level depth.

⭐ 6. "These patterns are not mutually exclusive; you'll often find yourself using several of them."

✔ What this means

Real systems combine many patterns at once.

✔ Example: Designing Twitter

Twitter internally uses at least **7+ patterns simultaneously**:

- Load balancing → for distributing traffic
- Caching → for timelines
- Queue-based fanout → distributing tweets
- Sharding → storing billions of tweets
- Pub/sub → notifications
- CDN → delivering media
- Replication → availability

No single pattern solves the entire problem.
You assemble multiple patterns to form the final architecture.

⭐ 7. “In our problem breakdowns, we'll call out the patterns so you can spot these commonalities.”

✔ What it means

Whenever they explain a system, they highlight which pattern is being used so that:

- You recognize repeating structures
- You learn to identify patterns quickly
- You build architecture intuition

✔ Example

If you see **“We use Redis to reduce read load”**, that's the **Caching Pattern**.
If you see **“We use Kafka for decoupling write workloads”**, that's the **Queue Pattern**.

With time, you start thinking:

> “Oh, this problem is similar to the 'feed fanout' pattern from Twitter.”

> “This architecture needs consistent hashing; I’ve seen this in 'sharded DB' patterns.”

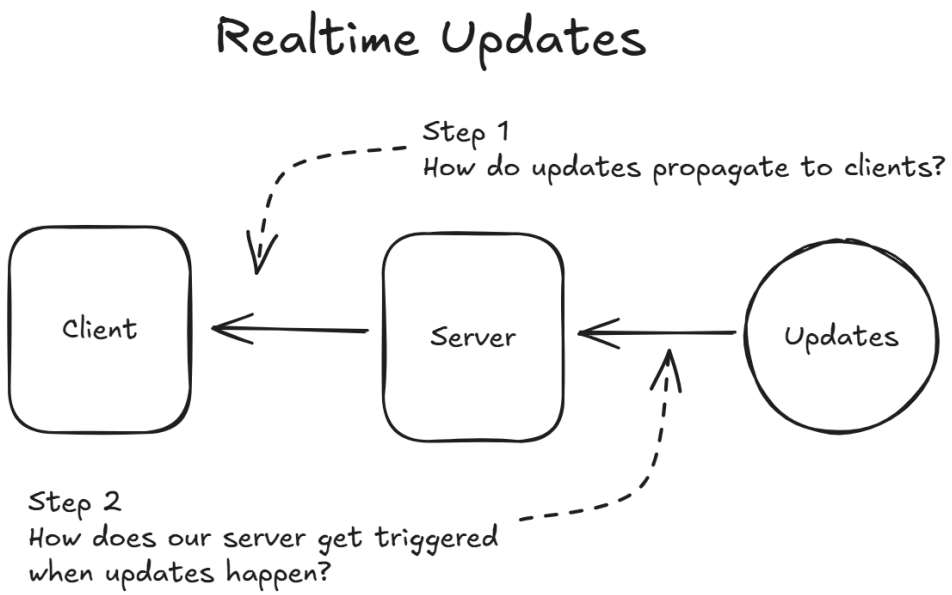
Pattern recognition becomes automatic.

✅ Summary in Simple Words

Here’s the essence of what all these statements mean:

> **System design interviews become much easier once you learn a small set of reusable architectural patterns.**
> **These patterns save time, reduce mistakes, show senior-level thinking, and apply to almost every real-world system.**

1.) Pushing Realtime Updates



Realtime Updates Challenges

Below is a **clear, expanded, step-by-step explanation** of the “Realtime Updates” pattern from HelloInterview — **with real-world examples (WhatsApp, Stock Market Apps, Google Docs, Uber location updates)** and a breakdown of the diagram you shared.

⭐ What Problem is “Pushing Realtime Updates” Trying to Solve?

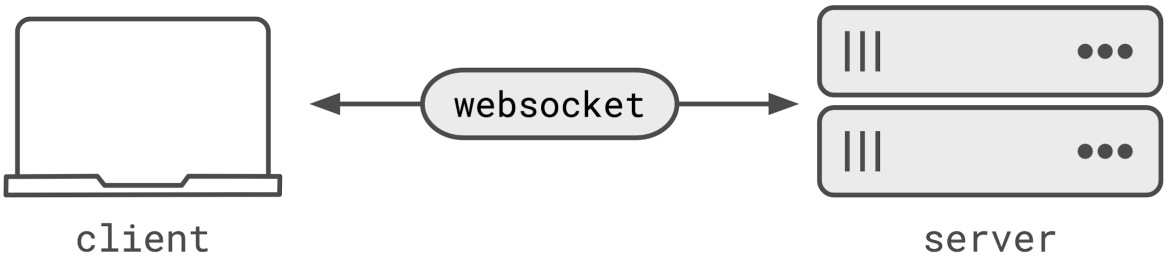
Some applications need to **instantly** show updates to users **without the user refreshing the screen**, such as:

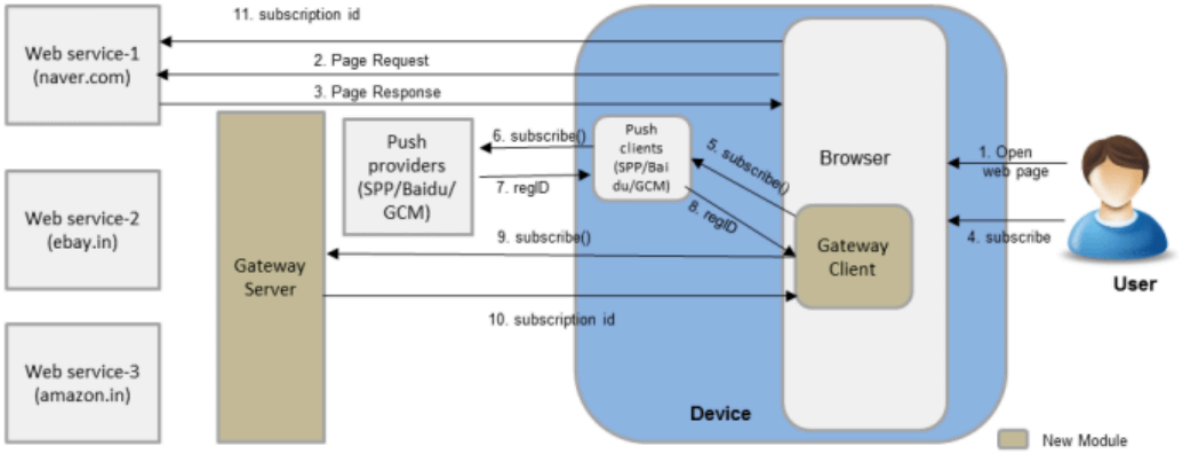
- WhatsApp → new message appears instantly
- Uber → driver’s location moves in real time
- Stock market app → price changes every second
- Google Docs → cursor moves and text updates as others type

This requires a way for **the server to push updates to the client** *as soon as* something changes.

That's what this pattern is all about.

⭐ Understanding the Diagram





The diagram shows **2 challenges**:

⭐ Challenge 1 — How do updates propagate to clients?

This is the **client → server → client** communication channel.

When something changes on the server, how does the server notify the user?

There are 3 common protocols:

1️⃣ HTTP Polling (simple but inefficient)

Client: “Any new updates?”
Server: “No.”
A few seconds later…
Client: “Any new updates now?”

This is like **refreshing Instagram repeatedly to see new posts**.

✔ Real-life example

Small chat apps or dashboards where latency isn’t critical.

2️⃣ Server-Sent Events (SSE)

A **one-way stream** from server → client.
Great for updates that flow mostly from server to UI:

- notifications
- stock price updates
- live scores

✔ Real example

Cricbuzz / stock tickers use SSE because the server is continuously pushing data.

3️⃣ WebSockets (two-way communication)

A **persistent bi-directional connection** between client ↔ server.

Fastest, lowest-latency protocol.

✔ Real-life uses

- **WhatsApp Web**
- **Google Docs collaborative editing**
- **Uber live location updates**
- **Gaming servers**

⭐ Challenge 2 — How does the server get triggered when updates happen?

This is the **internal architecture** challenge.

When something changes (new message, new comment, new location update), how does the server know to push it?

There are 2 main approaches:

⭐ Approach A — Using Pub/Sub (MOST COMMON)

Used in systems like WhatsApp, Slack, Twitter notifications

A **publisher** sends a message →
A **pub/sub system** distributes it →
All connected servers **receive and push** to clients.

✔ Real-life analogy

A WhatsApp group:

- Someone sends a message → the message is published
- All group members → subscribed → instantly get it

✔ Why Pub/Sub works great

- Servers don't need to know who is online
- Very scalable
- Great for fanout (one message → many receivers)

Pub/Sub examples

- Kafka
- Redis Pub/Sub

- Google Pub/Sub
- AWS SNS/SQS

⭐ Approach B — Stateful Servers (Consistent Hash Ring)

Used when each user needs to stick to the same server
(Ex: Google Docs collaborative editing)

Why?

- The server stores temporary state like cursor positions, partial changes, document session info.
- Switching servers would lose in-memory state.

✔ How it works

A consistent hash ring assigns each client to a specific server →
That server handles all their updates and pushes changes to others.

✔ Real-world example

Google Docs:

- User A edits an online document
- Server 3 handles this document's session
- It maintains conflict resolution (Operational Transform / CRDT)
- It pushes updates to other editors in real time

⭐ Putting It All Together (End-to-End Flow)

Let’s walk through a real-world scenario: **WhatsApp Web receiving a new message**

📱 Example: WhatsApp Web

Step 1: The update happens

Your friend sends you a message → this is the “Updates” circle in the diagram.

Step 2: Server gets triggered

WhatsApp backend publishes the new message to a **Kafka topic** (Pub/Sub system).

Step 3: Server receives the pub/sub event

The correct WebSocket server (the one you're connected to) receives the event.

Step 4: Server pushes update to client

The server instantly sends the message over your WebSocket connection →
Message appears on WhatsApp Web without refresh.

🚕 Example: Uber Driver Location Update

Driver app sends GPS location updates

→ publishes updates every few seconds

Backend pushes to pub/sub

→ Kafka or Pub/Sub

Connected clients (rider app) subscribe

→ server streams location via WebSocket/SSE

Rider instantly sees driver moving on the map

→ no refresh required

📊 Example: Real-time Stock Price Dashboard

Stock market feed publishes new price

→ to a streaming system (Kafka, Pub/Sub)

Backend processes updates

→ sends them via SSE/WebSocket

Browser dashboard instantly updates the graph

→ without polling or refreshing

📄 Example: Google Docs

User types a character

→ stateful server running a document session processes the change

Conflict resolution logic applied

→ CRDT / OT merges changes

Server pushes update to all collaborators

→ via WebSockets

Users see text appear in real time.

⭐ Why Start with Polling?

The article says:

> "We generally recommend starting with HTTP polling until it no longer serves your needs."

Because:

- Polling is simple
- No fancy infrastructure needed
- Works for low-frequency updates
- Easy to scale with a load balancer

Only when:

- updates must be **instant**
- user count grows
- polling causes wasted traffic

… then you switch to **SSE or WebSockets**.

This mirrors real industry practice.

⭐ Final Summary (Easy to Remember)

Realtime updates require solving 2 things:

1️⃣ How do updates reach the client?

Options:

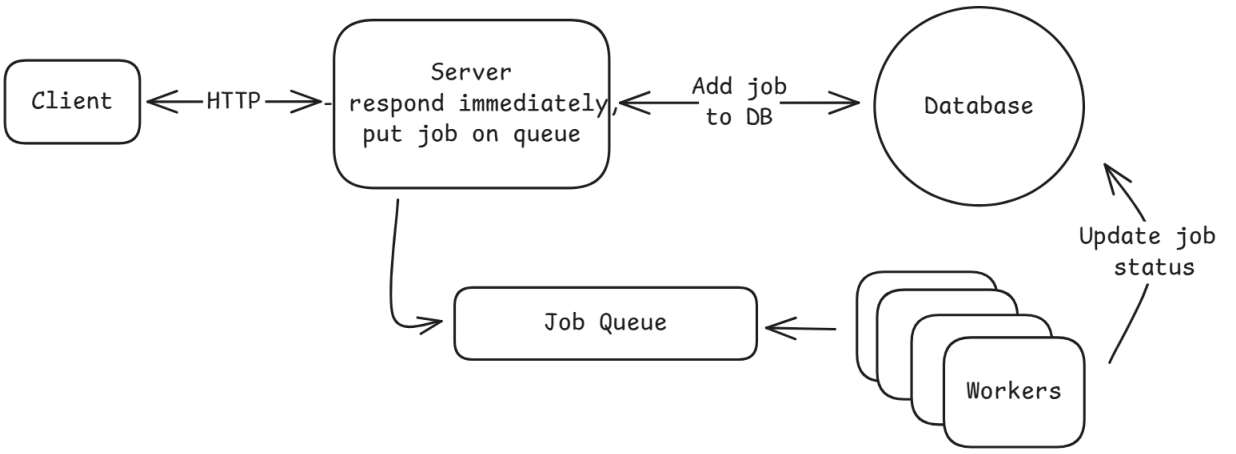
- Polling → simple
- SSE → one-way streaming
- WebSockets → two-way, fastest

2️⃣ How does the server know when updates happen?

Solutions:

- Pub/Sub (WhatsApp, Slack)
- Stateful servers / hash ring (Google Docs)

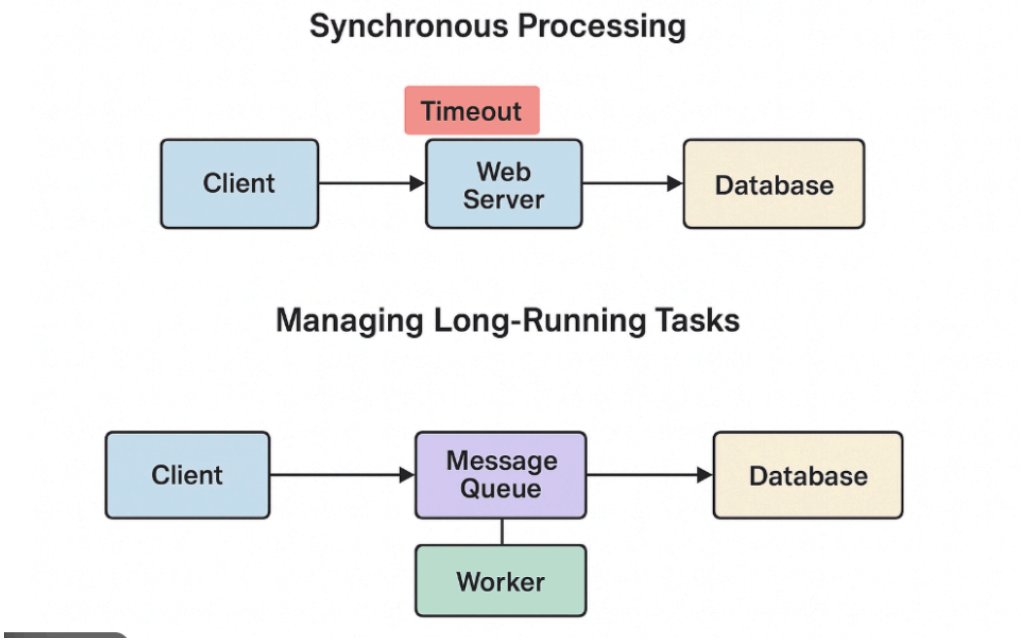
2.) Managing Long-Running Tasks

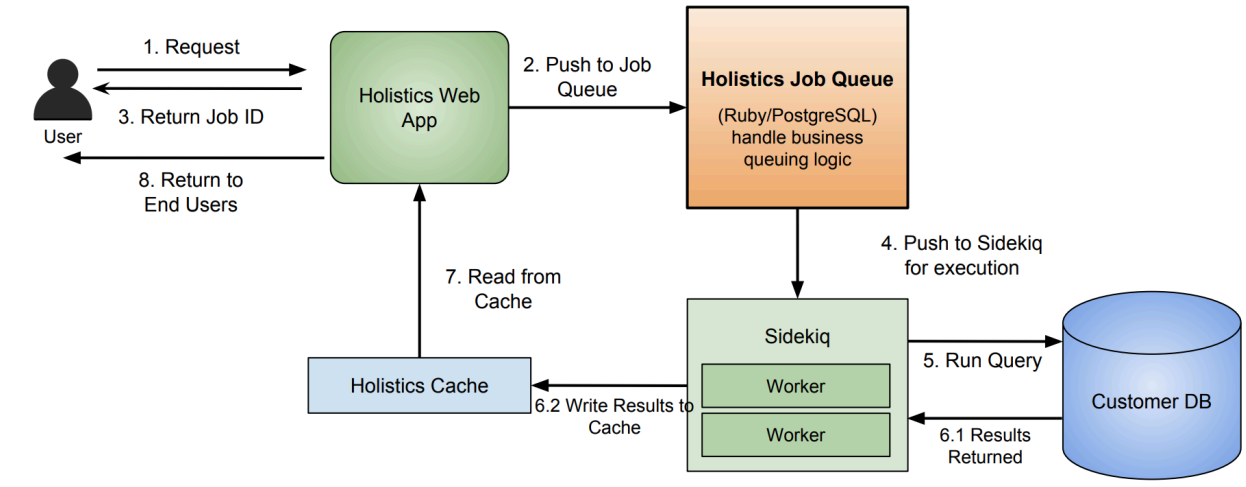


Long Running Tasks

let's explain the **correct diagram** step-by-step in the same deep, intuitive, real-world style as earlier.
This time, I'll directly walk you through every arrow and component in the image so it becomes crystal clear.

⭐ First, Here Is the Architecture You Shared (Mentally Visualized)





holistics.io

The diagram shows this flow:

1️⃣ Client → Server
2️⃣ Server → Database (store job)
3️⃣ Server → Queue (enqueue job)
4️⃣ Workers → Queue (pull jobs)
5️⃣ Workers → Database (update job status)
6️⃣ Client → Server (poll job status)

This is the **standard industry architecture** for handling long-running tasks.

⭐ Why This Pattern Exists (Simple Explanation)

Many operations take **too long to be completed during an HTTP request**, for example:

- YouTube video encoding (minutes)
- Generating a monthly PDF report (seconds/minutes)
- Exporting a CSV of 1 million rows
- Sending 50,000 emails
- Image resizing / applying filters
- Machine learning batch scoring

Browsers **timeout**, servers get overloaded, and users get frustrated.

So instead of executing heavy work immediately, we **accept the request instantly** and do the heavy work **asynchronously** in the background.

⭐ NOW LET'S BREAK DOWN THE DIAGRAM STEP-BY-STEP

(Every arrow explained with real-world examples)

🟩 Step 1 — Client sends request to server

Example:
A user in Google Analytics clicks:
👉 *“Generate monthly traffic report as PDF”*

User → HTTP Request → Server

This request includes parameters like:

- date range
- format (PDF, CSV, Excel)
- filters

🟦 Step 2 — Server responds immediately

In milliseconds, the server sends back:

- **Job ID**
- **Status = PENDING**
- A message like:
  “Your report is being prepared. Check status at /job/{id}.”

This keeps the UI fast and responsive.

✔ Why does this matter?

Users do NOT wait 10–120 seconds for a PDF to generate.
A fast response gives a better experience.

✔ Real examples

- Google Takeout: “We’re preparing your archive.”
- Gmail Export: “We’ll email you when it’s done.”
- Amazon Seller Central report: “Processing…”
- Zomato restaurant bulk campaign: “Campaign scheduled.”

🟧 Step 3 — Server saves the job in the Database

The server writes a record like:

JobID	UserID	Type	Status	Input	CreatedAt
12345	98	PDF_REPORT	PENDING	{dateRange}	now

This allows:

- worker tracking
- user polling
- retrying
- audit logs

Real-life analogy

Like giving you a token number at a bank counter.

🟨 Step 4 — Server pushes job to the Job Queue

💡 **This is the MOST important part.**

Queue systems used:

- Redis Queue
- Kafka
- RabbitMQ
- AWS SQS

The queue acts like:

- a buffer
- a load balancer
- a job distributor

Why use a queue?

Because heavy tasks should:

- not block web servers
- be retried if they fail
- be handled by workers dedicated to heavy processing

Real example

YouTube:
Upload finishes → Job placed in Google's internal Pub/Sub → workers start video transcoding.

🟪 Step 5 — Workers pull jobs from the queue

Workers = background service machines.

Workers continuously:

- read a job
- mark it RUNNING
- execute it
- update the database with COMPLETED or FAILED

Real-life examples

YouTube workers

- Convert video to 144p, 360p, 720p, 1080p
- Generate thumbnails
- Extract metadata

Amazon workers

- Generate settlement reports
- Build inventory reports

Finance systems

- Heavy SQL aggregation → done by workers
- Export PDFs with graphs
- Archive older data

These workers are scalable—if traffic increases → add more workers without touching the web servers.

🟫 Step 6 — Workers update job status in Database

Examples of status transitions:

- **PENDING → RUNNING → COMPLETED**
- **PENDING → RUNNING → FAILED → RETRYING**
- **FAILED → MOVED_TO_DLQ** (Dead Letter Queue)

This makes the system fault-tolerant.

🟩 Step 7 — Client checks job status (polling)

The frontend keeps calling:

```
GET /job/{jobId}/status
```

Server reads status from DB:

```
{
  jobId: 12345,
  status: "COMPLETED",
  resultUrl: "/downloads/report-12345.pdf"
}
```

Or the system can use:

- WebSockets
- SSE

- Email notification

⭐ END-TO-END — REAL LIFE WORKFLOW

Let's say you're on **Zerodha (stock trading)** and request:

📄 *"Download my 1-year trade report."*

Here's what happens internally:

1. **Client** → sends request
2. **Server** → validates & responds instantly
3. **Server** → stores job in DB
4. **Server** → pushes job to Kafka queue
5. **Report Worker** → pulls job
6. Worker generates PDF (takes 20 seconds)
7. Worker updates job as COMPLETED
8. You get an email or UI notification
9. You click "Download"

This is EXACTLY the architecture shown in the diagram.

⭐ When Should You *NOT* Use This Pattern?

This is a very important interview hint.

If the job takes **less than 500ms–1s**, queues add unnecessary complexity.

Examples:

- Insert a record
- Update profile info
- Add item to cart
- Mark notification as read

These should be synchronous.

⭐ Key Concepts You Absolutely Must Mention in Interviews

✔ Job Status Tracking

Maintain PENDING / RUNNING / COMPLETED / FAILED

✔ Retries

Retry failures with exponential backoff.

✔ Dead Letter Queue

Failed jobs go to DLQ → engineer inspects.

✔ Idempotency

Repeating the same job should not cause duplicates.

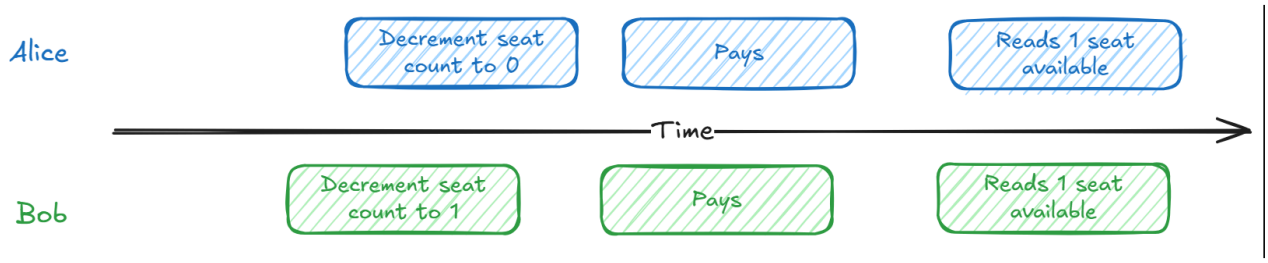
✔ Worker Scaling

Workers autoscale based on queue depth.

⭐ Ultra-Simple Summary (For Interview Use)

“Long-running operations cannot finish inside HTTP requests.
So we accept the request instantly, write a job to DB, enqueue it, and let background workers process it asynchronously.
This makes the system fast, scalable, and fault tolerant.”

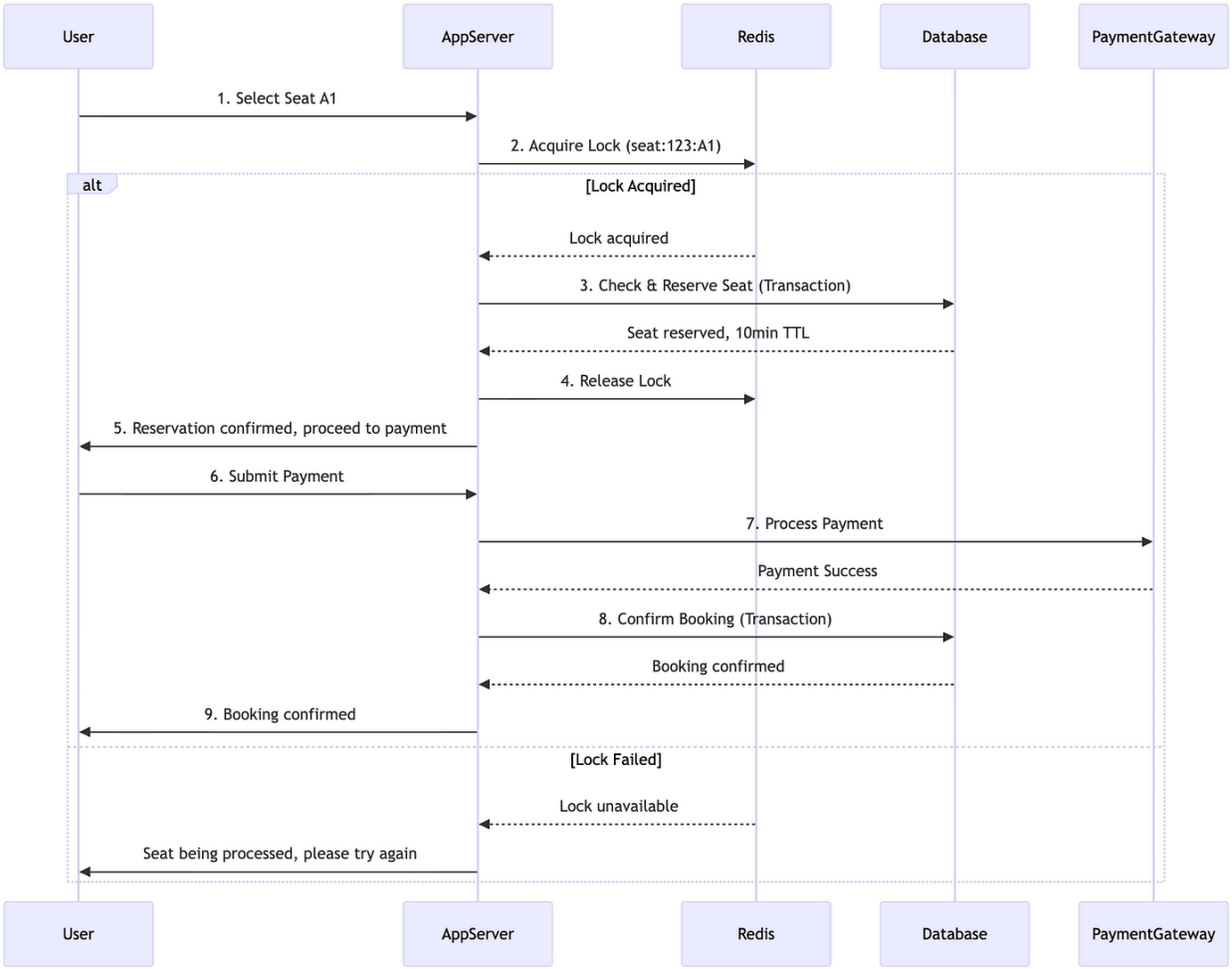
3.) Dealing with Contention

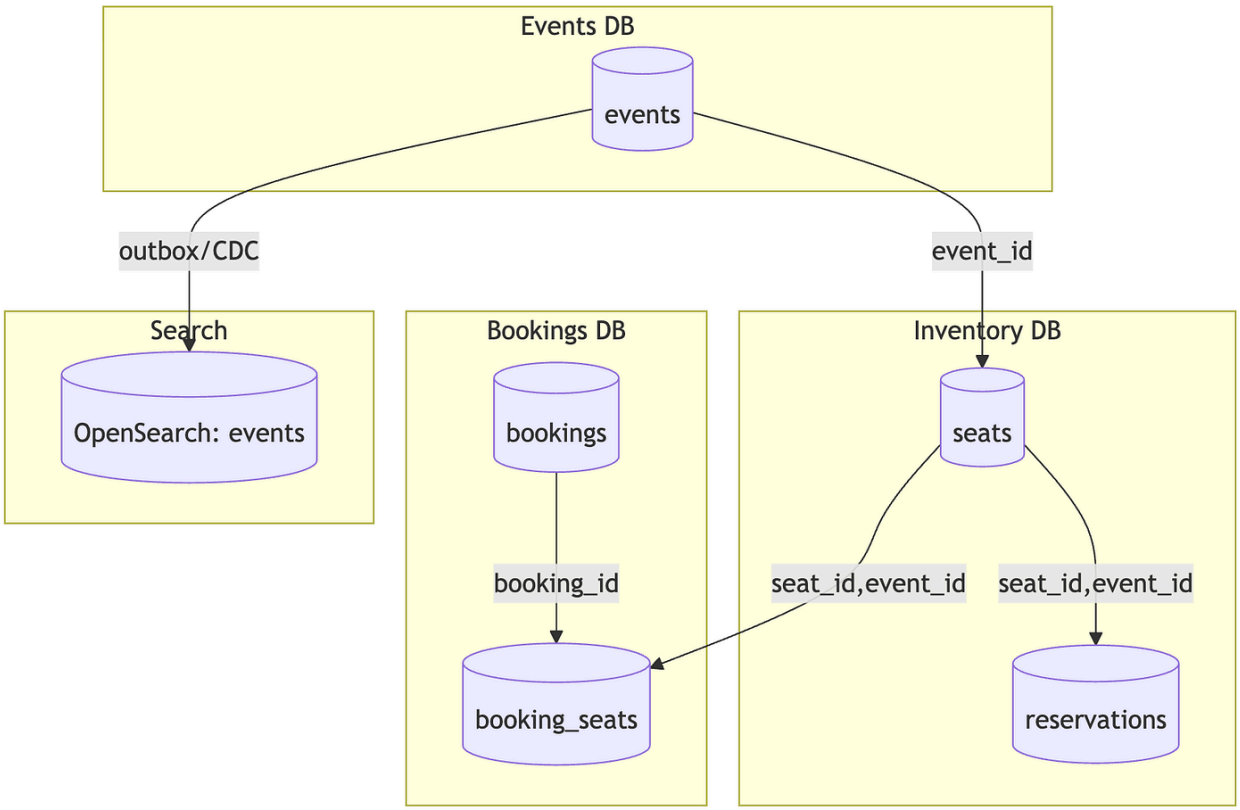


this is one of the **most important System Design topics**, because *race conditions, contention, double booking, and distributed consistency* come up in real systems constantly.

Let’s break down the diagram and the full concept **exactly like before**, with real-world examples (BookMyShow, Amazon Flash Sales, Uber surge pricing, Google Docs edits, seat reservation systems).

⭐ First: What the Diagram Shows (Race Condition Explained)


User
AppServer
Redis
Database
PaymentGateway
1. Select Seat A1
2. Acquire Lock (seat:123:A1)
alt
[Lock Acquired]
Lock acquired
3. Check & Reserve Seat (Transaction)
Seat reserved, 10min TTL
4. Release Lock
5. Reservation confirmed, proceed to payment
6. Submit Payment
7. Process Payment
Payment Success
8. Confirm Booking (Transaction)
Booking confirmed
9. Booking confirmed
[Lock Failed]
Lock unavailable
Seat being processed, please try again
User
AppServer
Redis
Database
PaymentGateway



The diagram shows **Alice** and **Bob** both trying to buy the LAST seat of an event.

There is **1 seat left**.

⭐ What Happens in the Diagram (Step-by-Step)

Alice flow

1. Sees **1 seat available**
2. Decrements seat to **0**
3. Pays
4. Reads again → system still shows **1 seat available (WRONG!)**

Bob flow

1. Sees **1 seat available**
2. Decrements seat to **1** (WRONG, it should become 0!)
3. Pays
4. He also sees **1 seat available**

⭐ Why is this happening?

Because **both users read the same stale value**, then both update based on that outdated value.

This is called a **race condition**.

Without proper locking or atomicity:

```
Read → Compute → Write
```

is NOT safe for concurrent access.

⭐ Real-World Examples Where This Happens

✔ BookMyShow / Ticketmaster

Two users try to buy one last ticket → **overselling** happens if not handled.

✔ Amazon Flash Sales (Lightning Deals)

Multiple buyers try to purchase limited-stock items → requires strict locking.

✔ Airline seat bookings

Two passengers choose the same seat simultaneously.

✔ E-commerce inventory

10 items left, 100 people checkout — how to prevent negative stock?

✔ Banking

Two ATMs withdraw from the same account at the same moment → can cause overdraft.

⭐ The Core Challenge: Contention

Contention happens when **multiple users want to modify the same data at the same time**.

Examples:

- Same seat
- Same wallet balance
- Same stock inventory
- Same row in DB
- Same shared resource

⭐ SOLUTIONS to Contention (Explained with Real Examples)

🟦 1. Atomic Database Operations

(Use when your data fits inside a single database)

Databases support atomic operations like:

```
UPDATE seats
SET available = available - 1
WHERE event_id = 10 AND available > 0;
```

This ensures:

- Only one user can decrement the seat
- If seat == 0 → update fails

Real Example

BookMyShow uses atomic SQL updates with conditional writes.

🟩 2. Optimistic Concurrency Control (OCC)

Used when conflicts are rare.

How it works:

1. Read seat with version number
2. Try to update seat with same version
3. If version changed → someone updated → retry or fail

Example

Google Docs uses OCC for document editing.
If two users edit the same section → merge or reject.

🟥 3. Pessimistic Locking

You lock the row **before modifying it**.

Used in traditional DBs:

```
SELECT * FROM seats WHERE id = 1 FOR UPDATE;
```

Then only one user at a time can book.

Example

Banking transactions often use pessimistic locking.

🟨 4. Distributed Locks

Used when you have **multiple database nodes**, microservices, or distributed systems.

Tools:

- Redis Redlock
- ZooKeeper
- etcd

Real Example

Uber uses distributed locks when assigning drivers to rides.
Only one driver gets the ride, even if many are nearby.

🟧 5. Queue-Based Serialization

Instead of letting all users hit the database:

- Requests go to a queue
- Workers process them **one-by-one**

This is the simplest way to avoid race conditions.

Real Example

Ticketmaster queues customers in a virtual queue during big events.

🟪 6. Reservation System (Soft Locking)

System temporarily holds the seat for a user:

1. Seat becomes “reserved” for 2 minutes
2. User completes payment
3. If payment fails, seat returns to pool

Real Example

BookMyShow, IRCTC, Ticketmaster
Seat gets temporarily locked while user pays.

⭐ What Interviewers Want to Hear

They want to know if YOU understand:

1. **Why contention happens**
2. **Different levels of solutions**
3. **Trade-offs**

4. **When to scale from single-DB to distributed locking**
5. **What guarantees each approach offers**

⭐ Deep Dive Into the Trade-offs

Method	Consistency	Speed	Use Case
Atomic DB updates	Strong	Fast	small systems, single DB
OCC	Medium	High	collaborative editing
Pessimistic locks	Very strong	Slower	banking, critical resources
Distributed locks	Strong	Medium	multi-node systems
Queues	Strong	Slower	high-traffic events
Soft reservation	Medium	Fast	ticketing, e-commerce flash sales

---

⭐ Why Interviewers Care About This

Because when you scale:

- From 1 DB to multiple DBs
- From 1 service to microservices
- From local consistency to distributed systems

…you LOSE the safety guarantees automatically given by a single database.

This is why the prompt warns:

> “Databases are built around problems of contention.
> When you separate your data into multiple databases, you’re giving up a lot.”

Meaning:

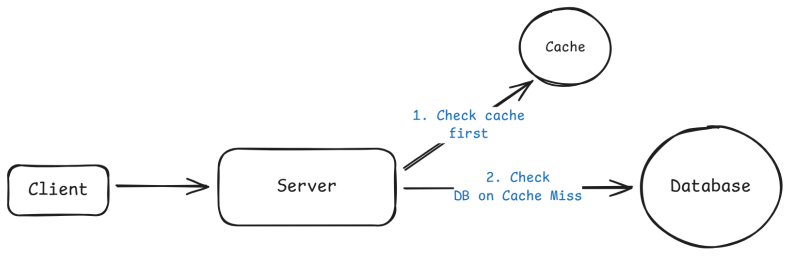
- No automatic locking
- No atomic updates
- No simple transactions
- No single-row consistency

So you must rebuild these mechanisms yourself.

⭐ The Simple Spoken Summary (For Interview Use)

> “Race conditions occur when multiple users try to modify shared data at the same time.
> A simple example is two users booking the last seat. Without locking or atomicity, both can decrement or read stale values, causing overselling.
> The solution is to use database-level atomic updates, optimistic/pessimistic locking, distributed locks, or queue-based serialization depending on scale.
> Start simple with single-DB atomic operations, only move to distributed approaches when required.”

4.) Scaling Reads



Database Read Scaling

⭐ PART 1 — SCALING READS (First Diagram)

The diagram shows:

1️⃣ Client → Server
2️⃣ Server → **Check Cache First**
3️⃣ If MISS → Read DB → Put into cache → Return to user

This is the most common **read scaling pattern** in the world.

🟦 WHY DO READS BECOME THE FIRST BOTTLENECK?

Because:

- Users **read** FAR more than they **write**.
- For Instagram:
  ✔ 1 write per day (post)
  ✔ 500+ reads (view feeds, profiles, stories)

That's why read traffic explodes first.

Read:Write ratio = 100:1 is normal.

🟩 STEP-BY-STEP (Diagram Explanation)

Step 1 — Client asks for data

Example: You open Instagram → need feed/likes/comments.

Step 2 — Server checks cache

Cache could be:

- Redis
- Memcached
- CDN (for images/videos)

Cache returns data in **1–5 ms**
Database takes **5–20 ms** (or more)

So cache reduces load dramatically.

Step 3 — If cache MISS → Read from DB

Example:

```
SELECT * FROM posts WHERE postId=123
```

Step 4 — Put result into cache

Now future requests become faster.

⭐ REAL LIFE EXAMPLES

1. Instagram Feed

- Every time you open the app → reads 50–100 posts
- Each post has multiple attributes (likes, user info, comments)
- All heavily cached

Without caching, Instagram would need **hundreds of DB reads per second per user**, which is impossible.

---

2. Amazon Product Page

Product pages have huge read traffic:

- Title
- Price
- Stock
- Reviews count
- Image URLs

All are cached because users rarely change this info (writes are rare).

---

3. Netflix / YouTube

The thumbnail images and metadata are all cached or served from CDN.

⭐ Why Caching Works So Well

- Fast (1 ms)
- Reduces DB load
- Handles hot keys (like a viral tweet or popular product)

⚠️ Key Problems To Address

🔥 Cache Invalidation

"When data changes, how do we update the cache?"

🔥 Cache Stampede

Millions of users hit DB on cache expiration.

🔥 Hot Keys

One key gets millions of hits → must use partitioned caching or replication.

⭐ Advanced Read Scaling (After Cache)

If caching isn't enough:

1️⃣ Read Replicas

- Multiple database replicas only for reads
- Writes go to primary
- Reads go to replicas
- Helps Instagram, Facebook scale

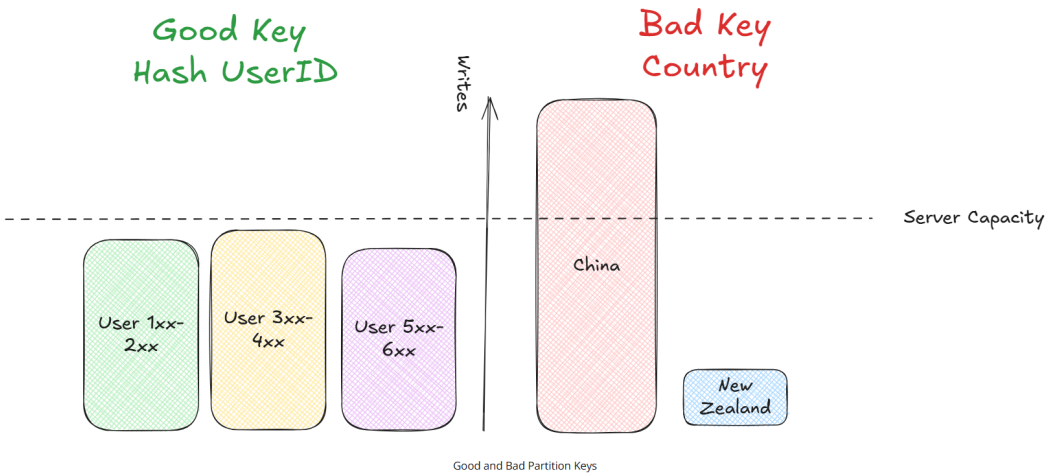
2️⃣ Denormalization

Store ready-to-read data.
Ex: "user_feed" precomputed to avoid expensive JOINs.

⭐ SUMMARY OF SCALING READS

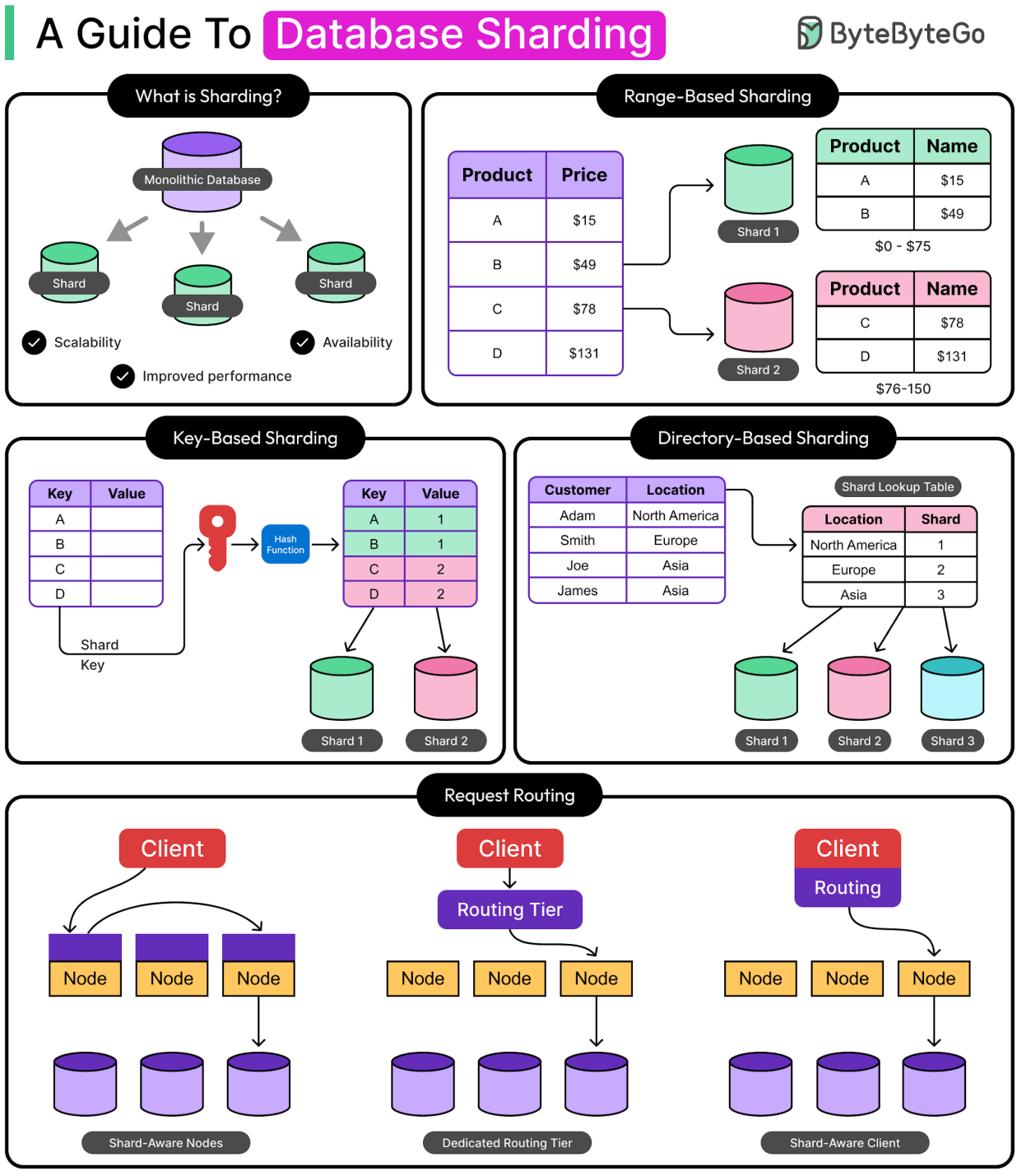
> "Reads scale using caching, read replicas, and denormalization.
> We always check cache first, then fallback to DB on miss.
> Read traffic grows 100× faster than write traffic, so caching is critical."

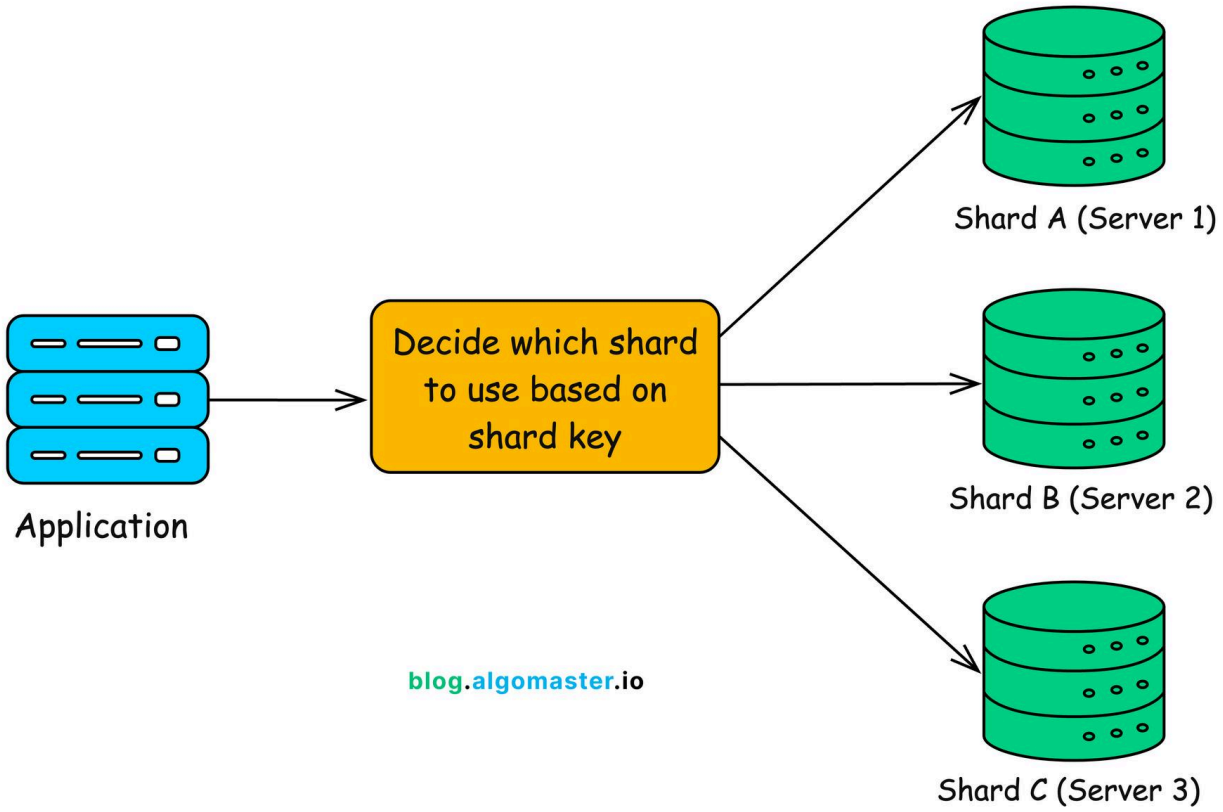
5.) Scaling Writes



Good and Bad Partition Keys

⭐ PART 2 — SCALING WRITES (Second Diagram: Good vs Bad Partition Keys)





This diagram shows **why some partition keys distribute load evenly**, while others create **hot partitions**.

🟥 THE PROBLEM: WRITE BOTTLENECKS

Even with more reads, **writes** eventually become the TRUE limiter.

Examples:

- Instagram: millions of comments per second
- WhatsApp: message writes = 50M/sec globally
- Amazon: millions of real-time inventory updates
- Uber: drivers constantly sending location updates
- TikTok: like counters exploding

At some point, **one DB server cannot handle all writes**.

⭐ SOLUTION: SHARDING (Horizontal Write Scaling)

Sharding = split data across multiple servers based on a **key**.

GOOD SHARD KEY = evenly spreads load

Example in diagram: **UserID hash**

- User 1xx → Shard A
- User 3xx → Shard B
- User 5xx → Shard C

Each shard gets balanced traffic.

🟥 BAD SHARD KEY = uneven load

Example in diagram: **Country**

- China → 1.4 billion users
- New Zealand → 5 million users

If you use country as shard key:

- China shard becomes HOT (overloaded)
- New Zealand shard stays almost empty

This is **catastrophic for scaling writes**.

⭐ Real Life Example Of Bad Shard Keys

Instagram

If they sharded by:

```
shard = country
```

India + US + Brazil = 90% traffic
Other shards = empty

System breaks.

Instead, they use:

```
consistent hashing of user_id
```

Uber

If sharded by:

```
city
```

New York / Delhi / London would be overloaded.
Instead, they use **hashed riderId** or **tripId**.

Amazon

If sharded by:

```
product category
```

Electronics would be overloaded.
Books / accessories almost empty.

Hence they shard using hashed productId.

⭐ OTHER WRITE SCALING STRATEGIES

1️⃣ Write Queues (Buffering Bursts)

For flash sales:

- Huge sudden writes (inventory updates)
- DB would crash
- A queue absorbs spikes
- Workers write gradually

Used by Amazon, Flipkart.

2️⃣ Batching Writes

Instead of writing one row at a time:

- Collect 100 updates
- Write them in a single batch
- Reduces overhead massively

This is used in:

- Analytics systems
- Logging pipelines
- Financial computations

3️⃣ Vertical Partitioning

Split data by feature:

Example for Instagram user table:

- user_profile DB (name, bio, picture)
- user_metrics DB (followers, likes count)
- user_settings DB

This reduces write pressure on a single DB.

⭐ TRADE-OFFS (Important for Interviews)

Technique	Pros	Cons
Sharding	Infinite write scaling	Complex queries, joins become hard
Queue buffering	Handles spikes	Delayed writes
Batching	Higher throughput	Loses real-time accuracy
Vertical split	Better locality	Schema complexity

⭐ SUMMARY OF SCALING WRITES

> "Writes are scaled using sharding, batching, buffering, and partitioning.
> Choosing the right shard key is critical — bad keys (like country) overload one server and collapse the system."

🟩 SUPER SIMPLE MEMORY TRICK

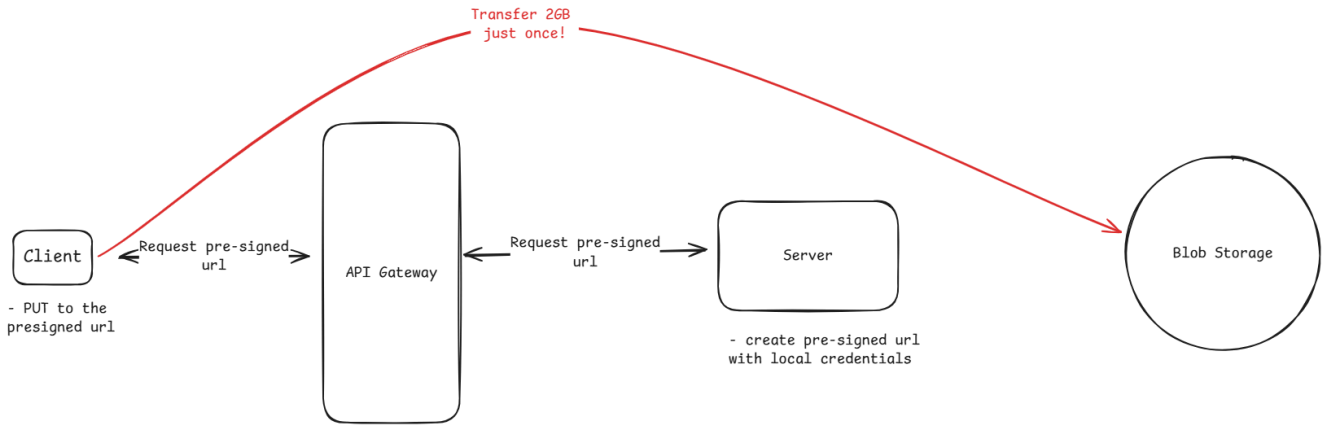
Scaling Reads → Cache First

“Cache → DB → Write-back.”

Scaling Writes → Partition Right

“Spread writes evenly → avoid hot shards.”

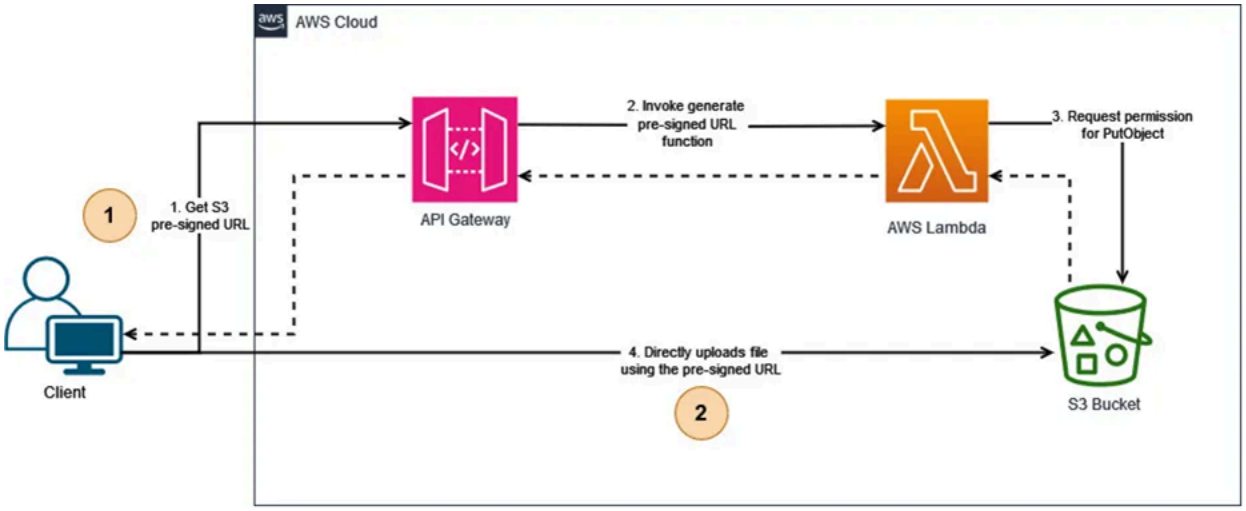
6.) Handling Large Blobs

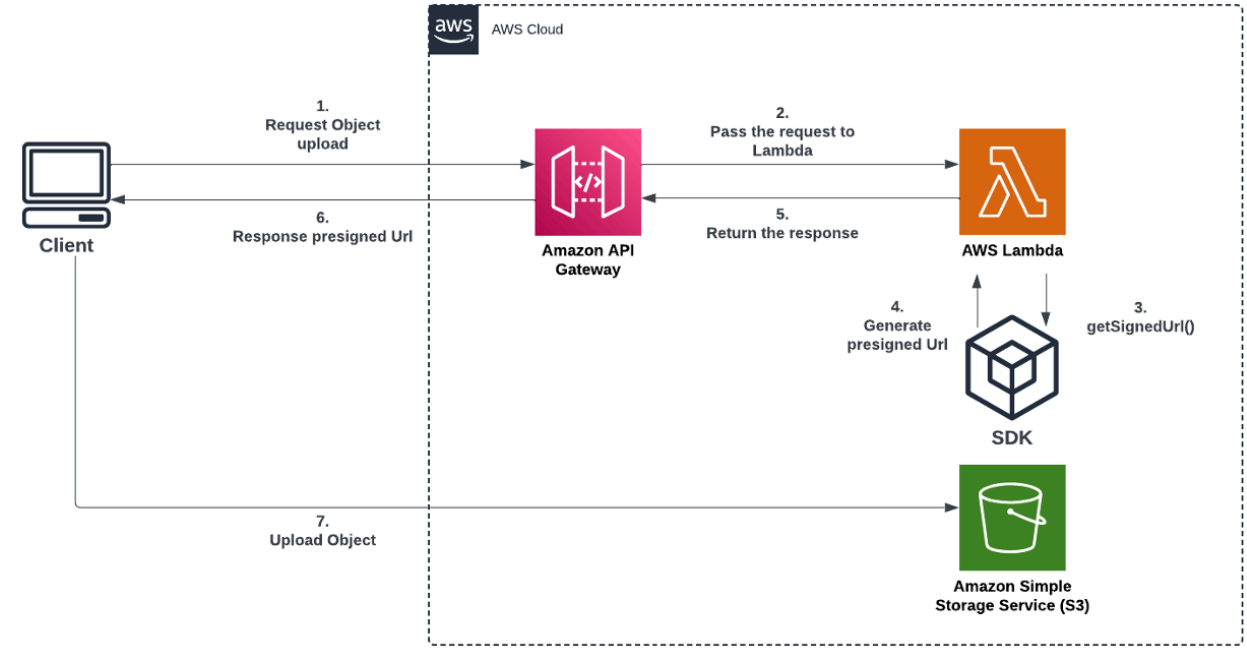


Large Blobs

let’s explain **Handling Large Blobs** in the same clear, deep, real-life style as before.
This is one of the most practical patterns used in modern systems like YouTube, WhatsApp, Google Drive, Instagram, TikTok, Zoom, etc.

⭐ First, Understanding the Diagram (What It Shows)





The diagram shows this workflow:

1️⃣ Client asks the server (via API Gateway) for a **pre-signed URL**
2️⃣ Server generates a **temporary URL** with upload permissions
3️⃣ Client uploads **directly to blob storage** (S3, GCS, Azure Blob)
4️⃣ Server never receives the file data — **only metadata**

The RED ARROW shows:

👉 *Transfer the huge file only once — from client → storage*
No going through your backend server.

This pattern is crucial for uploads of:

- large videos (1GB, 2GB, even 10GB)
- images
- PDFs and documents
- logs / backups
- audio files
- zip archives

⭐ Why Do We Need This Pattern?

If you upload a 2GB video like this:

Client → Server → Storage

The server becomes a bottleneck:

- High bandwidth usage
- Server becomes slow
- Expensive to scale
- CPU/memory wasted
- Latency increases
- Risk of server crash increases

Thus modern systems **never** upload large files through the backend.

Instead, they use **direct client-to-storage uploads**.

⭐ Real-World Examples

✔ YouTube Uploads

When you upload a video:

- The YouTube backend **never actually receives** your raw file.
- The upload goes directly to Google Cloud Storage.
- Backend only receives metadata like:
  filename, duration, uploadId

✔ WhatsApp Sending a Video

Client directly uploads the file to storage servers (Facebook Blob Storage).
Only a reference URL is sent to the WhatsApp backend.

✔ Instagram Post Upload

Photos/videos go directly to storage (example: S3-like).
Server only stores the link and metadata.

✔ Zoom Recording

Recordings upload directly to cloud blob storage using signed URLs.

⭐ Step-by-Step Explanation of the Diagram

Let's walk through it exactly as in the picture.

🟧 Step 1 — Client Requests a Pre-Signed URL

Client → API Gateway:

```
POST /upload-url?fileType=video/mp4&size=2GB
```

The server checks:

- user authentication
- file size limits
- permissions
- file type (e.g., no .exe)

🟦 Step 2 — Server Generates Pre-Signed URL

Server uses its cloud credentials (private keys) to ask S3:

> "Create a special upload URL that is valid for 10 minutes and only allows THIS user to upload THIS file."

S3 returns:

```
https://s3.amazonaws.com/bucket/uploads/video123?signature=XYZ&expires=10min
```

This URL is signed — meaning:

- It gives temporary permission
- Only for upload

- Only for a specific file
- Only for a limited time

Server sends this URL back to the client.

🟩 Step 3 — Client Uploads Directly to Blob Storage

Client → S3 directly:

```
PUT file.mp4 to https://s3.....signature
```

This is the huge 2GB upload.

Advantages:

✔ Upload does NOT touch your backend

Your server doesn't see the file data.

✔ Faster

Client uploads directly to a global CDN endpoint.

✔ More reliable

Blob storage supports:

- resumable uploads
- multipart uploads
- parallel uploads
- retries

✔ Cheaper

Blob storage bandwidth is cheaper than server bandwidth.

🟪 Step 4 — Server Updates Metadata (Not File)

Server now stores:

- file URL
- file size
- owner/userId
- upload timestamp

The server may not even know *exactly* when the upload finished.

So cloud storage sends **notifications** (S3 Events):

- “Upload completed”
- “New file created”
- “Upload aborted”

These events trigger:

- video processing pipeline
- thumbnail generation
- virus scanning
- metadata extraction

⭐ The Most Important Advantages (Interview Points)

🟦 1. Server No Longer a Bottleneck

Without this pattern:

- 100 users uploading 2GB each → 200GB going through your server
  Your app would die.

With pre-signed URLs:

- Server handles only metadata
- Heavy data goes directly to blob storage
- Server CPU and memory stay free for business logic

🟩 2. Better User Experience

Blob storage supports:

- multipart upload
- resume on network failure
- high upload speeds

This ensures smoother 2GB uploads.

🟧 3. Easier to Scale

If 100,000 users upload simultaneously:

- No backend overload
- Storage endpoints handle it naturally
- Blob storage auto-scales

🟨 4. Secure

Pre-signed URLs ensure:

- Temporary, scoped access
- No exposure of server credentials
- File type and size restrictions

⭐ Key Challenges You MUST Mention in Interviews

> These questions are asked in FAANG system design interviews.

⚠️ 1. Synchronization Between Database & Storage

Scenario:

Client uploads the file BUT:

- upload fails
- or client cancels
- or internet disconnects

Your DB may still think the file exists.

Solution:

- S3 event notifications
- background cron jobs
- upload completion callbacks

⚠️ 2. Upload Failures

If upload to S3 fails:

- user must retry
- server might need to re-issue a new pre-signed URL
- partial chunks must be cleaned up

⚠️ 3. File Lifecycle Management

You must handle:

- automatic deletion

- archiving
- moving files to cold storage
- expiring unused uploads

Example:
WhatsApp deletes undownloaded media after 30 days.

⚠️ 4. Security and Signed URLs

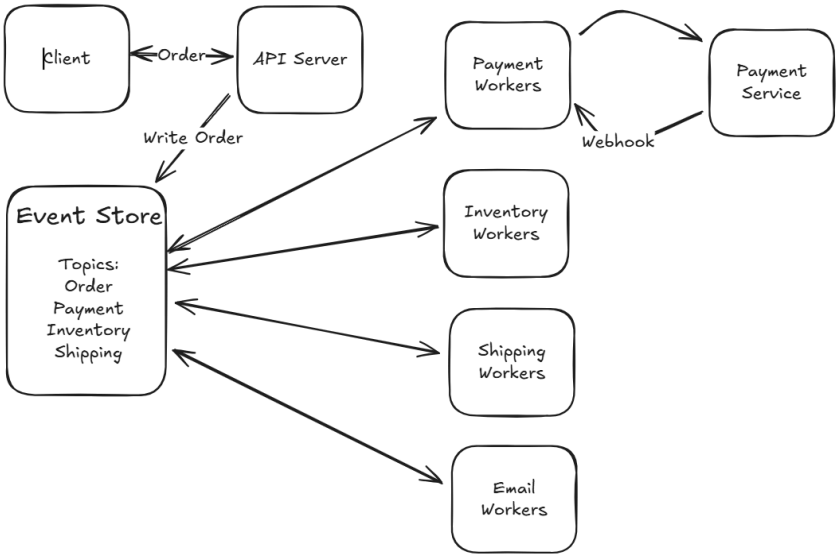
Ensure:

- short expiry (5–15 minutes)
- restricted HTTP methods (PUT/GET only)
- limited file size

⭐ SUPER SIMPLE SUMMARY (For Interviews)

> “For large files, you should never route data through your app servers.
> Instead, generate pre-signed URLs so clients upload directly to blob storage like S3.
> This eliminates server bottlenecks, reduces cost, improves performance, and allows resumable uploads.
> The server only handles metadata, while storage events keep system state consistent.”

7.) Multi-Step Processes

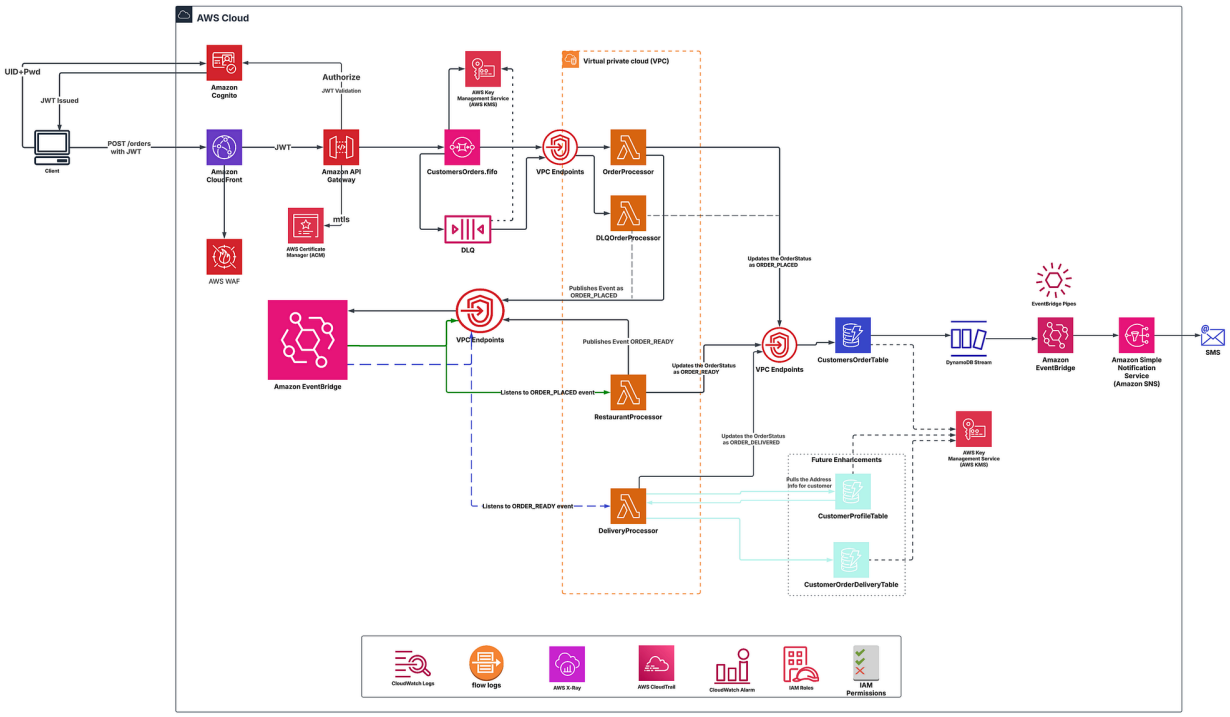


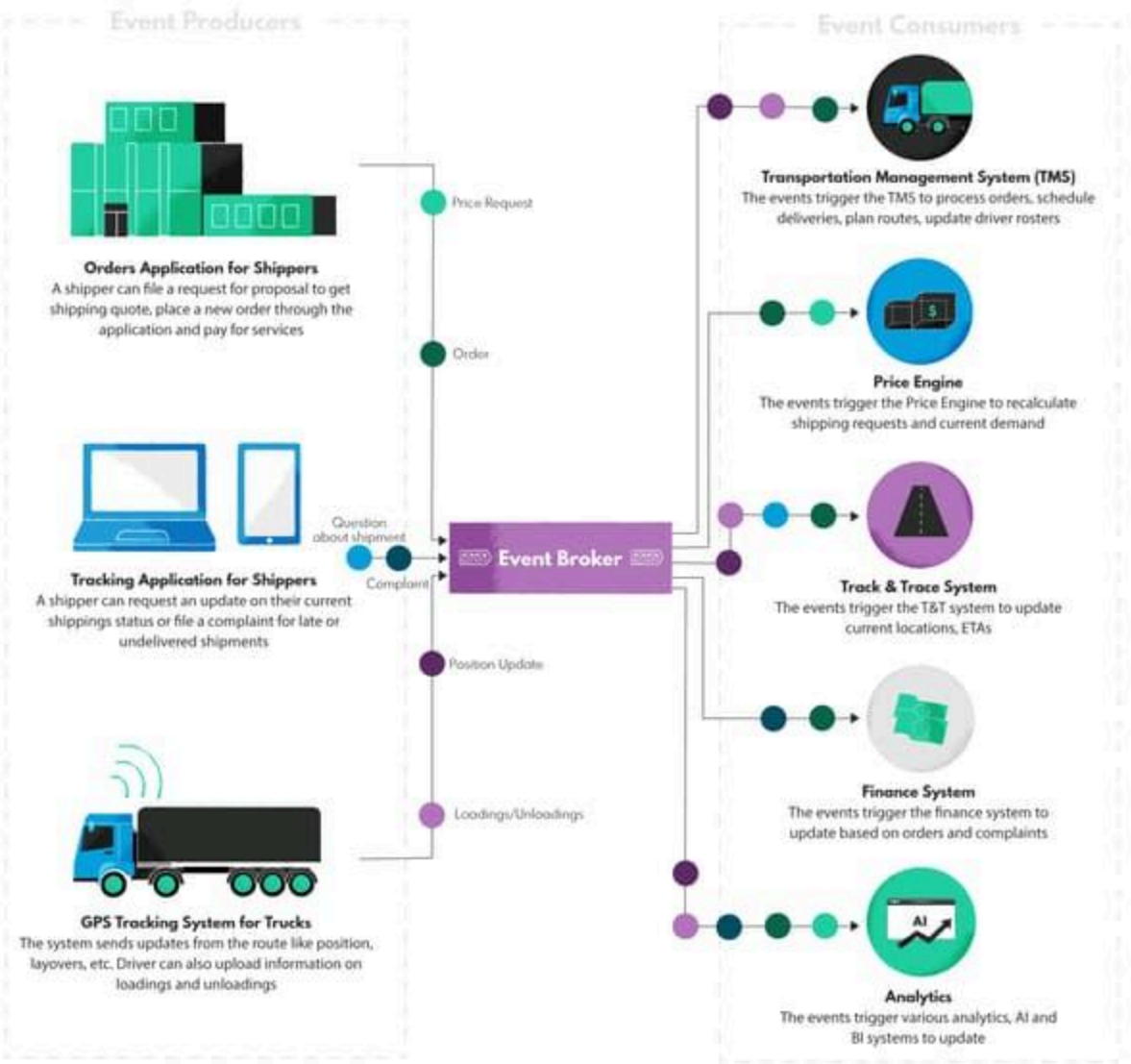
Multi-Step Processes

now let's explain **Multi-Step Processes** using the same deep, simple, real-world style as before.

This is one of the **most important patterns in modern system design**, especially for e-commerce, payments, onboarding, logistics, and any workflow involving multiple services.

⭐ First: What the Diagram Shows (Simple Interpretation)


AWS Cloud
UID+Pwd
JWT Issued
Client
Amazon Cognito
Authorize
JWT Validation
POST /orders with JWT
Amazon CloudFront
AWS WAF
JWT
Amazon API Gateway
mtls
AWS Certificate Manager (ACM)
AWS Key Management Service (AWS KMS)
CustomersOrders.fifo
DLQ
Virtual private cloud (VPC)
VPC Endpoints
OrderProcessor
DLQOrderProcessor
Publishes Event as ORDER_PLACED
Updates the OrderStatus as ORDER_PLACED
VPC Endpoints
Amazon EventBridge
Publishes Event ORDER_READY
Updates the OrderStatus as ORDER_READY
Listens to ORDER_PLACED event
RestaurantProcessor
VPC Endpoints
CustomersOrderTable
DynamoDB Stream
EventBridge Pipes
Amazon EventBridge
Amazon Simple Notification Service (Amazon SNS)
SMS
AWS Key Management Service (AWS KMS)
Updates the OrderStatus as ORDER_DELIVERED
Future Enhancements
Pulls the Address Info for customer
CustomerProfileTable
Listens to ORDER_READY event
DeliveryProcessor
CustomerOrderDeliveryTable
CloudWatch Logs
flow logs
AWS X-Ray
AWS CloudTrail
CloudWatch Alarm
IAM Roles
IAM Permissions



The diagram represents a classic **event-driven multi-step workflow**, often used by:

- Amazon (order + payment + inventory + shipping)
- Flipkart and Meesho (order fulfillment)
- Uber Eats (order → restaurant → courier assignment → delivery)
- Banks (loan approval workflow)
- AWS Step Functions / Temporal workflows

It shows:

1️⃣ Client places an **Order**
2️⃣ API server **writes order event** to Event Store (Kafka-like)
3️⃣ Multiple workers each listen to specific events:

- Payment Workers
- Inventory Workers
- Shipping Workers
- Email Workers
  4️⃣ Each worker updates progress and triggers **next events**

This is the foundation of **Event-Driven Architecture** + **Workflow Orchestration**.

⭐ Why Do We Need This Pattern?

Real-world business flows are **multi-step**, take time, and depend on external systems.

Let's take a real example:

User buys a laptop on Amazon

Steps:

1. Create order
2. Charge payment
3. Reserve inventory
4. Arrange shipping
5. Notify user
6. Update analytics
7. Update fraud systems

Each step:

- depends on the previous step
- may fail
- may take seconds or minutes
- requires retries
- must not lose state
- must not be executed twice

A simple synchronous API call cannot handle this.

Thus, we need a **reliable, fault-tolerant workflow system**.

⭐ Let's Break Down the Diagram Step-by-Step

🟩 Step 1 — Client places an Order

Client → API Server:

```
POST /order
```

Server performs minimal validation:

- product exists
- user is logged in
- not a fraudulent request

Then it writes an **OrderCreated event** to the Event Store.

🟥 Step 2 — API Server writes event to Event Store

Event Store = Kafka / Pulsar / Kinesis / NATS.

Topics include:

- Order
- Payment
- Inventory
- Shipping

Think of it as a **message hub** for the whole order workflow.

Publishing event:

```
ORDER_CREATED {
   orderId: 101,
   userId: 999,
   amount: 2499,
   item: “iPhone 14”
}
```

🟦 Step 3 — Payment Workers listen for ORDER_CREATED

When Payment Worker sees:

```
ORDER_CREATED(orderId=101)
```

It calls the Payment Service (Stripe, Razorpay, Paytm etc):

```
chargeCard(user, amount)
```

After payment succeeds, Payment Service sends a **webhook** back:

```
PAYMENT_SUCCESS(orderId=101)
```

The Payment Worker publishes:

```
PAYMENT_COMPLETED
```

into the Event Store.

🟧 Step 4 — Inventory Workers process PAYMENT_COMPLETED

Inventory Worker listens to:

```
PAYMENT_COMPLETED(orderId=101)
```

It checks inventory DB:

- If in stock → reserve item
- If out of stock → create failure event
- If stock is limited → lock or decrement count

Then publishes:

```
INVENTORY_RESERVED(orderId=101)
```

🟨 Step 5 — Shipping Workers listen for INVENTORY_RESERVED

Shipping Worker handles:

`INVENTORY_RESERVED(orderId=101)`

It creates:

- shipping label
- tracking number
- assigns courier
- calculates estimated delivery

Then publishes:

`SHIPPING_SCHEDULED(orderId=101)`

🟫 Step 6 — Email Workers notify the user

Email Worker listens for:

`SHIPPING_SCHEDULED(orderId=101)`

It sends:

- order confirmation
- shipping updates
- “Your order is on the way!” email

⭐ IMPORTANT: Each Worker is Independent

This is extremely powerful:

- Payment worker can crash without affecting shipping
- Shipping worker can restart without losing events
- Email worker can retry if sending fails
- Inventory worker can scale independently

This enables **fault tolerance**, **high availability**, and **scalability**.

⭐ Why Not Just Use a Long API Call?

Because:

- Payment may take 5 seconds
- Inventory check may take 100ms
- Retry payment 3 times if network fails
- Shipping may take 2 seconds

A long synchronous request would time out.

Also, if the server crashes mid-way, you lose all state.

Event-driven workflows **never lose state**.

⭐ When Workflows Get More Complex → Use a Workflow Engine

Tools:

✔ Temporal

Used by Netflix, Datadog, Descript, Coinbase.

✔ AWS Step Functions

Used for multi-step AWS workflows.

✔ Apache Airflow

Used for batch processing and ETL workflows.

These systems:

- automatically retry
- resume after failure
- maintain state
- track progress
- guarantee exactly-once execution
- provide history/audit log

⭐ Real-Life Example: Amazon Order Flow

Amazon actually does this:

1. OrderCreated → Payment Worker

2. PaymentCompleted → Inventory Worker
3. InventoryReserved → Shipping Worker
4. ShippingScheduled → Notification Worker

If payment fails → workflow rolls back:

- release inventory
- cancel order
- send failure notification

If inventory fails → refund payment.

This rollback logic is called **Saga Pattern**.

⭐ Real-Life Example: Uber Eats Order

Steps:

1. User places order
2. Payment authorized
3. Restaurant receives order
4. Restaurant accepts or rejects
5. Courier assignment
6. Courier picks up
7. Courier delivers
8. Payment captured

Uber built its system on:

- Kafka
- Cadence (Temporal predecessor)
- Event-driven microservices

⭐ Challenges You MUST Mention in Interviews

These are common interview questions.

⚠️ 1. Idempotence

Workers must not duplicate actions.

Example:

- Payment worker must not charge the card twice
- Email worker must not send duplicate emails

⚠️ 2. Retry Logic

If payment service is down → retry intelligently.

⚠️ 3. Event Ordering

Orders should arrive in correct sequence:

- PAYMENT_COMPLETED cannot arrive before ORDER_CREATED

Kafka helps maintain ordering per key (orderId).

⚠️ 4. Error Handling

If inventory fails:

- publish INVENTORY_FAILED
- trigger refund workflow

⚠️ 5. State Tracking

Where is the order at any moment?
Use:

- State machine
- Workflow orchestrator
- Database tracking

⭐ Simple Interview Summary (Say This to Impress)

> “Multi-step processes like order fulfillment require reliable coordination across payment, inventory, shipping, and notification systems. Using an event store with workers allows each step to produce events that trigger the next step. This ensures fault tolerance, retries, and progress tracking. For more complex workflows, we use workflow engines like Temporal or AWS Step Functions, which guarantee exactly-once execution and maintain complete audit history.”

8.) Proximity-Based Services

let's now explain the **Proximity-Based Services** pattern with the same clarity, depth, and real-life examples you've enjoyed so far.

This pattern is extremely important for system design problems like:

- **Design Uber / Ola**
- **Design Swiggy / Zomato delivery allocation**
- **Design Gopuff / Zepto**
- **Design Google Maps Nearby Search**
- **Design Tinder swipe matching**
- **Design Airbnb "find locations near me"**

⭐ WHY DO WE NEED PROXIMITY-BASED SERVICES?

Because many real-world apps need to **find nearby things**, such as:

- Uber → nearest drivers
- Zomato → restaurants near user
- Amazon → warehouses closest to delivery address
- Swiggy → delivery agent closest to restaurant
- Airbnb → listings within 5km
- Tinder → matches nearby

These queries must be:

- **FAST**
- **SCALABLE**
- **ACCURATE**

A naive search (scanning every item) becomes impossible when you have:

- millions of drivers
- millions of restaurants
- millions of stores

You can’t check each one manually.

⭐ FIRST: The Key Technology — Geospatial Index

This is a special index designed to answer questions like:

- “Give me drivers within 2 km radius.”
- “Which restaurants are in this square area?”
- “What are the 5 closest delivery partners to this store?”

Databases that support geospatial indexing:

🟦 PostgreSQL + PostGIS

Very accurate, supports polygons, distance queries.

🟥 Redis GEO Commands

- Very fast
- Great for real-time apps
- Supports radius queries, nearest neighbor

🟩 Elasticsearch (geo_point, geo_shape)

- Good for large-scale indexing
- Supports geofencing, bounding boxes

🟧 MongoDB geospatial index

- Great for location-based apps

⭐ REAL LIFE EXAMPLES (Deeply Explained)

🟦 1. Uber / Ola — Find Nearby Drivers

When you open Uber:

1. App sends your GPS (lat, lng) to backend
2. Uber looks for drivers within 500 meters
3. Ranks them by distance + ETA + rating + load
4. Assigns the nearest driver

How Uber stores driver locations:

Every time a driver moves, the app sends:

```
{
  driverId: 123,
  latitude: 12.9716,
  longitude: 77.5946
}
```

Backend writes this into **Redis GEO** or **PostGIS**, like:

```
GEOADD drivers 77.5946 12.9716 driver123
```

Finding nearby drivers:

```
GEORADIUS drivers 77.5946 12.9716 500m
```

Boom — instantly get all nearby drivers.

🟥 2. Swiggy / Zomato — Find Nearby Restaurants

When you open the app:

- They don't scan ALL restaurants.
- They search only those inside your **geohash region**.

Example:

- Your geohash = "tdt4"
- Look for restaurant IDs indexed under this prefix

Restaurants update location rarely → perfect for geospatial indexing.

🟧 3. Airbnb — Find properties in selected area

User draws a box on the map.

Airbnb uses:

- Elasticsearch geo-shape
- PostGIS polygons

Query:

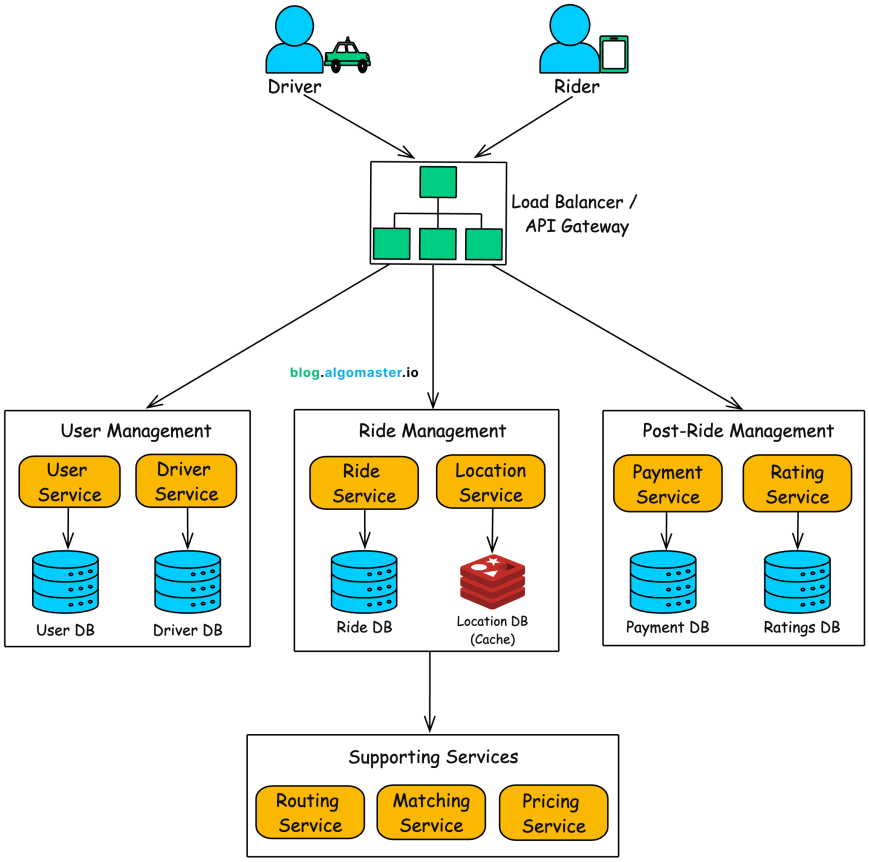
“Give me listings *inside* this polygon”.

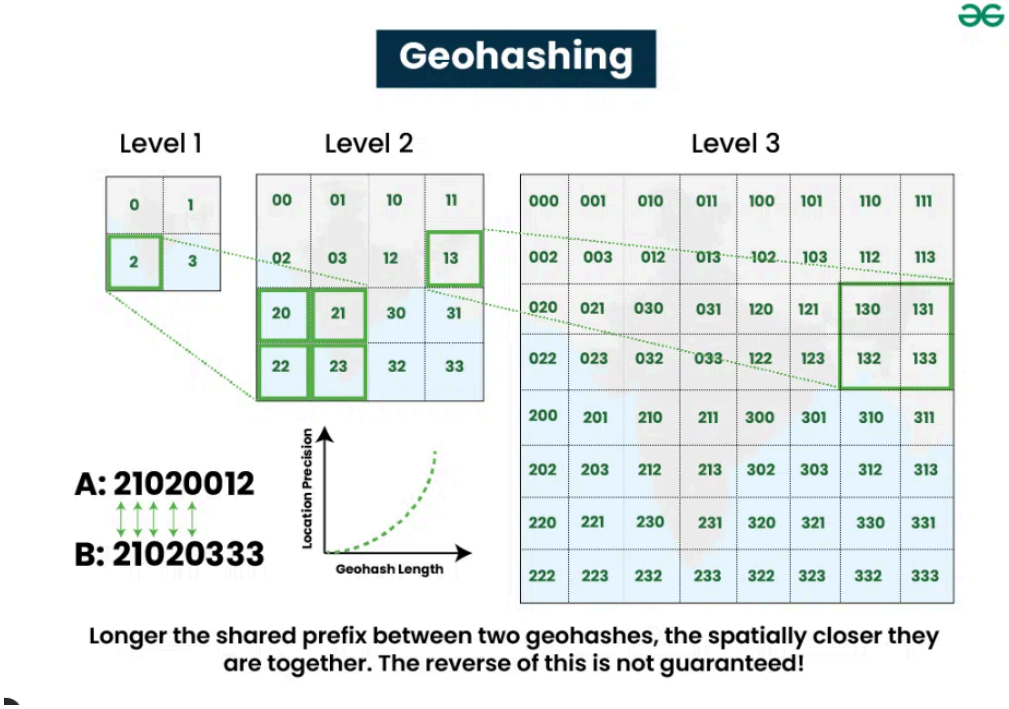
🟩 4. Tinder — Swipe nearby users

Tinder uses **geohashes** with Redis sorted sets.

- Each user is stored in a geohash bucket
- Tinder matches users in the same or nearby geohash region
- This reduces search from millions → thousands → hundreds

⭐ HOW THE ARCHITECTURE WORKS (Conceptually)





The geographic map is divided into **grid cells** (geohashes).

Each grid cell contains entities inside that region:

- drivers
- restaurants
- stores
- users
- delivery agents

When querying for nearby entities:

1. Determine user's geohash region
2. Search that region + neighboring regions
3. Filter by radius/distance
4. Return top N nearest entities

This drastically reduces search space.

⭐ KEY IDEA: You Don't Search Globally

For Uber:

- A rider in Bangalore does NOT need to check drivers in Delhi
- A rider in Koramangala doesn't need to check HSR Layout
- Only search within 2–3 km area

Thus **locality** is your superpower.

⭐ WHEN TO USE GEOSPATIAL INDEXING?

Use it when:

- You have **100k+ entities**
- Location updates are frequent
- Query frequency is high
- Users expect fast responses (100–300ms)

DON'T use it when:

- Only ~1,000 items exist
- Items rarely move
- Accuracy is not important

Example: If you have a small database of 500 stores, just scan them — faster than maintaining expensive indexes.

⭐ COMMON PATTERNS INSIDE PROXIMITY-BASED SERVICES

1. Geohashing

Convert lat/lng into a string bucket.

Example:

`12.97, 77.59 → "tdt482"`

Nearby coordinates share prefixes.

Used by:

- Uber
- Swiggy
- Tinder

---

2. Quadtrees / K-d Trees

Special trees that divide space into quadrants.

Used in:

- Google Maps
- GIS systems

---

3. Distance-based indexing

Compute bounding box around user:

```
lat +/- delta
lng +/- delta
```

Filter candidates inside the box.

⭐ KEY CHALLENGES (What Interviewers Ask)

⚠️ 1. ENTITY MOVEMENT (Drivers keep moving)

Driver location updates come every 2–5 seconds.

Need to update geospatial index efficiently.

⚠️ 2. LOAD BALANCING

Some areas are HOT:

- Airports
- Tech parks
- Malls
- Concert venues

Need to scale those partitions more.

⚠️ 3. LOCATION ACCURACY

GPS can be inaccurate inside buildings → must handle noisy coordinates.

⚠️ 4. QUERY RADIUS SELECTION

Too small radius → no drivers
Too large radius → slow search + far-away drivers

⚠️ 5. DATA VOLUME

Millions of drivers → need memory-efficient geospatial storage.

⭐ SUPER SIMPLE SUMMARY (For Interview Use)

> “Proximity-based services rely on geospatial indexes to quickly find nearby entities like drivers, restaurants, stores, or users.
> Instead of scanning millions of items, we divide the world into geo-regions (geohash or grid) and index entities inside them.
> Queries become fast and local. Technologies like PostGIS, Redis GEO, and Elasticsearch geo-query enable radius queries and nearest-neighbor searches.
> Use geospatial indexes only when scale demands it — otherwise simple scans are more efficient.”

9.) Pattern Selection

Absolutely — let’s explain **Pattern Selection**, the final concept in the Common Patterns section, with the same clarity, structure, and real-life depth as all the previous ones.
This one is extremely important because it teaches **how to combine patterns** and **how to avoid over-engineering**, both of which interviewers care about.

⭐ What “Pattern Selection” Really Means

In real systems — and in system design interviews — **you almost never use just one pattern**.

Instead, you combine multiple patterns together to solve different parts of a bigger problem.

This is exactly what the paragraph is saying:

> *"These patterns often work together to solve complex system design challenges."*

Each pattern solves ONE type of challenge:

Pattern	What it solves
Large Blobs	Handling big file uploads efficiently
Long-Running Tasks	Background processing of slow/expensive work

Realtime Updates	Live progress/status updates
Multi-Step Processes	End-to-end workflow coordination
Scaling Reads	Handling huge read traffic
Scaling Writes	Handling huge write traffic
Contention	Avoiding race conditions
Proximity-Based Services	Geo-search (Uber-style)

A well-designed system picks the **right mix** depending on the requirements.

⭐ REAL LIFE EXAMPLE: DESIGNING A VIDEO PLATFORM (YouTube / TikTok)

This is the best example to illustrate how patterns combine.

Let's walk through it step-by-step:

🟦 1. Large Blobs — Uploading a 2GB video

When a user uploads a video:

- App requests a **pre-signed URL** from server
- Client uploads video **directly to cloud storage** (S3, GCS, etc.)

✔ Server does NOT handle the 2GB file
✔ Cheap, scalable storage
✔ Supports resumable uploads

Pattern used:
👉 *Large Blobs / Pre-signed URL pattern*

🟩 2. Long-Running Tasks — Video Transcoding

Once the video is uploaded:

- A background worker picks up the metadata
- Starts transcoding into formats like 144p, 360p, 720p, 1080p
- Extracts thumbnails
- Runs content checks (copyright, safety, ML models)

This may take minutes and cannot be done in the request.

Pattern used:
👉 *Long-Running Tasks*

Workers scale horizontally based on workload.

🟧 3. Realtime Updates — Showing Upload & Processing Progress

While transcoding happens, user sees:

- "Uploading…"
- "Processing 25%"
- "Your video is almost ready…"
- "HD versions still processing"

This uses:

- WebSockets
- SSE (Server-Sent Events)
- Polling

Pattern used:
👉 *Realtime Updates*

🟨 4. Multi-Step Processes — Coordinating the Entire Pipeline

A video upload is not a single step.
It's a **workflow** made of multiple dependent steps:

1. Upload completed
2. Metadata saved
3. Transcoding started
4. Thumbnails generated
5. Transcoding completed
6. Video published
7. Notification sent to followers

These steps must:

- retry on failure
- be durable
- not lose state

- run in order
- support rollback if needed

Pattern used:
👉 *Multi-Step Processes (Orchestration / Saga / Temporal)*

YouTube actually uses a workflow orchestrator internally.

⭐ See how all patterns fit together?

This is what the text means by:

> **"A video platform might use multiple patterns together."**

Task	Pattern
Upload video	Large Blobs
Transcode video	Long-Running Tasks
Show progress	Realtime Updates
End-to-end workflow	Multi-Step Processes

This is the essence of pattern selection.

⭐ SECOND EXAMPLE: Ordering a Product on Amazon

1. Place order → synchronous API

2. Process order → write event

3. Charge payment → Long-running task + multi-step

4. Reserve inventory → Contention + Multi-step

5. Notify user → Realtime Updates

6. Package + ship → Multi-step process

7. Track package → Realtime location updates

8. Deliver → update inventory + analytics (write scaling)

This uses MANY patterns:

- Scaling Reads (product details, reviews)
- Scaling Writes (order DB, inventory DB)
- Contention (inventory locking)
- Long-Running Tasks (payment verification)
- Multi-Step Processes (order workflow)
- Proximity-Based (nearest warehouse)
- Realtime Updates (delivery tracking)

A system like Amazon literally uses **all patterns together**, but only where needed.

⭐ The BIG INSIGHT (What interviewers want to hear)

A good system designer:

✔ identifies which problem exists

✔ applies the correct pattern

✔ avoids unnecessary complexity

The paragraph says:

> **Start with simpler approaches and add complexity ONLY when required.**

This is gold in interviews.

Bad candidates:

- start with Kafka, distributed locks, 5 microservices, 3 caches (over-engineering)

Good candidates:

- start simple
- add patterns only when scale or requirements demand

Let's demonstrate:

⭐ Example: Designing a Notification System (Interview-Style Thinking)

Step 1 — Start simple:
Use polling every 10 seconds → easy to implement.

Step 2 — Add caching:
When reads increase → reduce DB load.

Step 3 — Add WebSockets:
When real-time is required.

Step 4 — Add workers + queues:
When sending millions of notifications.

Step 5 — Add sharding:
When write volume becomes huge.

You evolved the architecture **by selecting patterns as needed**.

THIS is exactly what interviewers look for.

⭐ Why "Pattern Selection" Shows Senior-Level Thinking

Because senior engineers:

- see common patterns
- know when to use which
- avoid premature optimization
- build modular, scalable designs

Interviewers evaluate:

- Is the candidate aware of trade-offs?
- Do they know when NOT to use a pattern?
- Do they scale sensibly?
- Do they keep the system simple?

⭐ SUPER SIMPLE SUMMARY (Use this in interviews)

> “A good system design is not about memorizing one perfect architecture, but about selecting the right patterns.
> Most real-world systems combine multiple patterns — for example, a video platform uses Large Blobs for upload, Long-Running Tasks for transcoding, Realtime Updates for progress, and Multi-Step Processes for orchestration.
> The real skill is knowing when to use each pattern and not over-engineering when a simple solution is enough.”

here is a **clean, interview-ready cheat sheet** summarizing ALL common system design patterns, followed by a **flowchart to help you decide which pattern to use** during a design interview.

I'll make this extremely clear, structured, and memorable.

⭐ SYSTEM DESIGN PATTERNS — COMPLETE CHEAT SHEET

(Perfect for interview preparation & rapid recall)

🟩 1. Scaling Reads (Read Optimization)

When to use:

Your database cannot handle the number of read requests.

What it solves:

High traffic reading the same data repeatedly (e.g., Instagram feed, Amazon product info).

Tools / Techniques:

- Cache first → Redis / Memcached
- Read replicas
- CDN for static content
- Denormalization
- Materialized views
- Query indexing

Example:

Instagram feed = 100 reads for every write.

🟥 2. Scaling Writes (Write Optimization)

When to use:

Your database can no longer handle write-heavy load.

What it solves:

High write throughput (e.g., TikTok likes, Twitter tweets).

Techniques:

- Sharding (hash by userId, orderId)
- Good vs bad partition keys
- Write buffering with queues
- Batching writes
- Vertical partitioning

Example:

Twitter writes go to sharded clusters by userId.

🟦 3. Caching Pattern

When to use:

Repeated data access slows down your DB.

Benefits:

- Ultra-fast reads (1–5 ms)
- Reduces DB load
- Smooths peak traffic

Cache strategies:

- Cache-aside (most common)
- Write-through
- Write-back
- TTL-based caching
- Anti-stampede protection

Example:

Amazon product page caches: title, price, rating, image URLs.

🟪 4. Dealing with Contention

When to use:

Multiple users try to modify the same resource simultaneously.

(Common in seat booking, inventory updates, banking.)

Solutions:

- Atomic DB updates
- Optimistic Concurrency Control

- Pessimistic locks
- Distributed locks (Redis, Zookeeper)
- Queue-based serialization
- Soft reservations

Example:

BookMyShow prevents double-booking seats using atomic DB operations.

🟨 5. Long-Running Tasks

When to use:

Operations take too long for a single HTTP request (over 2–3 seconds).

Examples:

- Video transcoding
- PDF generation
- ML processing
- Bulk emails

Architecture:

- Client → Server
- Server enqueues job
- Worker processes asynchronously
- Client checks job status

Tools:

Redis queues, Kafka, Celery, SQS, Sidekiq.

🟧 6. Realtime Updates

When to use:

You need to push updates to users instantly.

Examples:
Chat apps, stock tickers, Uber driver movement, notifications.

Protocols:

- Polling
- WebSockets
- SSE
- Long polling

Backend support:

- Pub/Sub
- Message brokers
- State tracking servers

🟫 7. Handling Large Blobs (Videos, Images, Files)

When to use:

Uploading large files (1GB+), images, videos, logs.

Architecture:

- Client asks server for pre-signed URL
- Server issues temporary upload URL
- Client uploads directly to S3/Blob Storage
- No server bottleneck

Benefits:

- Cheap
- Scalable
- Fast
- Resumable

Examples:

YouTube uploads, WhatsApp media, Google Drive file uploads.

🟦 8. Multi-Step Processes (Workflow Orchestration)

When to use:

Your operation has many dependent steps.

Examples:
Order → Payment → Inventory → Shipping → Notification

Tools:

- Kafka event-driven workflows
- Temporal
- AWS Step Functions
- Sagas (choreography or orchestration)

Solves:

- retries
- rollbacks
- exactly-once execution
- persistent workflow state

🟩 9. Proximity-Based Services (Geo-Queries)

When to use:

You need to find nearby entities.

Examples:
Uber drivers, Zomato restaurants, Tinder matches.

Tools:

- Geohashing (Uber, Tinder)
- Redis GEO
- PostGIS

- Elasticsearch geo queries

Workflow:

- Index entities by location
- Search only local regions
- Limit results by distance

🟧 10. Large-Scale Pattern Selection (Meta Pattern)

What it means:

Systems combine many patterns.

Example: **YouTube**

- Upload video → Large Blobs
- Transcode → Long-Running Tasks
- Show progress → Realtime Updates
- Manage workflow → Multi-Step
- Serve thumbnails → Caching

The art is knowing **which pattern to apply when**.

⭐ THE MASTER SYSTEM DESIGN PATTERN FLOWCHART

(Use this mental flowchart during interviews to choose the right approach)

Follow this step-by-step mental decision tree ⬇️

🔵 STEP 1: What is your bottleneck?

✔ Too many reads → Scaling Reads

✔ Too many writes → Scaling Writes

✔ Slow operations → Long-Running Tasks

✔ Multiple-step workflow → Multi-Step Processes

✔ Need instant updates → Realtime Updates

✔ Need large file handling → Large Blobs

✔ Concurrent modification → Contention

✔ Need nearby search → Proximity-Based

🔵 STEP 2: Is the operation synchronous or asynchronous?

✔ Must respond fast?

→ Use **caching**, **read replicas**, **pre-signed URLs**, or **queues**

✔ Can run in background?

→ Use **Long-Running Tasks** or **Multi-Step Processes**

🔵 STEP 3: Is the data volume huge?

✔ Yes → Sharding + Caching + Indexing

✔ No → A single DB with indexes is enough

🔵 STEP 4: Does the client need updates in real time?

✔ Yes → WebSockets / SSE → Realtime Updates

✔ No → Polling is enough

🔵 STEP 5: Is the workflow sequential with dependencies?

✔ Yes → Multi-Step Processes, Sagas, Temporal

✔ No → Simple queue or direct call

🔵 STEP 6: Are files huge (100MB–5GB)?

✔ Yes → Large Blob handling (direct-to-storage upload)

✔ No → Upload through backend is OK

🔵 STEP 7: Does location matter?

✔ Yes → Proximity-Based Services

✔ No → Normal indexing

🔵 STEP 8: Is the system globally distributed?

✔ Yes →

- Edge caching
- CDN
- Geo-replication
- Geo-indexing
- Partition awareness

✔ No → Keep it simple

⭐ PATTERN SELECTION EXAMPLE (INTERVIEW-LEVEL THINKING)

Let's say you are asked:
"Design Uber."

Using the flowchart:

✔ Location-based?

→ Use **Proximity-Based Services**
→ Geohash + Redis GEO

✔ Multi-step?

→ Yes: booking → driver acceptance → payment → trip completion
→ Use **Multi-Step Processes / Event-driven**

✔ Real-time updates?

→ Yes: driver movement
→ **WebSockets**

✔ Scaling reads?

→ Yes: checking nearby drivers
→ **Caching** + **Geospatial index**

✔ Scaling writes?

→ Yes: driver location updates every 3 seconds
→ **Write-optimized storage**

✔ Long-running tasks?

→ No, Uber does not do long video transcoding
→ Skip this pattern

Boom.
You designed Uber in 20 seconds using pattern selection.

⭐ FINAL CHEAT STATEMENT (Use this in interviews)

> “System design is pattern selection. Every architecture is just a combination of patterns applied at the right places.
> My approach is to identify the bottleneck (reads, writes, coordination, latency, file size, concurrency) and choose the simplest pattern that satisfies the requirements.
> I only add complexity when scale or constraints demand it.”