Examples

Monday, 15 September 2025 9:55 PM

----------URL Shortner----------

Ask:

- What is the traffic you are looking for? 'X per second'
- Retension period you are looking for? '10 years'
- What all char should we include in short url(num/alpha/upper/special char etc.,)? 'lower and upper alpha + number'(26 + 26 + 10 = 62)

Process:

- Find the total character and find the length of short url which should be greater than
  - X * 60 * 60 * 24 * 365 * 10 = Y
  - 62 ^ 7 > Y
- When 2 instances create same short url its collision, we can check in the table whether available and retry insert but its not efficient. We need to come up with predictable way to generate a shorturl knowing that there will not be any collision at all
- **Redis** is one option which will generate unique number auto increment and give it back, we just convert it from base 10 to base 62 and return back. we can keep multiple redis and give unique series to start with so that they don't cause duplicates again.. We need a manager
- Another efficient way is token service which assigns particular range of number to an instance which can be maintained in a mysql. The instance will increment one by one and once its over then it again asks for token service and it gives another. We can have multiple token service to avoid single pointer of failure

Beyond what they asked for :

- Now what ever they asked for is covered but can we make some analytical data from all those requests like from which country, which device, ip etc..,?
- We can store it in kafka but might increase latency?
  - Asynchronous call
  - Better option, add it to a queue and flush to kafka every few seconds or upon reaching a threshold
- How to do analytics?
  - Naïve - dump into hadoop and use queries to develop
  - Spark streaming job and aggregates and provides output which is stored in a db

Design:

- Cassandra preferred for large-scale writes/reads.
- MySQL can work with sharding, but less optimal.
- Analytics : Hadoop with query(Amature) or Spark streaming job storing the summary into a table

Take away:

- Always clarify **scale** (requests/sec, retention time).
- Calculate short URL **length vs character set**.
- Address **collision avoidance** explicitly.
- Avoid **single points of failure**.
- Highlight **analytics/observability** as value-add.
- Tradeoffs:
  - Simplicity vs efficiency (e.g., lost tokens).
  - Strong consistency vs availability.

----------Hotel booking----------

Ask (Questions to clarify with interviewer)

- What scale are we designing for?
  - Number of hotels, rooms, bookings per second.
- What level of consistency is required in bookings?
  - Strong consistency (no double booking) or eventual OK?

- What are the latency expectations for search & booking flows?
- What's the expected geographical distribution of users and data centers?
- Should the system support dynamic pricing and analytics out of the box?
- What payment methods and failure/retry policies are required?

Process (Thinking process & complex logic)

- **User roles**: Hotels (onboard, manage properties, view revenue) vs. Travelers (search, filter, book, view trips).
- **Data model**:
  - Hotels/rooms/facilities in relational DB.
  - Active bookings in MySQL (with ACID guarantees).
  - Archived/completed bookings in Cassandra for scalable reads.
- **Search flow**:
  - Hotel updates → Kafka → Search consumer → Elasticsearch.
  - Supports fuzzy search, filters (price, tags, location, date).
- **Booking flow** (critical consistency):
  - Check available_rooms in MySQL.
  - Create booking in **RESERVED** state + decrement available count in same transaction (constraint: quantity ≥ 0).
  - Hold reservation via **Redis TTL** (e.g., 5 mins).
  - If payment succeeds → mark **BOOKED** + invoice id.
  - If payment fails/TTL expires → mark **CANCELLED** + restore availability.
  - If TTL expires before payment success arrives → decide policy (refund vs. reallocate).
- **Notifications**: Kafka → Notification service (confirmations, cancellations, invoices).
- **Analytics**: Push all Kafka events into Hadoop/Spark for business insights (top hotels, revenue, demand/supply pricing).

Beyond what they asked for (Out-of-box functionalities)

- **Dynamic pricing**: Use analytics (Spark/Hadoop) to adjust price between min_price and max_price based on demand.
- **Booking archival**: Move terminal bookings to Cassandra for cheap storage + scalable reads.
- **Regional distribution**: Split into regions (e.g., Americas vs. Asia) with local primaries for lower latency + DNS failover.
- **Monitoring & observability**: Use Grafana/alerts on CPU, memory, disk usage of Redis, ES, Kafka.
- **Caching strategies**: Redis caching for booking queries (critical path); optional for hotel CRUD (non-critical).

Design (Key systems picked)

- **Hotel Service**: CRUD APIs → MySQL (clustered).
- **Images**: Stored in **CDN**, only URLs in DB.
- **Search Service**: Elasticsearch (or Solr) for fuzzy search.
- **Booking Service**: MySQL (ACID + transactions), Redis TTL for reservations.
- **Archival Service**: Cassandra for completed/cancelled bookings.
- **Notification Service**: Consumes Kafka events → notifies hotels & users.
- **Analytics**: Kafka → Spark streaming/Hadoop → reporting DB.
- **Scalability**: All services horizontally scalable, Kafka & Hadoop cluster scale independently.

APIs & DB Schema (from transcript)

Hotel Service APIs

- POST /hotels → Create hotel (onboarding).
- GET /hotel/{hotel_id} → Fetch hotel details.
- PUT /hotel/{hotel_id} → Update hotel info.
- PUT /hotel/{hotel_id}/room/{room_id} → Update/create room details.

Hotel DB (simplified)

- **hotel**(id, name, locality_id(FK), description, original_images, display_images, is_active).
- **rooms**(room_id, hotel_id(FK), display_name, is_active, quantity, price_min, price_max).
- **facilities**(facility_id, description).
- **hotel_facilities**(hotel_id, facility_id).
- **room_facilities**(room_id, facility_id).

Booking Service APIs

- POST /book → Creates a booking request.
  - Inputs: user_id, room_id, quantity, start_date, end_date.
  - Price is server-side (not client-supplied).

Booking DB (simplified)

- **available_rooms**(room_id, date, initial_quantity, available_quantity CHECK ≥ 0).
- **booking**(booking_id, user_id, room_id, start_date, end_date, number_of_rooms, status [RESERVED | BOOKED | CANCELLED | COMPLETED], invoice_id).

Important MySQL (InnoDB) properties we're using

- **ACID transactions**
  - Atomically: *insert booking in RESERVED* **and** *decrement available_rooms* happen in one commit.
  - If payment fails/TTL expires, we **rollback/compensate** predictably.
- **Row-level locking + Next-key (gap) locks**
  - Prevents two concurrent requests from grabbing the **last room**.
  - The SELECT … FOR UPDATE on the availability row(s) serializes writers under contention and avoids phantom inserts in the same range.

Take away (Lessons & principles)

- Always clarify **scale assumptions** (no. of hotels, rooms, bookings/sec).
- **Critical consistency**: rely on **RDBMS transactions + constraints** for inventory.
- Use **Redis TTL** for temporary reservation, handle expiry vs. payment races explicitly.
- **Decouple reads/writes**: MySQL for source of truth, Elasticsearch for search, Redis for cache, Cassandra for archival.
- **Event-driven** with Kafka ensures eventual sync between services and enables analytics.
- **Avoid single points of failure**: multiple Redis/token ranges, multi-region DCs.
- **Trade-offs**:
  - Cost (extra Redis/cluster) vs. performance gain.
  - Simplicity (retry logic) vs. efficiency (transactional guarantees).
  - Strong consistency (MySQL) vs. scalability (Cassandra).

How Redis fits these roles

- **Millisecond operations + TTLs** → perfect for short reservation holds.
- **Notifications & data structures** (keyspace events, sorted sets) → simple, reliable expiry orchestration.
- **In-memory cache** → big reduction in read load for booking/history views without touching the transactional path.

---------- Amazon ----------

Simple diagram incase if below is complex :
https://chatgpt.com/canvas/shared/68d15b38a4e48191b7d94f37332a01d0

Ask (clarify with interviewer)

- Traffic & scale: peak RPS for search, add-to-cart, checkout; DAU/MAU; SKUs; sellers.
- Latency SLOs: search (<200 ms p95), PDP (<150 ms p95), checkout (<300 ms p95).
- Consistency expectations: inventory & payments (strong), search & recos (eventual).
- Delivery surface: web, iOS/Android; regions; multi-currency/tax.
- Serviceability scope: what constraints (pincode, weight/size classes, courier rules)?
- Orders retention: online vs archival windows (e.g., 90 days hot → archive).
- Payment flows: supported gateways, success/failure/timeout SLAs, idempotency keys.
- Security/compliance: PCI boundaries, PII handling, authN/Z approach.
- Reporting & ML: freshness for "trending", window sizes (e.g., last 30 min/1 hr).

Process (thinking & key logic)

- Split read paths vs write paths:
  - **Read-heavy** (Home/Search/PDP): highly available, low latency, eventual consistency OK.
  - **Write-critical** (Inventory/Orders/Payments): strong consistency, transactional.
- Event-driven spine (Kafka): suppliers → item ingestion → search index, analytics, recos; orders/payments → inventory, search visibility, notifications, analytics.
- **Serviceability/TAT precompute**: build M×N (warehouses×pincodes) reachability & TAT and cache it; read-time is just lookup + filters.
- **Checkout saga**:
  1. Create order (CREATED) in MySQL → 2) Block inventory (no negative stock) → 3) Hand off to Payment → 4a) Success ⇒ mark PLACED; emit events
     4b) Failure ⇒ mark CANCELLED + inventory rollback

4b) Failure ⇒ mark CANCELLED + inventory rollback
4c) User abandons ⇒ Redis TTL expiry callback ⇒ cancel + restock.

- **Race handling**: on payment success/failure, proactively delete Redis hold to reduce expiry races; reconcile periodically.
- **Data life-cycle**: active orders stay in MySQL; terminal orders (DELIVERED/CANCELLED) are archived to Cassandra; "Orders View" reads live+historical.

Beyond what they asked (value-adds)

- **Search result gating** by serviceability & ETA at query time (great UX).
- **Signals pipeline** from search, wishlist, cart, orders → near-real-time "trending" and personalized recos.
- **Inventory-aware search visibility**: out-of-stock removal via consumer reacting to order/stock events.
- **Reconciliation & idempotency**: background checker to guarantee inventory/order invariants.
- **Cost/scale levers**: isolate Wishlist vs Cart infra; archive aggressively; cache user/profile hot keys.

Design (key building blocks & choices)

- **Item Service** on **MongoDB**: flexible product attributes (shirts vs TVs vs bread).
- **Search**: Search Consumer → **Elasticsearch** index; Search Service provides query/filter/fuzzy.
- **Serviceability & TAT**: precomputed graph cached (in-memory/Redis) from Warehouses + Logistics.
- **User Service**: MySQL with Redis read-through cache.
- **Cart & Wishlist**: separate services, each on **MySQL**.
- **Inventory Service**: transactional decrements with "no negative" constraint.
- **Orders (OMS)**: Order Taking + Order Processing on **MySQL**; Historical Orders on **Cassandra**.
- **Payments**: gateway abstraction; handle success/failure/timeout.
- **Notifications**: email/SMS/push via Notification Service.
- **Analytics/ML**: Spark streaming + Hadoop; recos stored and served via Recommendation Service (e.g., Cassandra).
- **Event bus**: **Kafka** for all domain events.

Table designs (compact sketches)

- **users**(id PK, name, email …); **addresses**(id PK, user_id FK, pincode, …)
- **items** in Mongo (sku_id, title, attrs{…}, seller_id, category, description)
- **search_index** in ES (sku_id, title, description, attrs, popularity, stock_flag)
- **inventory**(sku_id PK, warehouse_id PK, count, updated_at) with CHECK count >= 0
- **carts**(user_id PK, updated_at), **cart_items**(user_id, sku_id, qty, …, PK(user_id, sku_id))
- **wishlists**(user_id PK), **wishlist_items**(user_id, sku_id, …, PK(user_id, sku_id))
- **orders**(order_id PK, user_id, status, total, created_at, updated_at, address_id, …)
- **order_items**(order_id, sku_id, qty, price, …, PK(order_id, sku_id))
- **payments**(payment_id PK, order_id FK, status, txn_ref, amount, method, created_at)
- **serviceability_cache** (pincode, sku_class/size_bucket, warehouse_set, ETA, allowed, updated_at)
- **historical_orders** in Cassandra, modeled for query patterns:
  - by **order_id** (partition: order_id)
  - by **user_id** (partition: user_id, clustering: created_at desc)

Core APIs (representative)

- Item: GET /items/{id}, POST /items, PATCH /items/{id}, POST /items/bulkGet
- Search: POST /search{q, filters, userId?} → results gated by serviceability+ETA
- Serviceability: POST /check{pincode, skuId|attrs} → {deliverable, eta} (cached)
- Cart: GET/PUT/DELETE /users/{id}/cart, POST /cart/items
- Wishlist: GET/PUT/DELETE /users/{id}/wishlist
- Inventory: POST /inventory/block{orderId, skuId, qty}, POST /inventory/release{orderId}
- Orders: POST /orders (create+status=CREATED), GET /orders/{id}, GET /users/{id}/orders
- Payments: POST /payments/charge{orderId, …}, webhooks: /payments/callback
- Notifications: POST /notify{userId, channel, template, params}
- Archival: internal cron → GET /orders?status in [DELIVERED,CANCELLED]&before=… → POST /historical-orders

Important MySQL properties used & why MySQL here

- **ACID transactions** (InnoDB) to update orders, payments, inventory atomically.
- **Row-level locking** for safe concurrent inventory decrements.
- **Foreign keys & referential integrity** across orders/order_items/payments.
- **Unique/Check constraints** (e.g., prevent negative stock; idempotency keys on payments).
- **Indexes & replicas**: covering indexes for hot reads; replicas for read-heavy endpoints.
- **Isolation level**: REPEATABLE READ (default) or tune to READ COMMITTED to balance phantom reads vs contention.

- **Partitioning/sharding strategy** once rows grow (e.g., by order_id hash or time).
- **Mature ops**: backups, PITR, well-understood failover.
  Chosen over a NoSQL store for **write correctness, relational joins**, and **transactional guarantees** during the mutable life of an order. (Historical reads shift to Cassandra to scale cheaply.)

How Redis is used (all the places)
- **User cache**: read-through cache for User Service (profile, default address).
- **Order hold TTL**: ephemeral key {orderId: expiryTs} to guard payment timeout; expiry callback triggers cancel + restock.
- **Serviceability/TAT cache**: store precomputed reachability & ETA (can also be in process cache; Redis makes it shared & durable across pods).
- **General hot caches**: e.g., item summaries, config/feature flags, rate limits (optional).

Takeaways (interview checklist)
- Separate **mutable, transactional** data (MySQL) from **immutable/archival** (Cassandra) and **search** (ES).
- Guard **inventory correctness** with constraints + transactions; treat checkout as a **saga** with explicit compensations.
- Prefer **precomputation + caching** for serviceability; never compute routes at request time.
- Drive **UX** with infra: serviceability-aware search, instant ETA, inventory-aware listing.
- Emit **everything** to Kafka; build **near-real-time insights** and recos from behavior signals.
- Plan for races, retries, idempotency, and reconciliation from day one.

----------Navigation System Design (like Google Maps)----------

https://chatgpt.com/g/g-p-68c843f3a588819196b40ecc5aafd585/c/68d1668b-7b6c-8322-ad39-f8493abc3ef1

Ask (Questions to clarify with interviewer)
- What is the expected scale of the system? (e.g., MAUs, requests per second)
- What is the acceptable SLA for route calculation? (e.g., 2–3 sec vs. real-time)
- What level of accuracy is acceptable? Perfect optimal route or “good enough”?
- Do we need to support real-time traffic/weather/accident updates?
- Should we design for global coverage or a smaller region initially?
- How frequently can we collect user GPS pings? (battery vs accuracy trade-off)
- Do we need to handle disputed regions/borders? If yes, how?
- Should historical ETA predictions (based on day/hour) be supported?

Process (Thinking process & logic)
- Model the map as a **graph**: intersections = nodes, roads = directed weighted edges.
- Use **segments** (1km x 1km tiles) → manageable sub-graphs.
- Precompute within-segment shortest paths (Floyd-Warshall/Dijkstra) and cache.
- Identify **exit points** of each segment → connect segments into larger graphs.
- For long distances: introduce **mega-segments** (hierarchical abstraction).
- Weights on edges: distance, time, average speed.
  - Don’t hardcode traffic/weather as weights → treat them as **attributes that affect speed**.
- Use **dynamic programming + caching** to avoid recalculating routes every time.
- For deviations: continuously track user movement → recalculate route if deviation occurs.
- Bubble-up mechanism: traffic/ETA updates at road → segment → mega-segment.
- Use **historical ETAs** + real-time data for better predictions.

Beyond what they asked for (Extra functionality / out of box)
- **Automatic road discovery** from crowd-sourced GPS traces (detect new/unmapped roads).
- **Vehicle-type inference** (car, bike, bus, etc.) from driving patterns → tailor routes.
- **Hotspot detection** (large crowds) → infer possible traffic build-ups/events.
- **User profiling** (home/work inference, habits, interests from location history).
- **Third-party integration** (Waze, weather APIs) with trust scoring based on accuracy.
- **Disputed territories handling** → show different boundaries based on user’s country.

Design (Key systems & choices)
Data & Processing
- **Graph Storage** → Cassandra (high write throughput for location updates, scalable).
- **Real-time updates** → Kafka + Spark Streaming (traffic updates, road discovery, hotspot detection)

- **Real-time updates** → Kafka + Spark Streaming (traffic updates, road discovery, hotspot detection).
- **Historical data** → Cassandra (ETAs by day/hour).
- **Search** → ElasticSearch for fuzzy location search and address resolution.

Routing
- **Graph Processing Service** → Runs Dijkstra/A* for route calculation.
- **Caching** → Store precomputed shortest paths within segments + between exits.
- **Dynamic adjustments** → Bubble-up ETA updates from road → segment → mega-segment.

Client & Edge
- **WebSocket connections** → persistent for location pings (frequency varies by movement).
- **Navigation Tracking Service** → deviation detection + trip analytics.

Extra Infra
- **Map Update Service** → adds new roads from streaming jobs.
- **Traffic Update Service** → updates edge weights based on real-time info.
- **Third-party Data Manager** → aggregates traffic/weather feeds.
- **Analytics** → evaluate ETA accuracy, route adoption, popular POIs.

Takeaways (Key lessons to remember)
- **Segmentation hierarchy** (segments → mega-segments) makes massive-scale routing feasible.
- **Do not model traffic/weather as separate graph weights**; instead adjust average speed dynamically.
- **Cache aggressively** (within segments & exits) to avoid repeated Dijkstra runs.
- **Bubble-up updates** ensures changes propagate efficiently without recalculating whole graph.
- **Crowdsourced data (organic GPS traces)** is critical for accuracy and discovering new roads.
- **Disputed areas** handling is a real-world edge case often overlooked in interviews.
- **Design must balance accuracy vs performance**: "good enough in 2–3 sec" is better than perfect but slow.

---------- chat app (whats app, fb messanger) ----------

Ask (clarify with the interviewer)
- Users & scale: DAU/MAU? peak concurrent connections? peak messages/sec? max group size?
- Delivery model: store-and-forward like Messenger vs. delete-after-delivery like WhatsApp?
- Consistency: do we require ordered delivery per-conversation? exactly-once vs at-least-once?
- Latency SLOs: P95 for send→deliver, deliver→read, media fetch?
- Features in scope: E2EE, push notifications, typing/online indicators, message edits/deletes, pinned messages?
- Attachment limits: max size/duration, formats, thumbnails, transcoding?
- Retention/Compliance: data TTLs, legal holds, GDPR/DSR, audit trails?
- Reliability targets: availability per region, disaster recovery RPO/RTO?
- Abuse & safety: spam limits, user reports, content moderation?
- Analytics/privacy: what's allowed if E2EE is on?

Process (how to think / tricky parts)
- Real-time fanout path:
  1. Device → WebSocket Handler (WSH) (bi-directional TCP)
  2. WSH → Message Service (persist)
  3. WSH → WebSocket Manager (WSM) to locate recipient's current handler
  4. WSH(src) → WSH(dst) → recipient device
  5. Acks (sent/delivered/read) flow back, update state, notify sender
- Why WebSocket **Manager** (vs only handlers): central, fast, consistent mapping of **user → handler(s)** and **handler → users** so any handler can route to any user after reconnects, cross-AZ, multi-region mobility, and to prune stale mappings quickly. Cached on handlers with tiny TTL to avoid thundering herds while staying fresh.
- Offline delivery: if recipient offline, persist as "sent"; on reconnect, WSH(dst) pulls "pending" messages in bulk (by user) and backfills acks.
- Race handling: do writes first (persist), then route; add periodic "missed messages" pull (by conversation watermark) to close gaps.
- Ordering & idempotency: per-conversation **(sender, conversation_id)** monotonic client_seq + server timeuuid; dedupe on (conversation_id, msg_id); deliver in order within a single conversation.
- Groups: producer writes one message; **Group Fanout Worker** consumes from Kafka, resolves members via Group Service, batches by destination handler, and routes.
- Media: client compresses → Asset Service → S3 (hot) + CDN; message carries asset_id; optional pre-upload **hash check** (multi-hash) for de-dup.
- Presence/last-seen: app/user events into **Last-Seen Service**; write-heavy → wide-column store (Cassandra) over MySQL/Redis.
- Backpressure: per-device send queue, rate limits, drop oldest typing indicators before messages, circuit-break

media uploads on congestion.

Beyond what they asked for (useful extras)

- E2EE envelope w/ double-ratchet (keystore service only stores encrypted blobs & device public keys).
- Multi-device sync: per-user device fanout with per-device read cursors.
- Edits/deletes: "tombstone" events versioning messages; reconcile on recipients.
- Push fallback: if WS down/backgrounded, enqueue FCM/APNs push with summary.
- QoS tiers: small control channel (acks, typing) vs data channel (messages) with priority queues.
- Spam/abuse: per-sender token bucket, anomaly detection on Kafka stream, ML classifiers.
- Observability: RED + USE metrics; Kafka lag alerts; handler imbalance detector (rebalance users across WSH shards).
- Cost controls: cold-storage old media to Glacier; tiered retention per conversation type.

Design (key components, tech picks & alternatives)

- **WebSocket Handlers (WSH)**
  - Stateless compute pods keeping WS connections; local caches: (users-on-this-handler), and short-TTL (user→handler).
  - Alt: Envoy TCP proxy + stateful WS workers; or MQTT brokers if you accept protocol change.
- **WebSocket Manager (WSM) + Redis Cluster**
  - Canonical maps: user→handler_set, handler→user_set; pub/sub for disconnects; TTL on entries.
  - Alt: Aerospike / Scylla for larger mappings; or etcd if you want strong consistency (slower).
- **Message Service + Cassandra/Scylla**
  - Write-optimized, time-series access, per-conversation partitioning; tombstones acceptable.
  - Alt: DynamoDB (LSI/GSI) or FoundationDB (for strict transactions).
- **Group Service + MySQL(+Read Replicas) + Redis**
  - Many-to-many membership; APIs: list members, roles, join/leave.
  - Alt: Postgres w/ partitioning; or graph DB (only if complex queries).
- **Kafka (or Pulsar)**
  - Topics: group-messages, delivery-events, analytics-events.
  - Alt: Pulsar for geo-replication out-of-the-box; Kinesis (managed).
- **Group Fanout Worker**
  - Kafka consumer that batches by destination handler; idempotent on (msg_id, user_id).
- **Asset Service + S3 + CDN**
  - Upload, virus scan, thumbnailing, signed URLs; dedup by multi-hash index.
  - Alt: GCS/CloudFront equivalents; image proxy like imgproxy/Thumbor.
- **Last-Seen Service + Cassandra**
  - Schema: (user_id) → last_seen_ts, status; high write throughput.
  - Alt: Redis for small scale; timeseries DB if you keep history.
- **User Service (profiles), AuthN/Z**
  - MySQL (clustered) + Redis cache; OIDC; device tokens; key directory if E2EE.
- **Analytics/Reporting**
  - Kafka → Spark/Flink → Object Store/Hive; privacy gate if E2EE.
- **Infra & ops**
  - Global Anycast/Geo-DNS → regional LB → WSH pools; autoscale on concurrent WS; shard users by hash(region, user_id).

Lean schemas (good enough for interviews)

- **Cassandra (messages_by_conv)**
  - conversation_id (PK part), timeuuid msg_ts (clustering asc), msg_id, from_user, payload, type, status_per_user (map<user, status>), deleted, edit_version, asset_id
  - One table per access pattern if needed: pending_by_user(user_id, msg_ts, conversation_id, msg_id, …)
- **Cassandra (last_seen)**
  - user_id (PK), last_seen_ts, presence_state (online/away/offline)
- **MySQL (groups)**
  - groups(id, name, created_by, created_at)
  - group_members(group_id, user_id, role, joined_at, status) (PK (group_id,user_id))
- **Redis keys**
  - u2h:{user_id} -> set(handler_ids) TTL few minutes
  - h2u:{handler_id} -> set(user_ids)

Minimal APIs (surface area the interviewer expects)

- WebSocket events: send_message, delivery_ack, read_ack, typing, presence_update
- REST/gRPC:
  - POST /messages (idempotent with Idempotency-Key)
  - GET /conversations/{id}/messages?after=cursor&limit=N

    - GET /users/{id}/pending
    - POST /assets → {asset_id, urls}
    - GET /groups/{id}/members / POST /groups/{id}/messages

Trade-offs & choices (quick justifications)

- **Cassandra over MySQL** for messages/last-seen due to write throughput, time-series partitions, and linear scale; MySQL stays for relational bits (users, groups).
- **Redis** as hot path locator cache; tiny TTL prevents stale routes during mobile flaps.
- **At-least-once delivery** + idempotency & ordering per conversation is simpler and robust; exactly-once is expensive at this scale.
- **Handlers stay lightweight** (no group fanout/business logic) to maximize WS throughput and minimize GC pauses.

Takeaways (what to keep in mind)

- Separate **connection plane** (WSH/WSM) from **data plane** (Message/Group/Asset).
- Persist **before** route; add **periodic pull** to seal races.
- Use **short-TTL caches** for user→handler to balance freshness vs. WSM QPS.
- Prefer **batch fanout** and **per-handler batching** for groups.
- Design for **idempotency, ordering, and backpressure** from day one.
- Measure and autoscale on **concurrent sockets**, **msg ingress rate**, and **Kafka lag**.

If you want, I can turn this into a clean architecture diagram next (with grouped color buckets and arrows) or add a deeper table/API spec for interviews.

---------- Youtube ----------

Functional Requirements (FRs)

- **Upload video** – by individuals or production houses.
- **Homepage & Recommendations** – show popular/recommended videos.
- **Search & Play** – users should be able to search and play videos.
- **Multi-device support** – web, mobile, smart TVs, wearables, etc.
- **Adaptive Bitrate Streaming** – client adjusts video quality dynamically based on bandwidth.

Non-Functional Requirements (NFRs)

- **Low Latency + High Availability** – minimal buffering.
- **Increase User Session Time** – better UX, better recommendations.
- **Scalability** – horizontally scalable components.
- **Support for multiple formats, resolutions, bitrates** – handle diverse devices and networks.

Key Design Components

1. Content Upload & Processing
    - **Asset Onboarding Service** – ingestion of content (via UI, SFTP, or direct S3 upload).
    - **Amazon S3** – stores raw and processed video files.
    - **Cassandra** – metadata store (video info, tags, thumbnails, chunk status).
    - **Workflow via Kafka Events** – decoupled processing.
    - **Content Processor Workflow**:
        - **File Chunker** → splits video into segments.
        - **Content Filter** → checks for piracy, nudity, illegal content.
        - **Content Tagger** → auto-tags, generates thumbnails.
        - **Transcoder** → converts to different formats (AVI, MP4, MKV, etc.).
        - **Quality Converter** → creates multiple resolutions/bitrates (1080p, 720p, 480p).
        - **CDN Upload** → final chunks pushed to CDNs.
2. User Management
    - **User Service (MySQL + Redis)** – authentication, subscriptions, profile, caching.
    - **Device Fingerprinting** – detect credential sharing.
    - **Login Flow** – works across all device types.
3. Video Delivery
    - **Host Identity Service** – maps user location to nearest CDN.
    - **Main CDN** – stores all videos globally.
    - **Local CDN** – region-optimized caching for popular content.
    - **Adaptive Bitrate Streaming on Client** – adjusts resolution (1080p ↔ 480p).
4. Search & Homepage
    - **Search Consumer → Elasticsearch** – fuzzy search, typo-tolerance, autocomplete.
    - **Search Service** – queries ES, applies user-based filters.
    - **Homepage Service** – personalized feed.
5. Analytics & Recommendation

5. Analytics & Recommendation
- **Stream Stats Logger** – tracks how long videos are watched (proxy for rating).
- **Analytics Service** – collects homepage & search click behavior.
- **User Profiling** – builds genre preferences.
- **Recommendation Engine**:
  - Content-based (tags, genres).
  - Collaborative filtering (ALS model).
- **Traffic Predictor** – predicts future demand, pre-loads local CDNs.

6. Data Infrastructure
- **Kafka** – event backbone (uploads, search, play, analytics).
- **Spark Streaming** – aggregation (tags, thumbnails, user behavior).
- **Hadoop** – offline analytics, ML training, thumbnail testing.
- **Cassandra** – scalable metadata storage.
- **Elasticsearch** – search functionality.

Optimizations
- **Torrent-style CDN replication** – local CDNs share chunks to reduce S3 load.
- **Netflix Open Connect Model** – deploy appliances at ISP level for caching.
- **Thumbnails A/B Testing** – choose thumbnails based on CTR, possibly ML-driven per-user.
- **Proxy ratings from watch-time** – instead of explicit ratings.

Trade-offs & Technology Choices
- **Cassandra** → high write/read throughput, no single master, suited for metadata scale.
  - Good: predictable queries (by video_id, chunk_id).
  - Bad: not for fuzzy search/aggregation → handled by Elasticsearch & Hadoop.
- **Redis** → cache for user/session data (low-latency lookups).
- **Elasticsearch** → fuzzy search, autocomplete, typo handling.
- **Kafka** → decoupling + event-driven processing.
- **CDNs** → reduce latency, improve availability.

Takeaways
- **Separation of concerns** – Upload/Processing, Delivery, Search, Analytics, Recommendation.
- **Event-driven + parallel workflows** – chunking & parallel processing critical for scale.
- **Client-side intelligence (Adaptive Bitrate)** – shifts load from server to device.
- **Hybrid infra (real-time + batch ML)** – Spark + Hadoop + Kafka + Cassandra.
- **Scalability & fault-tolerance** built into every layer (Cassandra rings, Kafka partitions, CDN hierarchy).

---------- Uber / Ola ----------
Ask (clarify with the interviewer)
- Traffic & scale: monthly/DAU, peak concurrent users/drivers, expected QPS for pings/booking.
- Geo scope: single city, country, or global (multi-region)?
- Latency targets: “see nearby cabs” (p50/p95), price/ETA, booking confirmation.
- Consistency vs availability: which flows must be strongly consistent (e.g., single assignment) vs highly available (e.g., map tiles, user/profile reads)?
- Location ping cadence: driver ping interval (e.g., 5–10s), payload size.
- Segmentation granularity: segment size bounds, dynamic split/merge thresholds.
- Matching mode: “best driver” deterministic vs “broadcast, first-accept”.
- Trip data retention & audit: how long to retain full paths, legal/audit needs.
- Payments: instant vs batch; supported gateways.
- Cost constraints & managed services allowed (Redis/Cassandra/MySQL/Kafka/Spark etc.).
- Fraud, surge & ML: which are in scope for v1 vs later?

Process (core thinking & key logic)
- Spatial index via **city segments (grid)**: map each (lat,lng) to a segment id; dynamically **split/merge** segments based on driver density.
- **Nearby search**: get rider’s segment S, then query **S + adjacent segments**; compute **road distance/ETA** (not aerial) to shortlist N drivers.
- **Matching modes**:
  - *Best-driver*: rank via Driver Priority Engine → assign top 1.
  - *Broadcast*: notify N candidates → first accept wins.
- **Single-assignment correctness**: use a small, strongly consistent section (e.g., DB row/lease/compare-and-set) to avoid double-assign.
- **Hot path vs cold path**:

- Hot: WebSockets, nearby drivers, booking, acceptance — low latency.
- Cold: archiving, analytics, ML, heatmaps — via Kafka → Spark/Hadoop.

- **Storage split**:
  - MySQL for live, transactional trip state.
  - Cassandra for append-heavy, time-series (location traces) & archived trips.
  - Redis for fast lookups: user/driver profiles, segment→driver sets, WebSocket routing.

Design (services & responsibilities)

- **User Service**: user CRUD, profile, trips proxy. *(MySQL + Redis cache)*
- **Driver Service**: driver CRUD, docs, ratings, payout history. *(MySQL + Redis)*
- **Map Service**: segment grid mgmt (split/merge), point→segment, neighbors(S), route, ETA, distance. (May proxy Google/own maps.)
- **Location Service**: ingest driver pings (WebSocket), compute segment, update:
  - Redis: segment:{id} -> set(driver_ids)
  - Cassandra: driver_location_timeseries, driver_last_location
- **Cab Request Service**: WebSocket with rider; UI streaming of nearby cabs; orchestrates booking UI updates.
- **Cab Finder**: core matcher — asks Location/Map for nearby drivers, consults **Driver Priority Engine**, triggers assignment, emits Kafka events.
- **Driver Priority Engine**: ranks candidates by score (ETA, rating, cancellations, reliability, recent acceptance, fairness, etc.).
- **WebSocket Handler(s)**: keep persistent connections with drivers (and riders).
- **WebSocket Manager**: directory in **Redis**:
  - driver_to_host:D123 -> H7 and host_to_drivers:H7 -> {D...}; handles joins/leaves.
- **Trip Service**: source of truth for trip lifecycle (CREATED→ASSIGNED→EN_ROUTE→PICKED_UP→COMPLETED/CANCELLED).
  - *Live rows in MySQL*; *archive to Cassandra* via **Trip Archiver** (cron/batch).
- **Payment Service**: consumes Kafka "trip_completed"; records payables (MySQL); triggers gateway (instant) or batches (cron).
- **Analytics/ML**: Kafka → Spark Streaming (heatmaps, no-driver-found), Hadoop data lake for profiling, fraud, ETA improvements, surge signals.

Key Data (sketch)

MySQL (OLTP)

- users(id, name, rating, ...), drivers(id, docs, status, rating, ...)
- trips(id, rider_id, driver_id, status, src_lat, src_lng, dst_lat, dst_lng, est_eta, est_price, start_ts, end_ts, ...) (normalized with auxiliary tables if needed)
- payments(id, driver_id, trip_id, amount, status, created_ts, settled_ts, ...)

Cassandra (TS / heavy write)

- driver_last_location(driver_id) -> {lat,lng,ts,segment}
- driver_location_timeseries(driver_id, ts_bucket) -> list<point>
- archived_trips(trip_id) -> wide row of trip facts/events

Redis (fast lookups)

- segment:{segment_id} -> SET(driver_ids)
- driver_to_host:{driver_id} -> host_id
- host_to_drivers:{host_id} -> SET(driver_ids)
- hot user/driver profile caches; idempotency tokens; short-lived locks/leases.

Core Flows (high level)

Driver location update

1. Driver → WebSocket Handler → Location Service (every 5–10s).
2. Location → Map: resolve segment; if segment changed → update Redis segment set.
3. Persist driver_last_location + append to driver_location_timeseries (Cassandra).
4. Emit Kafka location_updated (for heatmaps/traffic/ETA tuning).

Rider requests cab

1. Rider → Cab Request Service (WebSocket): {src,dst}.
2. Cab Request → Cab Finder.
3. Cab Finder → Location: segment of rider + neighbor segments; shortlist N drivers with **Map ETA**.
4. Cab Finder → Driver Priority Engine: rank; pick mode:
   - Best-driver: assign top; or
   - Broadcast: push to N via WebSocket Handler(s); first accept wins.
5. Single-assignment: compare-and-set on trips row / Redis lock to avoid double assignment.
6. Notify rider (Cab Request WS) + notify chosen driver (WebSocket Handler).
7. Trip Service persists status; emit Kafka trip_assigned.

Trip lifecycle & payment

- Status transitions in MySQL; upon COMPLETED, Kafka event → Payment Service → payable in MySQL → gateway/batch settle.

APIs (sample, concise)

- POST /cab-request (WS init or HTTP): src,dst → stream nearby, then assignment.
- GET /trips/{id}; GET /users/{id}/trips; GET /drivers/{id}/trips
- POST /drivers/{id}/location (WS under the hood)
- GET /price-estimate?src=&dst=; GET /eta?src=&dst=
- Internal: Map: /segment?lat=&lng=, /neighbors?segment=, /eta?src=&dst=
- Internal: Location: /nearby?segment=&k=

Availability vs Consistency (where each matters)

- **Highly available (read-heavy):** user/driver profiles (via Redis), "show nearby cabs", price estimate, map tiles/ETA (stale-tolerant).
- **Strong consistency (write-critical):** single driver assignment to a trip; payment ledger writes; driver/rider state transitions for the *active* trip.
- **Eventual consistency:** segment membership sets, heatmaps, analytics, archived trips.

Why WebSocket Handler & Manager?

- **Handler** keeps long-lived bi-directional connections (lower overhead vs repeated HTTP; server→device pushes for offers/updates).
- **Manager** is the directory so any backend can find "which Handler hosts driver D?" instantly (O(1) Redis lookups), plus reverse mapping for fan-outs/broadcasts.

Segmenting & Nearby Search (nutshell)

- Grid segments (e.g., geo hash or fixed grid).
- Map (lat,lng) → segment_id by bounds check; keep **neighbor map** per segment.
  For rider's segment S: query S ∪ neighbors(S) driver sets from Redis; compute road ETA (Map Service) to pick top-K.

Kafka Streams & ML (beyond basics)

- **Heatmaps & surge**: stream no_driver_found + demand/supply ratios by segment; driver app overlay.
- **Fraud signals**: high rider–driver correlation, spoofed pings, abnormal acceptance/cancel patterns.
- **Driver scoring**: acceptance rate, on-time arrival, rating trend, ETA accuracy; feed Driver Priority Engine.
  ETA improvement: leverage fleet telemetry → traffic speeds per road segment.

Failure Handling & Resilience

- Driver reconnects to a different Handler → Manager updates mappings (idempotent).
- If chosen driver times out/rejects → next in ranking or re-broadcast.
- Redis persistence & AOF/RDB for Manager directory; multi-AZ, replicas.
- MySQL: primary-replica, semi-sync, failover; narrow transactions around assignment.
- Cassandra: multi-DC replication; tune CL (e.g., LOCAL_QUORUM) for last-location reads/writes.
- Backpressure: throttle pings; drop/merge duplicate location updates per driver/time window.

Takeaways

- Separate **hot, low-latency paths** (WebSockets, Redis lookups, minimal DB writes) from **cold/analytic** paths (Kafka→Spark/Hadoop).
- Use **segmentation + neighbor search** to bound fan-out and keep matching fast.
- Blend **availability** for read/UX paths with **consistency** for single-assignment, payment, and active trip state.
- Choose **MySQL for transactional live trips**, **Cassandra for write-heavy timeseries & archives**, **Redis for real-time indexes/caches**.
- Keep WebSocket routing explicit (Manager) to scale to fleets across regions.

If you want, I can also add:

- a compact ERD + a sequence diagram for "request→assign" flow,
- or a table listing each service, datastore, and its SLA (p50/p95),
  just say the word and I'll drop it in the same style as your other docs.

---------- Twitter ----------

1) Requirements (quick scan)

Functional

- Post Tweet (≤140 chars, links, media via Asset Service)
- Retweet
- Follow/Unfollow (directed graph)
- Read Home Timeline & User Timeline
- Search tweets
- Notifications / live updates

Non-Functional

- Read-heavy (~100× reads vs writes — assumption for reasoning)

- p95 < 1s for Home Timeline render; fast tweet ACK
- High availability; eventual consistency acceptable (tweet visibility may lag seconds)
- Horizontal scalability; cache-first thinking

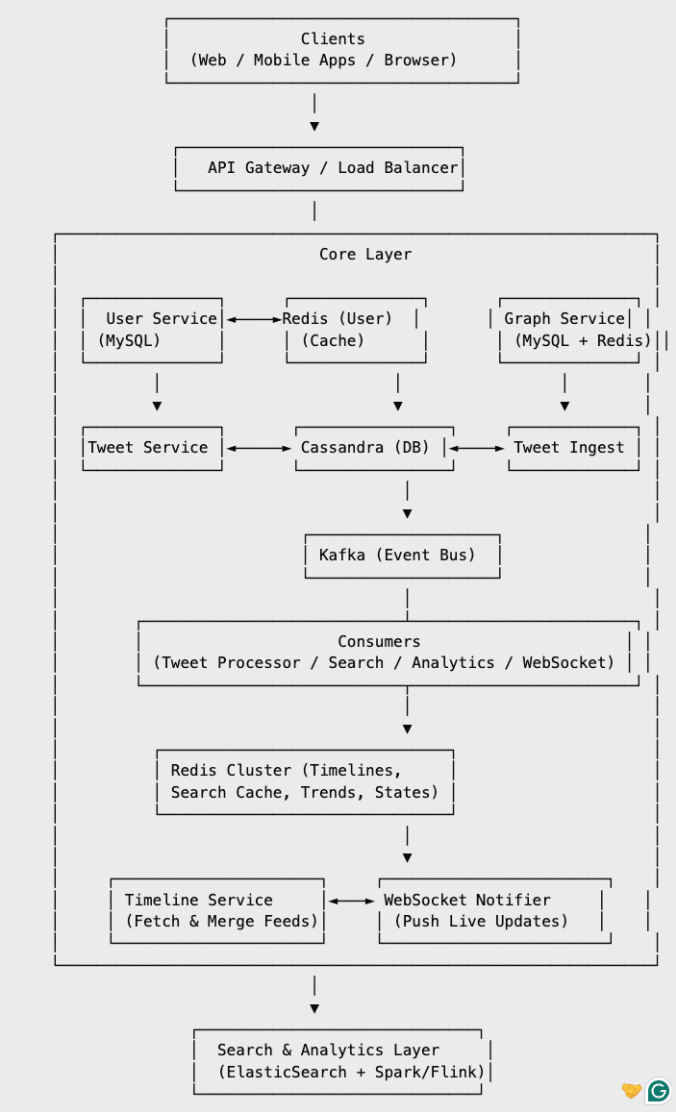
**Scale (assumptive)
**- ~150M DAU, ~350M MAU, ~1.5B accounts

- ~500M tweets/day (~5.7k/s avg; ~12k/s peak)

User Segments

- Famous (huge fan-out), Active (recently used), Live (online now), Passive, Inactive

2) Architecture (at a glance)



3) Core Components (what & why)

User Service (MySQL + Redis)

- Source of truth for profile; cache by userId for reads

Graph/Follows Service (MySQL + Redis)

- Table (userId, followerId, createdAt); caches:
  - following:{uid}; followers:{uid}

Tweet Ingestion → Cassandra

- Append-only writes; emits {tweetId, userId, ts, ...} to Kafka

Tweet Service (read APIs)

- Get by id; list by user (partition (userId, timeuuid))

Timeline Service (Redis + on-demand build)

- Active users → serve precomputed timeline:{uid}
- Passive users → build on miss (Graph → followees; Tweet Service → recent tweets; merge+cache)
- Famous users → merged at read time; keep famousMergedAt:{uid} marker (short refresh window)

Tweet Processor (Kafka consumer)

- **Fan-out on write** for normal authors (push tweetId into followers' Redis lists)
- **Skip mass fan-out** for famous authors (hybrid model)

- **Skip mass fan-out** for famous authors (hybrid model)
- Live followers → publish WebSocket hint for instant UI

WebSocket Notifier

- Tracks live sessions; pushes "new tweets" events; writes activity state to cache

Search

- Consumer → Elasticsearch (Lucene/BM25)
- Service reads ES; cache hot queries in Redis (TTL 2–3 min)

Trends / Analytics

- Stream word counts per region (30–60m window) → Redis trends:{region}
- Data lake for offline BI/ML, newsletter jobs, etc.

4) Data Model (essentials)

Cassandra: tweets

- PK: (userId, timeuuid); cols: tweetId, text, mediaRefs, retweetOf, lang, engagement...

MySQL

- users(userId PK, handle, name, email, prefs, status, ...)
- follows(userId, followerId, createdAt) composite PK + dual indexes

Redis Keys (sketch)

- timeline:{userId} → list of {tweetId, ts, authorId} (bounded N)
- timeline:famousMergedAt:{userId} → timestamp
- user:{userId} → profile blob
- followers:{userId} / following:{userId} → sets
- search:{q}:{page} → results JSON (TTL)
- trends:{region} → top-N terms (TTL)
- user:state:{userId} → {state: live|active|passive, lastSeen}

5) Read vs Write Path (step-by-step)

Write: Post Tweet

1. UI → Tweet Ingestion (validate, URL-shorten, attach media refs)
2. Persist to Cassandra
3. Publish to Kafka(tweets)
4. Tweet Processor:
    - Normal author → push to each follower's timeline:{uid} (left-push; trim to N)
    - Famous author → skip mass fan-out; optionally notify famous-followers only
    - Live followers → WebSocket hint

Read: Home Timeline

1. Try timeline:{viewer} in Redis
2. If hit → return; else (passive/miss) build: Graph→followees; Tweet Service→recent per followee; merge/sort; cache
3. Merge famous-authors' tweets; update famousMergedAt (short TTL refresh)

User Timeline

- Read from Cassandra partition (userId, timeuuid); optional read-through cache

6) Ranking (starter)

- Reverse-chronological baseline
- Optional boosts: author affinity, engagement, freshness decay

7) Trade-offs (interview talking points)

Fan-out on write vs read

- Write-fan-out → blazing reads; bad for famous authors
- Read-fan-out → uniform writes; heavier per-read compute
- **Hybrid** recommended (normal = write; famous = read)

Freshness vs Cost

- Shorter TTLs → fresher but pricier; segment by Active/Live vs Passive

Cassandra vs HBase

- Cassandra: simpler ops; HBase: tight Hadoop integration & scans

Search freshness

- Accept minutes of lag; cache hot queries to shield ES

8) Bottlenecks & Mitigations

- **Redis RAM**: bound list length; compress; shard via Redis Cluster; eviction policies; separate clusters by domain
- **Cassandra**: good partitioning; tune compaction; CLs (write QUORUM, read LOCAL_QUORUM); multi-DC

- **Kafka**: enough partitions; idempotent producers; consumer groups; backpressure
- **Elasticsearch**: shard planning; read replicas; query warming; request cache + Redis frontend
- **WebSocket**: shard connections; sticky sessions; fan-in via Kafka

9) Ops / Hygiene
- TTLs on timelines/search/trends; background expiry
- GDPR deletes & backfills; data lake retention
- Cache warmers for top Active users; rate limits & abuse detection streams

10) Service → Datastore Map

Service	**Primary Store**	**Cache(s)**
User Service	MySQL (users)	Redis user:*
Graph Service	MySQL (follows)	Redis followers:*, following:*
Tweet Ingestion	Cassandra (tweets)	—
Tweet Service	Cassandra	(optional)
Timeline Service	Redis (timelines)	—
Tweet Processor	—	Redis (write); Kafka in/out
Search Consumer	Elasticsearch	—
Search Service	Elasticsearch	Redis search:*
Trend Service	Redis (trends)	—
WebSocket Notifier	—	—

11) Interview Prompts (use live)
- What % users are "famous"? Threshold? (e.g., >1M followers)
- How fresh must Home Timeline be for Active vs Passive?
- Regional multi-DC requirements? Active-active or active-passive?
- Retention & legal deletes (right to be forgotten)?
- Max home feed length N? (e.g., 500–1000)

12) Personal Notes (edit during interview)
Assumptions I stated
Numbers I computed
Open questions from interviewer
Follow-ups / extensions
- Ranking service, spam/abuse pipeline, media pipeline, blocks/mutes handling, privacy scopes

NOTES
- **Fuzzy logic** deals with *gradual truth values* in reasoning and decision-making. (Elastic search using Levenshtein edit distance or Apache Solr, etc.,)
- **Semantic search** deals with *understanding meaning and context* in language.
- Redis cluster - It's a **distributed, in-memory key-value database** that acts as a **cache, data store, and message broker** at scale.
- WebSocket Handler - Maintains the **actual bidirectional socket** with each phones or systems, its responsible for Authenticate & upgrade HTTP → WS, Receive **location pings**; push **reroute/alert** messages instantly, Do lightweight validation + enqueue to Kafka, Apply backpressure, batching, and heartbeats
- Have to check on CPU, GPU useage on individual components
- All with have 5 layer :
  - Application - client & edge
  - Experience Api's
  - Core services
  - Data store like DB
  - Streaming for analytics and recommendation
- low latency and high availability : CDNs with caching + multi-region deployment + replication (Cassandra, Kafka, Redis) + stateless scalable services and the **trade-off** is **higher infrastructure cost and complexity in consistency, caching, and operations**.

MISTAKES:

- **Clarify Scope** → HLD or LLD? Don't assume. HLD : Components, modules, data flow, and interactions. LLD : Classes, methods, database schema, design patterns.
- **Confirm Functional Req.** → Which features exactly?
- **Check Non-Functional Req.** → Scale, users, throughput, data size.
- **Manage Time** → 40–45 mins → cover breadth first, then depth (1–2 components).
- **Avoid Assumptions** → Ask interviewer what to deep dive into.
- **Be Honest** → Admit gaps, explain how you'd approach.
- **Collaborate** → Ask feedback: "Am I on the right track?"