Structuring a Scalable Systems Design QA

Example: Design Netflix

Just a mental flow guideline.
Interviews require spontaneity

Step 1:

- Collect functional requirements
  - This is the detailed problem statement
  - High level
  - Spend a few minutes to show that you can communicate given an unknown problem
  - Ask questions to clarify all doubts as much as possible.

- Collect design constraints
  - Numbers, how many, how much
  - Required for answering scalability
  - Often interviewers throw these back to the candidate, so it is beneficial to research on them
  - Can be collected at a later step

Step 1 for Netflix: Example Functional Requirements

- User registration + login + profiles
- Search
- Recommendations
  - User activity tracking
  - Recommendation generation
  - Dashboard
- Payment
- Content Ingestion and Delivery
  - Rights and licenses
  - Categorization
- Notifications
- Trending Movies
- Watch History
- Endorsements

Step 1 for Netflix: Example Design Constraints

- Number of users: 100 million
- Daily active users: 10%
  - Usage pattern per region
- Volume of content ingested / sec:
- Volume of content delivered /sec:
- Content size
- Rate of user activity tracking

Step 2:

- Bucketize functional requirements into Microservices
  - Simple high level clustering of requirements
  - Imagine if all requirements can be handled by the same team or not
  - One approach can be: if data models and APIs to address two requirements do not look same, then put them in different buckets
  - Not deterministic, depends on every individual

- From this, it gets clear whether problem is breadth-oriented or depth-oriented

Step 2 for Netflix: Example Microservices

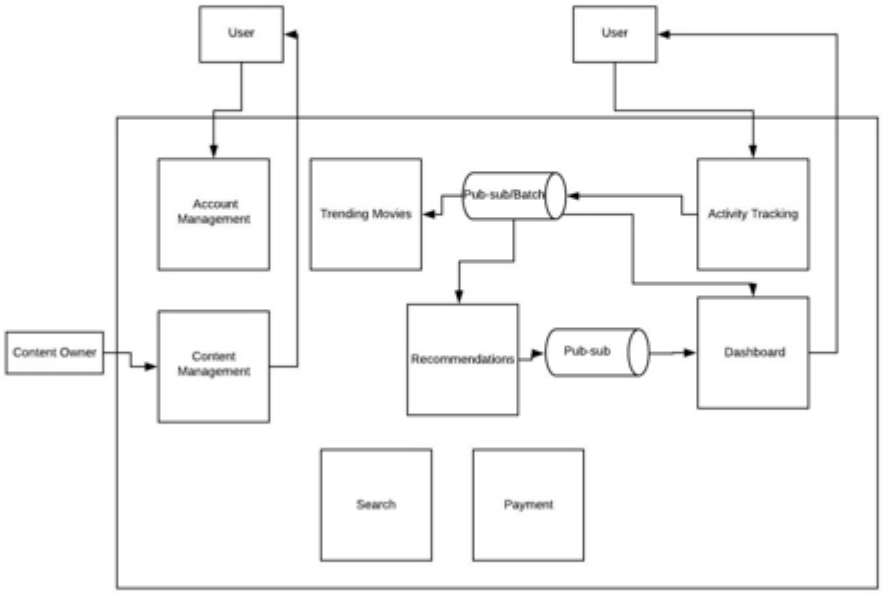
- User registration + login + profiles
- Search
- Recommendations
- Payment
- Content Ingestion and Delivery
  - Rights and licenses
- Notifications
- Trending Movies
- Watch History

Clearly, a breadth oriented problem

Step 3:

- Draw logical architecture
  - Block diagram of each Microservice
  - Use an API gateway (optional)
- Draw and explain data/logic flow between them
  - Rules of thumb:
    - If high volume of data (where each event is small) needs to be pushed in near real time between two microservices, use publisher subscriber
    - Pub-sub is a microservice of its own
    - If data needs to be pulled from server from client, use REST APIs
    - If data transfer is offline, you may use batched ETL (extract transform load) jobs

Step 3 for Netflix: Logical Block Diagram



Step 4:

- Now deep dive on each microservice at a time
- If too many microservices,
  - Negotiate with the interviewer on which ones to focus on
  - You only have constant time
  - Depending on the number of microservices to focus on
    - Budget your time per microservice
- Each microservice consists of one or more tiers:
    - App server tier - to handle application logic
    - Cache server tier - for high throughput data access
    - Storage server tier - for data persistence

Microservices to focus for Netflix

- User registration + login + profiles - No
- Search - No
- Recommendations - Yes (Lets defer to homework) Let’s talk about it
  - User activity history service
  - Recommendation Generator
  - Recommendation dashboard
- Payment - No
- Content Ingestion and Delivery - Yes
  - Rights and licenses
- Notifications - No
- Trending Movies - Yes (Lets defer to homework)
- Watch History - Yes

Step 4a.

- For each microservice
  - Solve each tier logically (as if solving for a single server)
  - Scale is not in the picture yet
  - Identify Data Model (what data needs to be stored) to match the functional requirements
  - Discuss how data will be stored in storage and cache tiers
  - Propose APIs to match the functional requirements
  - Propose workflow/algo for the the APIs in each tier
  - Propose flow across tiers within the microservice
    - This is Phase 1 per microservice
    - **Most of technical design interviews have meat in this phase**
    - **Non-deterministic, every candidate will come with his or her own proposal**
    - **Changes from problem to problem**
    - **This constitutes the 'most thinking' portion of the interview**
    - **Most chances to flunk here unless candidate is prepared**

Step 4b.

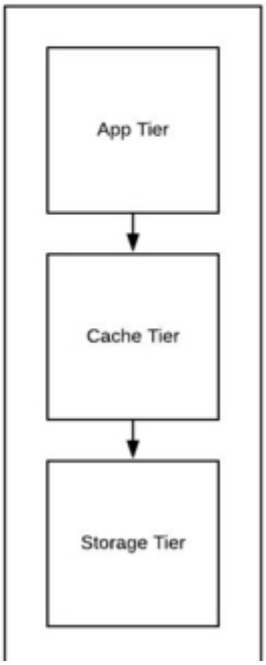
- For each microservice
  - Check whether each tier needs to scale
  - This is Phase 2 per microservice
  - A deterministic set of reasons can be posed across all interviews
    - Need to scale for storage (storage and cache tiers)
    - Need to scale for throughput (CPU/IO)
    - Need to scale for API parallelization
    - Need to remove hotspots
    - Availability and Geo-distribution
  - The constraints or numbers change from problem to problem
  - Solve algebraically first and then put numbers
    - Algebraic solution is same for all problems
  - If phase 1 takes more time, do not spend time on calculations/estimations here unless asked for and get out fast

Step 4c.

- For each microservice
  - If a tier needs scale, scale the tier and propose the distributed system
- This is Phase 3 per microservice
- This is the most deterministic portion of the solution
- Steps
  - Draw a generic distributed architecture per tier
  - If app server tier and stateless, just round robin requests
  - If cache or storage tier
    - Partition the data into shards or buckets to suit requirements of scale (changes from problem to problem)
    - Propose algorithm to place shards in servers (consistent hashing) (same across all problems)
    - Explain how APIs work in this sharded setting (problem specific)
    - Propose replication (same across all problems)
    - Propose CP or AP (algorithms for CAP theorem do not change from problem to problem)
- This is how each microservice looks within a single data center, either on-premise or VPC in cloud

Step 4a for Content Ingestion and Delivery Microservice

- App tier, Cache for content metadata, Storage for both metadata and data
- Data Structures
  - Metadata
  - K: content id: V: JSON object for the metadata:
  - Data:
  - K: content id: V: bytestream
  - How to store metadata:
    - Hashmap in memory
    - Row oriented in storage
  - How to store data
    - Filesystem
- APIs: create(K, V) for ingestion, read(K) from metadata and read(content id, offset) from filesystem

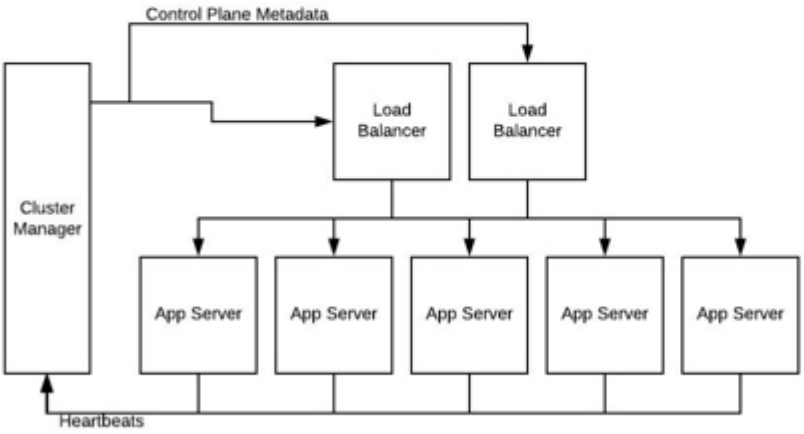


Step 4b for Content Microservice

- Need to scale for storage ? yes
  - A*B: A: number of K-Vs * B: size of a K-V
  - Metadata: 20000*10KB: = 200 MB storage is trivial
  - Data: 20000*10GB = 200 TB
- Need to scale for throughput ? yes
  - Number of create(K, V) /s = negligible
  - read(K, offset)/s: 3MB/s* 10 million users: 30 TB/sec
    - Assumption (Fair) : A single server delivers around 1 -2 GB/s (SSD) and 300 MB/s (spinning disks)
  - Metadata read is negligible
- Need to scale for API parallelization ? not relevant
  - APIs themselves are constant latency, so no need to parallelize
- Availability ? yes
- Hotspots : Definitely
- Geo-location based distribution ? yes, CDNs or content delivery networks

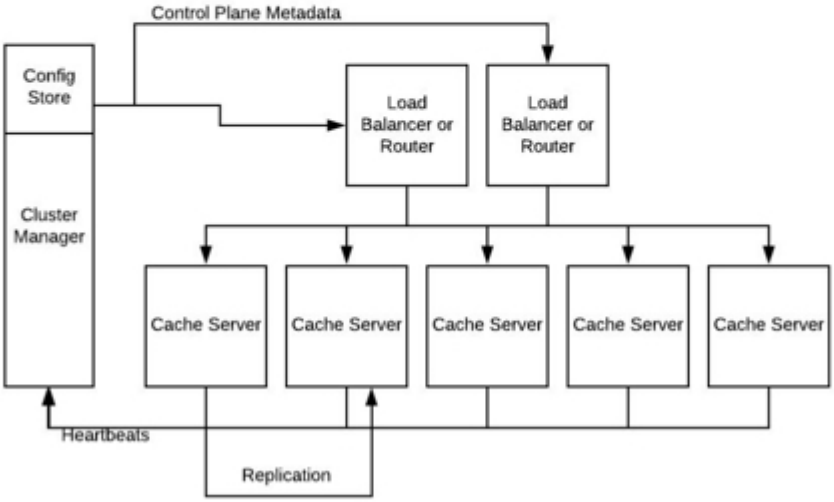
Step 4c for Content Microservice

Architectural layout for every tier



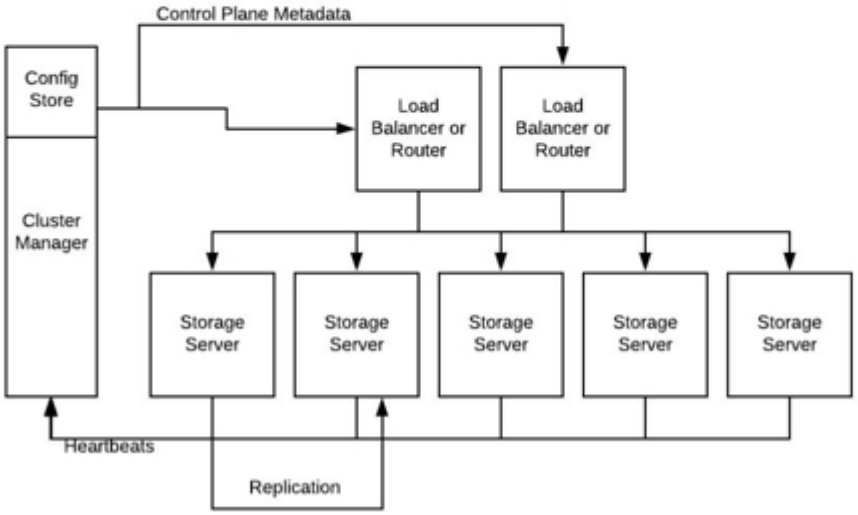
Step 4c for Vehicle Microservice

Architectural layout for every tier



Step 4c for Vehicle Microservice

Architectural layout for every tier



Step 4c for Content Microservice

- Sharding
  - Metadata: No need
  - Data:
    - Horizontal by content id:
    - Hybrid: content id + chunk id [128 MB]
- Placement of shards in servers
  - Consistent hashing
- Replication
  - Yes for availability as well as for throughput
- CP or AP?
  - AP: does not need to be strictly nanosecond level consistent
  - Best configuration: N = 3 or more, R = 1, W = 1, high throughput, lowest latency

K: V (thin or fat)

K: content id 1: V: bytestream

K: content id 2: V: bytestream

K: content id 3: V: bytestream

K: content id 4: V: bytestream

Using hybrid

hash(content id)range[0MB -128MB] -> shard -> servers

Step 4a for Recommendation Service

User activity tracking

App, Cache, Storage

Data Model: <User id/Content id>: Like, Ratings, offset at which the user

APIs: CRUD APIs

How will I store: Typical in-mem and storage K-V stores

Dashboard

App, Cache, Storage

Data Model: User id: Ring buffer of content ids per genre

APIs: CRUD APIs

How will I store: Typical in-mem and storage K-V stores

Step 4a for Recommendation Service

Recommendation Generator

Cache, Storage

- Batch consume data from user activity microservice
- Create structures as TF-IDF (inverted indexes)
- **Content based filtering**
  - Purely based on user and content relationships
- **Collaborative filtering**
  - Based on some interests and similarity with other users,

Step 4b for Recommendation Service: User Activity

- Need to scale for storage ? yes, probably
  - Amount of storage in cache and storage tiers: Number of unique location ids times size of each K-V pair
- Need to scale for throughput ? high
  - Keeping track of movie offsets
- Need to scale for API parallelization ? no
  - APIs themselves are constant latency, so no need to parallelize
- Availability ? yes
- Geo-location based distribution ? yes

Step 4b for Recommendation Service: Dashboard

- Need to scale for storage ? yes, probably
  - Amount of storage in cache and storage tiers: Number of unique location ids times size of each K-V pair
- Need to scale for throughput ? low
- Need to scale for API parallelization ? no
  - APIs themselves are constant latency, so no need to parallelize
- Availability ? yes
- Geo-location based distribution ? yes

Step 4b for Recommendation Service: Generator

- Need to scale for storage ? yes, in a distributed filesystem
- Need to scale for throughput ? no
- Need to scale for API parallelization ? yes
  - Parallelize the algorithms
- Availability ? does not matter
- Geo-location based distribution ? does not matter

Step 4c for Any Service

Architectural layout for every tier same as previous microservice

Step 4c for User Activity Service

- Sharding
  - Horizontal sharding
- Placement of shards in servers
  - Consistent hashing
- Replication
  - Yes for availability
- CP or AP? Best effort
  - AP: does not need to be strictly nanosecond level consistent
  - Best configuration: N = 3 or more, R = 1, W = 1, high throughput, lowest latency

Step 4c for dashboard Service

- Sharding
  - Horizontal sharding
- Placement of shards in servers
  - Consistent hashing
- Replication
  - Yes for availability
- CP or AP? Best effort
  - AP: does not need to be strictly nanosecond level consistent
  - Best configuration: N = 3 or more, R = 1, W = 1, high throughput, lowest latency

Step 4c for generator Service

- Sharding
  - Map reduce like
  - Scatter gather
- Placement of shards in servers
  - Consistent hashing
- Replication
  - No need
- CP or AP? Does not matter
  -

Step 4a for Trip Management Service

- Tiers: App server tier, Storage tier
- Data Model: K-V pair
  - K: <Trip id>: V: *rider, driver, properties, state of trip*
  - K: <driver/Trip id>: V: *rider, driver, properties*
  - K: <rider/Trip id>: V: *rider, driver, properties*
- How to store in Storage tier - as a row oriented K-V store
- APIs
  - create(K, V) for model 1 when trip requested
  - update (K, V) for model 1 when when trip changes state
  - Update (K. V) for models 2 and 3 when trip completed
- Algorithm in APIs
  - Simple K-V workload
  - Not much meat
- Flow of APIs
  - Request trip -> app server calls Location service to get list of vehicles
  - Create trip
  - For each vehicle, send notification to driver
  - If driver accepts, update trip

Step 4b for Trip Management Service

- Need to scale for storage ? no
  - Amount of storage in cache and storage tiers: Number of trips per second*size of K-V pair per trip * sizing for a couple of years
  - Really minimal
- Need to scale for throughput ? no
  - Number of update API calls per second: few hundreds per sec
- Need to scale for API parallelization ? no
  - APIs themselves are constant latency, so no need to parallelize
- Availability ? yes
- Geo-location based distribution ? yes

Step 4c for Trip Management Service

Architectural layout for every tier same as previous microservices

Servers needed only for replication