SYSTEM DESIGN INTERVIEW

Distributed System

It's a group of computers which work together to complete a single task.Level of abstraction in which multiple nodes coordinate their action by communicating with each other on a network and still act as a single unit.

Characteristics of distributed system :-

1. Scalability
2. Availability
3. Reliability
4. Efficiency
5. Serviceability and Manageability

Scalability - In a distributed system that can continuously grow inorder to support increasing demand is scalable. It's of 2 types:

Horizontal scaling (scale by adding more server, can be scaled dynamically/no capacity limit, ex- MongoDB/Cassandra, LB required, Resilient i.e ability to recover quickly from failure, more Network calls, data inconsistency may happen)

vertical scaling (scale by adding more power to a single machine,it can be scaled till the capacity is reached beyond that involves downtime, ex- MySql, single point of failure, inter process communication, Data is consistent)

Advantage of Horizontal scaling - Cost efficient, Fault tolerance, Low latency (time taken to respond to req decrease since req won't travel far away)

Disadvantages of Horizontal scaling - Introduce complexity and involve cloning servers, need of sticky sessions, Consistency of data should be maintained in downstream servers such as caches and databases.

Reliability - It is the probability that a system will fail in a given period. In a distributed system to ensure reliability we need to remove every single point of failure. Or how long does the service function well until the next failure i.e Mean time to failure

Availability - the time a system remains operational to perform its required function in a specific period. I.e Mean time to failure / (Mean time to failure + Mean time to repair)

Load Balancer(Software LB(easy to setup + cheap) - HAProxy, Hardware LB(difficult to maintain/setup + expensive) - Cisco,Citrix**)**

It helps to spread traffic across a cluster of servers to improve responsiveness and availability of application/websites/DB. Also keep track of status of all requests and servers i.e have a health check monitoring system. It also helps with horizontal scaling, improving performance and availability.

We can put LB between - User and Web-Server (Internet) / Web-Server and Backend/Application/Cache Server / Server and Databases.

Benefits :- 1. Users experience faster and uninterrupted service.
2. Service providers experience less downtime and higher throughput.
3. Decrease wait time and response time for users.

Algorithm - Round Robin, weighted Round Robin, IP Hash(IP of client used to find server), Least connection methods, Least Response time method (req send to fewest active connection + lowest avg response time).

Single point of failure - Install a second LB, each LB monitors the health of the other and when failure occurs in active LB then passive takes over it.

Sticky Session - LB has to be configured (LB issue cookies,which contain data about user session, and store them) in such a way that if the same req or diff req from the same user comes then it has to redirect the req to the same server which it redirected when the user requested for the first time.

Disadvantage - Increased complexity, can be a bottleneck if we didn't configure it properly, single LB is a single point of failure configuring multiple load balancers further increases complexity.

Caching

Store result of an operation so that in future req. return faster, it's mainly found near the front-end where they're implemented to return data quickly without req server.

In-memory cache - These cache store entire set of data in their own memory ex - memcache and redis

Application Server cache - cache is placed on request layer node i.e local storage, when req is made data is returned from local storage. If request layer is expanded and we have multiple node so a LB will redirect request to any node which can lead to cache miss (think of item in cart in Amazon)

Distributed Cache - Each node own a part of cached data which is divided using consistent hashing function, so that if a required Node is looking for a piece of data then it quickly knows where to look for it.

Global Cache - All nodes use the same cache space. Global cache → LB → Server

Cache Invalidation- when data is modified in DB then we need to sync data into cache as well.

1. Write through cache - Data is written in cache and DB at the same time, so consistency is maintained but at the cost of high latency and performing write operation twice.
2. Write around cache - Data is written into DB and flushed from cache, when new req comes then cache fetched modified data from DB
3. Write back cache - Update data in cache and acknowledgment is sent to client for confirmation and write to DB will be done at regular interval or when traffic is less, has risk of data loss in case of cache failure.

Cache Eviction :- When the cache is full, we remove old entries using LRU,LFU,FIFO,LIFO etc.

Concurrency issue in caching :- When multiple client tries to update cache at the same time, then conflict occurs like if 2 clients compete for same slot updation and one of them wins then data can become inconsistent (say if 2 user withdraw 100 rs at same time from bank having balance = 100rs then data inconsistency is there since end entry in the cache would be 0 rs, but 200rs would be withdraw), so to avoid this we can store every updation in commit logs and some background process will execute all logs and update cache asynchronously. Or we can also use locks and when one client finishes processing then another client is allowed to process.

Generally, Caching works best when used to store static or infrequently changing data, and when the sources of change are likely to be single operations rather than user-generated operations

CDN (caches the static web content at one geographical location, when user req something then req goes to nearest CDN rather than main server hence reducing response time and providing security too**)**

There are many disadvantages of Global caching :-

1. Single point of failure
2. End user is far away from global cache => response time increases
3. Popular req send far away many time => wastage of bandwidth

To solve these problems, we store static data in nearby CDN (kinda cache), HTTP headers are used for configuring how the CDN cache gives a piece of content.

Features - Speed increases to serve data, downtime/load on server decrease,deliver compressed content to decrease bandwidth, provide security/block attack from outside to main server.

Pushing content -

1. Push strategy - when we deploy info to server then at the same time we deploy info to all CDN servers as well, this approach is costly. Works well with sites having less traffic.
2. Pull strategy - We deploy info to the main server, when a new req comes then the CDN req server and cache the response. Works well with site having heavy traffic

To update the content in the CDN server we use HTTP caching headers.

Sharding

Used to break-up the DB into small parts.

1. Horizontal partitioning - partitioning across rows i.e range based partitioning. If range is not chosen carefully that would lead to unbalanced load on the server.
2. Vertical partitioning - Divide data across columns, i.e storing user_id, phone_no in separate DB, problem is one feature would consume a lot of memory other won't i.e uneven load on server.
3. Directory based partitioning - query directory server that holds mapping b/w each row to its DB server. Ways to map each row :-
    a. Hash-based - apply hash to a primary key of entry, ensure equal distribution but number of server gets fixed or on adding new server => changing hash function => redistribution of data and downtime → can be solve by consistent hashing
    b. Round Robin
    c. List partitioning - partitioning according to the key like country/state.
    d. Composite - combining any of above schema ex list partitioning + hash based

When shards fails then use master-slave, write goes to master and reads are distributed across the slave (assuming number of reads > writes). When master fails then one of the slave become master.

Challenges with Sharding:-

→ Resharding data: Needed when:-

1) a single shard could no longer hold more data due to rapid growth.

2) Certain shards might experience shard exhaustion faster than others due to uneven data distribution. When shard exhaustion happens, it requires updating the sharding function and moving data around. {*Consistent hashing* is the solution}.

→ Celebrity problem: Excessive read/write req are coming to one shared because it contains data of celebrity people (who have large following).This is also called a hotspot key problem. Imagine data for Katy Perry, Justin Bieber, and Lady Gaga all end up on the same shard. For social applications, that shard will be overwhelmed with read operations.{*We might need to allocate a shard for each celebrity*}

→ Join and denormalization: It's hard to perform join operations across database shards spreaded across multiple servers. {*de-normalize the database so that queries can be performed in a single table.*}

Indexes

They are created to make a faster search operation to find a row. It's a data structure of a table of content that points to the location where actual data lives, key/index -> pointer to the row. Ex - if we want to sort a table according to a column (say marks) then we need to store the whole row and a pointer pointing to that row in the index table, so that after sorting row data doesn't get mis-mapped.

It's a self balancing B-tree that keeps data sorted and allows search/delete/update/insert operation in logarithmic time.

Disadvantages - Using Indexes means using more space. Also writes could be slow since along with DB we need to update the index as well.

Proxies

Proxy is an act which someone else does on your behalf, Proxy server lies between client and back-end server. They are used to filter req, log request and sometimes transform req (by adding/removing header, encrypting/decrypting or compressing a resource), also it caches the resource i.e it merges the same req from multiple users into one. Useful in high load situations/limited caching.Provide anonymity and used to bypass IP address blocking.

Forward Proxy - Placed between client and web, act as a security/firewall for client i.e don't reveal client IP to web, caching, the org where client belongs can block certain websites using forward proxy.

Reverse Proxy - Placed between web and server, It retrieves resources on the behalf of a client from one or more servers, these responses are returned to the client as if they have originated from proxy itself.
Advantage - Maintains anonymity of server, Can act as a LB for server, Protects server from any kind of attack, manipulates req according to server requirement, caching, compress req which leads to decrease bandwidth and load on server.
Disadvantages - Increased complexity and single reverse proxy is a single point of failure, configuring multiple reverse proxies (ie a failover) further increases complexity.

Load Balancer vs Reverse Proxy
LB is useful to distribute traffic when there are more than one server present, but RP can be used even with one server along with benefits of security, scalability, compression, caching etc which LB does not have.

Redundancy and Replication
Redundancy is duplication of critical components, to help in removing a single point of failure by keeping a backup if needed in a crisis. Ex- if we have 2 instances of a service running in production one fails then sys can use another instance. In short it is basically making 1 or more alternatives (i.e. backups) to the element that is critical for high availability.
Replication is sharing info among redundant resource (mechanism of copying data from one resource to another) like Master-Slave, copying the data from one DB to another inorder to improve fault tolerance, relaibility and in case if one fails. Replication means to duplicate (make copies of, replicate) your database.
Replication ensures redundancy in the database if one goes down. But now the problem is to synchronize the data. Replication on write and update operations to a database can happen synchronously (at the same time as the changes to the main database) or asynchronously .
Basically, replication is master-slave where both master and slaves are used while redundancy is having a backup of critical component for emergency

Message Queue
(Ex- Kafka**) -** These are used to async communication between microservices. They store all the messages produced by producers in the queue until they're consumed by consumers. They're also used as a fault tolerance in case of service failure they can retry requests. Message queue is the combination of Notifier, LB, heartbeat mechanism and persistence (storage). It takes tasks → persists them (store them) → assign them to the correct server i.e balances load → waits for them to complete, if it takes them too long to ack then it assign it to next server (check heartbeat of server)

CAP theorem [Consistency, Availability, Partition Tolerance]
In a distributed system :-
It states that it's impossible for a system to provide C/A/P all at the same time. Basically it's the trade off between consistency and availability in presence of partition.
Consistency - All the users see the same data at same time and all update themselves when data at one place changes.
Availability - System continues to function even with a node failure.
Partition tolerance - System continues to function even if communication fails b/w the nodes.
Since partition failure can't be avoided in a distributed system (coz networks aren't reliable) so we assume that it will happen => we need to choose between consistency and availability. (No partition failure in centralized system i.e one server)
In a centralized system :-
we need to choose between consistency and latency.

**NOTE - In the case of Banks they choose consistency over Availability. Once an ATM is disconnected from the network you would not want to allow withdrawals (thus, no Availability). Or, you could pick partial Availability, and allow deposits or read-only access to your bank statements.

Consistency Pattern

1. Weak Consistency - After a write, readers may or may not see it. Ex - in a Video call when we lose reception for a few seconds, when you regain connection you do not hear what was spoken during connection loss.
2. Eventual Consistency - After a write, readers will eventually see it (typically within milliseconds). Data is replicated asynchronously. This approach is seen in systems such as DNS and email. Eventual consistency works well in highly available systems.
3. Strong Consistency - After a write, readers will see it. Data is replicated synchronously. This approach is seen in file systems and RDBMSes. Strong consistency works well in systems that need transactions.

Availability Pattern

There are two complementary patterns to support high availability: fail-over and replication.

1. Fail-over :-
   a). Active-Passive - also referred as Master-slave, With active-passive fail-over, heartbeats are sent between the active and the passive server on standby. If the heartbeat is interrupted, the passive server takes over the active's IP address and resumes service.
   b). Active-Active - also referred as Master-master, In active-active, both servers are managing traffic, spreading the load between them. If the servers are public-facing, the DNS would need to know about the public IPs of both servers. If the servers are internal-facing, application logic would need to know about both servers.
2. Replication - Master-slave archieture, Master-master archiecuture.

Master-Slave architecture

The true data is kept at the master database i.e it is write only and the data is copied asynchronously to the slaves from the master database.The slaves are typically used when the master fails or for batch processing. If the clients don't need live data then master database do the same work as other replicas slaves but an additional task is controllers in this case the masters are chosen arbitrarily between one of these replicas and if the master fails then the replicas choose the new master automatically or we can have a coordination service (*zookeeper)* to choose master.

Master has all writes, slaves have all reads.

Problems :-

→doesn't increase write speed, lag while copying data from master to slave and a read req comes to slave which read outdated data.

= When data increases then divide slaves by range and store data. (Problem - failure in complex queries, joins become difficult)

→In case of master or slave failure

= When a slave becomes unavailable then read req is temporary transffered to master, once slave becomes available then read req are again sends to slaves.

When master become unavailable then one of the slave takes over and become master using software like etcd (which is a store of key-value pairs that offers both high availabity and string consistency by using consensus algo)

Why do we need it?

When multiple servers are performing the same task then we need to ensure that only one server is given the responsibility for updating some third party API because multiple updates from different servers

could cause issues or run up costs on the third-party's side. Also, to maintain data consistency (since writes are only at master).

Master-Master architecture
Both masters serve reads and writes and coordinate with each other on writes. If either master goes down, the system can continue to operate with both reads and writes.
Disadvantage - A LB is needed to determine where to write, most master-master systems are either not consistent (violating ACID) or have increased write latency due to synchronization, conflict resolution comes more into play as more write nodes are added and as latency increases.

Trade-off -

1. Performance vs Scalability - When a system is slow for a single user then there is a performance issue, if it's fast for a single user or slow under heavy load then there is scalability issue.
2. Latency vs Throughput - Latency is "how fast" i.e in how much time the req is made and how fast it'll get the data back from the server, throughput is the amount of data that can be sent per unit time.
3. Availability vs Consistency - choose consistency when requirement dictates atomic read and write. Choose availability when requirement allows for some flexibility in sync data.
4. Single point of failure (create back-up/use master slave/create multiple instances[master-master])
5. Scaling cost (do horizontal scaling)
6. Latency increase to communicate with far away server (use CDN)
7. High Availability
   → 2 ways - Fail-over and replications
   a). *Active-Active / Active-Passive*
   Active-Active – same as master-master, both server respond to req and we need LB to route traffic, when one fails then all load is transferred to other server.(problem is sync data updation)
   Active-Passive – same as master-slave and a heartbeat sys is to check when "active" fails.
   b). Make replica i.e have *back-up*, to ensure consistency in b/w replica and original we do :- a.keep a local copy of updated version in coordinator i.e zookeeper, when update will fail then coordinator will re-do the operation, b.for every operation in each server we've a commit log like we've in github which is a history of all updates, so when a update fails then we can lookup in commit log and update replica as well.
8. High Consistency (compromise availability)
9. Sliding window type pointer

Avoid single point of failure

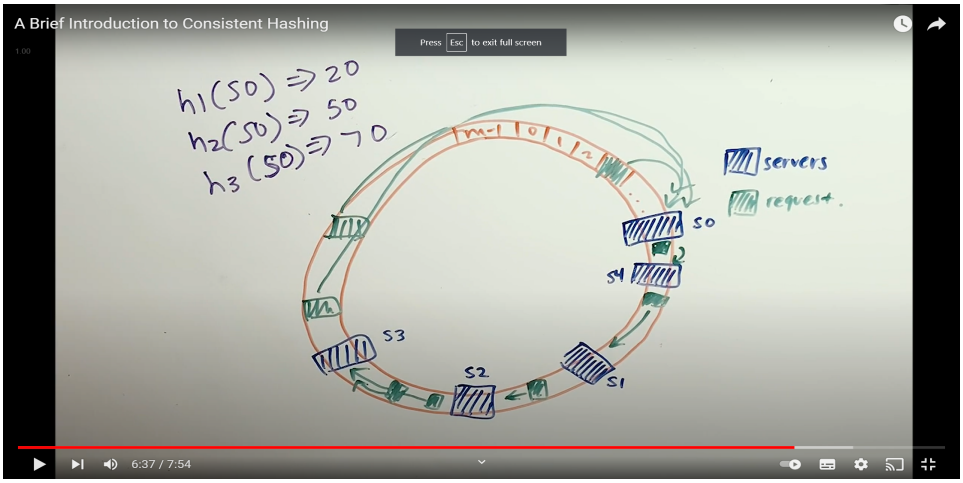
1. Use multiple instance [master-master] of every component in a service i.e more nodes/servers or backups
2. Master-Slave architecture
3. Use backups

In rare cases when the whole system fails, you should place the same system in different geographical locations. (when asteroid hits so place human in mars for humanity to continue)

Consistent Hashing
In case of normal hashing, the hash function is key%N, where N is no of the server. If N is increased or decreased then we need to change the hash function and all the keys need to be remapped which takes a lot of downtime.
Consistent hashing allows us to distribute data across a cluster in such a way that will minimize reorganization when nodes are added or removed, hence it becomes easier to scale up or scale down.

When a host is removed from the system, the objects on that host are shared by other hosts; when a new host is added, it takes its share from a few hosts without touching other's shares.

We make a conceptual ring of hashes and then calculate the hash of all servers and then we calculate hashes of req ID and map it onto the ring and will send that req to the server that is immediately clockwise to them. In this case uniform distribution is assumed but it's not practically true, so to uniformly distribute req we use virtual node i.e calculate hashes of server using various hashing algo and place server at more than one location and then route req to nearest clockwise server, i.e Hash_1(S1) = 20, Hash_2(S1) = 40, Hash_1(S2) = 30, so now S1 will handle → 0-20 + 30-40 and S2 will handle → 20-30, now when a req comes Hash(req) = 15 → routed to S2



Sync vs Async connection
Sync connection is when a client sends a request to the server and expects an immediate response. Also, the client should receive the response in an ordered manner. ex- Video calling.
Async connection is when a client sends req to server and doesn't expect immediate response or continuous exchange of info is not necessary also not necessary in an ordered manner. ex- texting someone.

PUT vs POST vs PATCH HTTP request
PUT - When we want to modify a resource which is already present in resource collection, it overwrites the entire resource.
PATCH - If you want to update a part of a resource, it updates the resource without overwriting it.
POST - When you want to add a new resource in a resources collection.

HTTP short polling vs Long polling vs Web-Sockets vs Server-Side events
HTTP req → Client makes req to server, server computes the response and sends it back to client.
HTTP Short polling/Ajax Polling → Client keeps sending req to server at a fixed interval of time and server computes the response (can be empty) and send it back to client, problem is wastage of bandwidth/unnecessary network call.
HTTP Long Polling → Client make req to server using HTTP req and waits for server response until then connection remain open (till a time limit, if time out occurred and response is not received then client again sends another req to server), after receiving response connection gets closed.
Web-Sockets → Bi-directional communication channel and it sits over TCP protocol, first client does handshaking(establishing connection) with server and then TCP connection gets established b/w client and server. Clients can send data to the server and vice versa. Reduce Overhead of handshaking again and again i.e we're establishing connection only once at the beginning and not closing it, ex- whatsapp.

Server-Side events - Server sends data to client via HTTP connection but client can't, used in real-time stock exchange price Server pushed tech. Enabling clients to receive automatic updates from a server without requesting it.It is based on HTTP protocol.

API Gateway

It's a single entry point where all the customer goes and hits the API gateway decides which service to call and redirect requests there and give the correct response to the client.
Ex- Apigee[Google],Tyk[OpenSource],Amazon Api management
Why is it needed?

1. Filter users and find which API (Internal/External) needs to be exposed to the user.
2. API analytics (most frequently used API, slowest/fastest response API, API traffic)
3. API security - authentication/authorization of user, rate limiting
4. Load balancing

Zookeeper

It's a part of hadoop software. In a distributed system it's used to coordinate among nodes in a cluster where multiple nodes are involved i.e distributed application by sharing config info of all the servers across all the servers, select leader/Master from all servers. It also monitors the health of nodes in a cluster by periodically sending heartbeats.
Disadvantage - hard to maintain, high operational cost, a single point of failure.

Choice of DB

Factor impacting choice of DB - structure of data you have, query pattern you have, amount of scale needs to be handled.
1.Use *Redis for cache* - work as a key-value pair, use because it's mainly used in big tech and is stable. To store *data in form of images/videos* we use *Blob Storage* (Binary Large Object included objects such as images and multimedia files) (used because it allows user to store large amount of unstructured data on storage platform), provider for Blob Storage is Amazon S3 (AWS) along with CDN to avoid overloading AWS server and distribute data across globe.
2.For *text searching* you need fuzzy search, so for this you need to use Text Search Engine which is provided by *ElasticSearch* and Solr, both are built on top of Apache Lucene. They store data by indexing it, so that it can perform a read request in huge data in ~O(1) time .
3.For storing *metrics kind of data* use *Time Series Database*, in this we can't do random update, but can do sequential update in append only mode like first entry at T1 then second entry (can be seen as updating DB) at T2, third at T3. Also read queries are bulk read queries with time range like for the last few minutes/hour/day etc. Ex - InfluxDB, openTSDB (open time series database), Amazon Timestream..
4.When you want to *store large amounts of data(analytical data used for analysis)* for doing various analyses then we need a *Data Warehouse* (basically a large database where you can dump all the data and provide various querying capabilities on top of the data). Ex - Hadoop.
5.When dealing with *map objects* → use Graph database

Relational vs Non-Relational DB -
Structured data (in form of row/column) [YES] → ACID properties required [YES] → RDBMS (eg MySql,Oracle, SQL server, Postgres)
Structured data (in form of row/column) [YES] →ACID properties [NO] → choose any relational/non db
Structured data (in form of row/column) [NO]→ Storing different product having different unique attributes → We need to run fast read queries on vast varieties of different attributes → DocumentDB (eg - Couchbase, MongoDB) → *many attributes i.e complicated data*

Structured data (in form of row/column) [NO] → limited variety of attribute i.e simpler data but size of DB is ever increasing → Columnar DB (eg - Cassandra, HBase) (mostly use Cassandra as it is easy to deploy while for HBase we need to setup Hadoop first) → *Huge data but less variety in queries*

SQL vs No-SQL
SQL - Fixed table design (modifying it requires changing the whole DB and going offline) and is vertically scalable hence very expensive to scale and lookups by indexes are fast here. Use only when we have structured data, the schema is not likely to change, need to perform complex joins and want ACID compliance.
No-SQL - Dynamic schema and rows doesn't need to contain all col, horizontally scalable hence cheaper to scale, sacrifices ACID compliance for performance and scalability. Use only when schemas are dynamic, when large volumes of data with rapid development needs to be stored, when data needs to be stored across servers of different regions. NoSQL is efficient for data like Metadata/lookup table, Frequently accessed table, log data, temporary data like shopping cart.

Consistency + Availability → MySql, Oracle
Consistency + partition tolerance → Redis, Couchbase, MongoDB
Availability + partition tolerance → Cassandra, CouchDB

Types of DB :-
1. Graph DB - Model data with nodes and edges i.e good at representing data with lots of relationships. Ex - CosmosDB
2. Document DB - A key is mapped to a JSON object i.e a key mapped to an object having various attributes like in amazon product_id mapped to it's attribute. Ex - MongoDB, DynamoDB, CouchDB
3. Key-Value store - A key is mapped to a single value, it uses a hash table for implementation hence read, write, update, delete operations are fast. Ex - Redis, memcached.
4. Column DB - groups related columns for storage (easy to scale). Ex - Cassandra, HBase.
5. Search DB - It has a large amount of unstructured data and provides fuzzy search service. Ex - ElasticSearch, Solr.
6. TimeSeries DB - Data is ordered and sorted by time. Ex - InfluxDB, Amazon Timestream

Cassandra
We chose Cassandra because it is fault-tolerant, scalable (both read and write throughput increases linearly as new machines are added). It supports multi data center replication and works well with time-series data.Cassandra is a wide column database that supports asynchronous masterless replication. Best suited when we have multiple data storage centers in multiple locations.

Monolithic vs Microservice
Monolithic is a big container where all the software component lies and are tightly coupled,
Pros -
1. Good for cohesive and small teams
2. Less duplication
3. Faster, no network calls

Cons -
1. When a new service is added or even a small update is made then the whole app needs to be deployed.
2. Scalability issues
3. Single point of failure.

Microservice - Diff components are made for different services communicating through REST Api, each service has their LB, indexes, caches.

Pros -

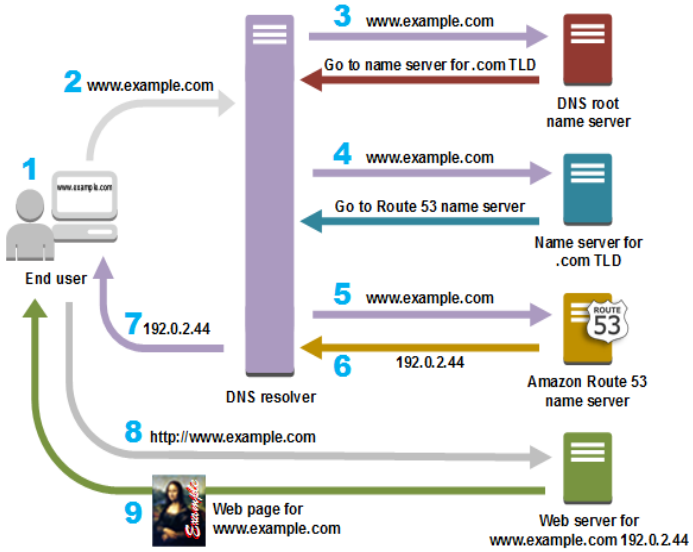
1. Easier to scale the codebase, loosely coupled so adding new functionality is easy.
2. Easy for new developers to work without understanding the full functionality of the app.
3. Easy to work in parallel.
4. Easy to scale.
5. Decoupled.

Cons -

1. Harder to design i.e skilled architecture needed.

DNS

Store mapping of domain name (www.google.com) to its IP address. In order to communicate with devices in internet we need the address of other devices so this address is IP address, now since each device cannot store address of all other devices in the world and same with server so we created DNS which maps the domain name (or the server/device which we're trying to reach) to it's IP address. Process -> Typed something → browser looks into cache to find IP → if not found then look into DNS cache on its' machine → if not found then look into host file on it's machine → If no found then req goes to Recursive DNS system i.e ISP and will look into cache → if not found then req goes to Root DNS server and will look into TLD(a place where info of all .com/.xyz reside) info → look into TLD server which returns Authoritative Name server → Authoritative Name server contains data of all domain name and their IP.



TCP/IP

Data over IP is typically sent in multiple packets because each packet is fairly small (2^16 bytes). Multiple packets can result in (A) lost or dropped packets and (B) disordered packets, thus corrupting the transmitted data. Hence TCP solves these problems by **guaranteeing transmission of packets in an ordered way.** Since it's built on top of IP and the packet has TCP header and IP headers where TCP header contains information about the ordering of packets, and the number of packets etc. This ensures that the data is reliably received at the other end. It is generally referred to as TCP/IP because it is built on top of IP.

TCP needs to establish a connection between source and destination before it transmits the packets, and it does this via a "handshake".

Transport layer
The transport layer is used to get data from the application layer and break it down into smaller pieces (packets) and attach meta info to each packet like src/dest of ports and other info which is used by the receiver transport layer to combine the incoming data and pass it to the receiver application layer.

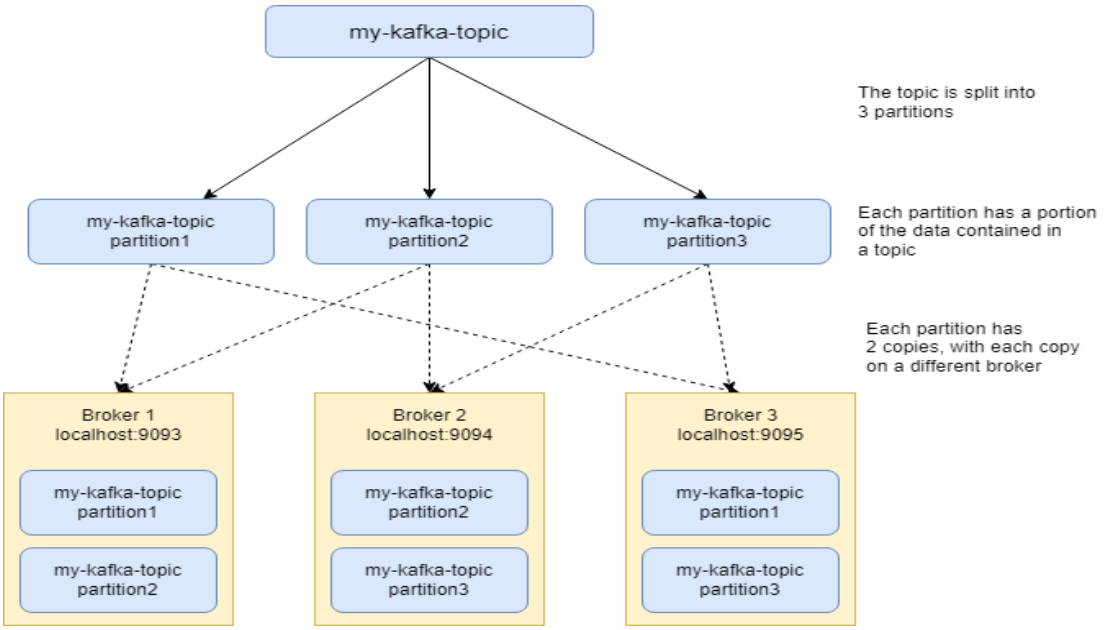
TCP
TCP is a connection oriented protocol (needs to establish connection b/w client and server inorder to send packets to receiver), a connection is established by three-way handshake i.e client sends syn packet (equivalent to saying “hey I want to establish a connection”) to server, then server reply by syn ack packet (equivalent to saying “hey I’m ready to accept connection”), then client reply by sending ack packet back and now client and server have a connection established and now packet transmission can start.
TCP packets have info attached to each packet in the header which is used to identify the order of arrangement, that’s why header size is 20 byte.
Features of TCP :- Packet transmission, Delivery Acknowledgement, Retransmission of lost packet, Order delivery, Congestion control.
Now the client sends a packet to the server and the server needs to send an ack back that packet has been received, and if the packet gets lost in mid then the client waits for some time and re-transmit the packet to the server.
Order delivery → Client sends the packet to the server in any order, then the server sorts them with the info provided in the header and sends it to the application layer.
Congestion control → Client has the choice to slow/fasten the transfer of packets depending upon network traffic, like if we notice many packets are getting lost due to heavy traffic so we’ll slow down sending packets to the server until congestion is removed.
EX- File transfer, text messages

UDP
UDP is a connectionless protocol (with header size(8 byte) smaller than tcp header size, since it does not contain meta info about ordering), when data comes to transport layer it divides it into packets and without establishing connection to server it start sending packets and also since server don't need to send ack so client need not to wait and start sending packet, if the packets get lost then they’re not going to be re-transmitted, also there is no order delivery so server is not going to rearrange them before sending to application layer, server gonna send packets to application layer as it receives them from client, also no congestion control in udp so the client continue to send packets at the rate at which it is transferring the packets.
EX - Video/audio streaming

OSI Model
OSI (Open system interconnection) Model - used to send data from sender to receiver.
Application Layer (provide network services to user like HTTP/FTP/SMTP) -> *Presentation* (find in which format data should be send so that receiver can understand it also encrypt data for security reason)-> *Session* (responsible for authentication and reconnection if network disconnects) -> *Transport Layer* (divide data in chunks, contain TCP(establish connection first and send acknowledgment)/UPD(fast sending data but no acknowledgement)) -> *Network Layer* (divide data into packet, store IP of sender n receiver and redirect data to correct address i.e routing data) -> *Data Link* (remove error from data, maintain data transferring speed from receiver and sender, find physical address of receiver) -> *physical* (send data to physical address of receiver).

Kafka

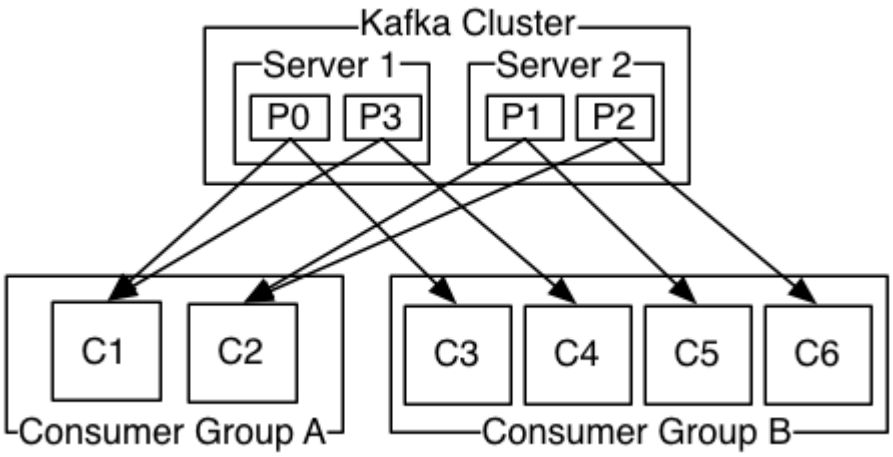
- Instead of sending msg through Ajax calls one by one we put them in a queue and then send it.
- It's used to communicate between 2 software component and it's async (it means sender and receiver need not to interact with messaging service at the same time)
- It takes tasks, stores them and assigns them to the correct subscriber and removes them when tasks get executed.
- In API calls producer is aware of consumer => highly coupled .Kafka decouple consumer and producer and make them unaware about each other and also eliminate spikes.
- Used as a queue + pub/sub
- It's Horizontally scalable, since it can have thousands of topic + have low latency ~real-time
- Kafka cannot be a single point of failure since it's assumed to be resilient.
- It's fault tolerance, in case of any failure we can retrieve lost data from the queue.



Terms :-

Producers - Producer is responsible for publishing the data. It will push the data to the topics of their choice. The producer will choose which record to assign to which partition within the topic.
consumer - Consumers will consume data from topics. A consumer will be a label with their consumer group. If the same topic has multiple consumers from different consumer groups then each copy has been sent to each group of consumers.
Cluster - contain 100s of broker
Topic -It categorizes the data. Topics in Kafka are always subscribed by multiple consumers that subscribe to the data written to it.
Broker - Which is responsible for holding data. Each Broker holds no of partition
Partition - Topics are further splitted into partitions for parallel processing.
Offset - each msg within the partition has an msg id called offset
Topic + partition + offset = location of a particular msg
Consumer group - group of consumer who read data in parallel
Zookeeper - Management of the brokers in the cluster is performed by Zookeeper

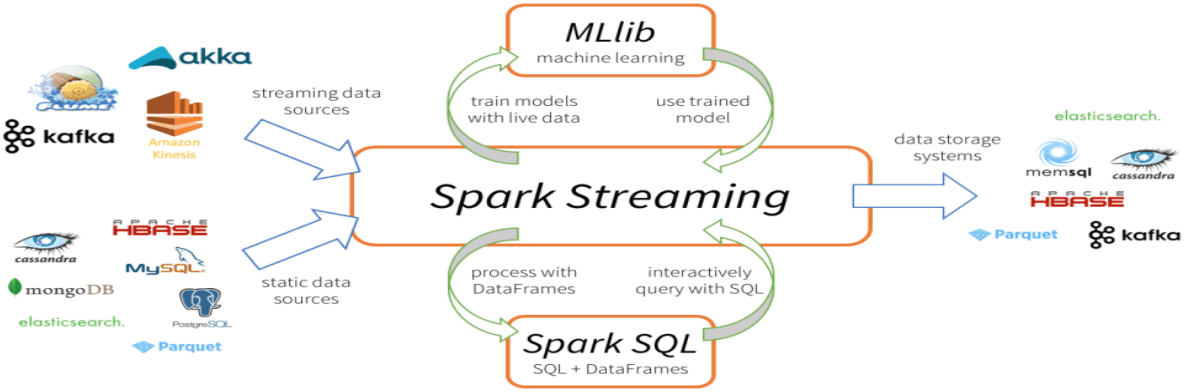
If you want Kafka to act like queue -> put all consumers in one group
If you want Kafka to act like pub/sub -> put all consumer in different group because each partition can be consumed by multiple consumers in different group because the partition is group dependent



Spark Vs Kafka
Spark - Micro batching is done for life data (though time for making a batch is small ~real-time), Can process static + stream data
Kafka - Process data in real time, only processes stream data

Batch processing vs Micro-Batching vs Stream Processing vs Real-time Processing
Batch data processing -> data is downloaded as batches or large chunk for processing
Micro-batch processing -> data is collected in small groups ("batches") and then processed, data processing occurs more frequently than batch processing since batches are small, data is collected based on predetermined threshold or freq, ex - Logstash, Apache Spark streaming (less time for making new batch ~10ms)
streaming data -> continuous flow of data with simultaneous data processing/analyzing, in real-time the moment they're created, ex- Hazelcast Jet with Hazelcast IMDG (and as a managed service as Hazelcast Cloud) enable you to collect, process and analyze data in real time.

How Stream Processing works?
Stream processing is based on the pub/sub model. Data generated by publisher (streaming data source-Kafka or static data source-MySql/MongoDB/Cassandra) -> processing/analyzing/transformation (using Spark SQL/MLlib) of data done inside streaming engine -> delivered to subscriber (can be file system or stored in DB or live dashboard)



Spark Streaming divides live input data streams into batches of 500 ms intervals which are processed by Spark Engine. Internally Data Stream is a sequence of RDD (Resilient Distributed Dataset)

Challenges of Real-time Application.
Scalability - when something fails inside streaming processing machines or the current streaming service can't handle the incoming data spikes then the incoming raw generated data will log exponentially from kilo to giga byte in few second, so for scaling (adding more capacity/resources/servers) need to be done instantly.
Ordering - In chat app data must be in ordered format, or when a developer is looking at log to debug issue then ordering is needed, there are discrepancies in timestamps and clocks of the device generating data. hence ACID properties must be taken care of.
Consistency and Durability: Data consistency and data access is always a hard problem in data stream processing. The data read at any given time could already be modified and stale in another data center in another part of the world. Data durability is also a challenge when working with data streams on the cloud.
Fault Tolerance & Data Guarantees: these are important considerations when working with data, stream processing, or any distributed systems. With data coming from numerous sources, locations, and in varying formats and volumes, can your system prevent disruptions from a single point of failure? Can it store streams of data with high availability and durability?

Design Stream Printer Service
Code - **LeetCode**

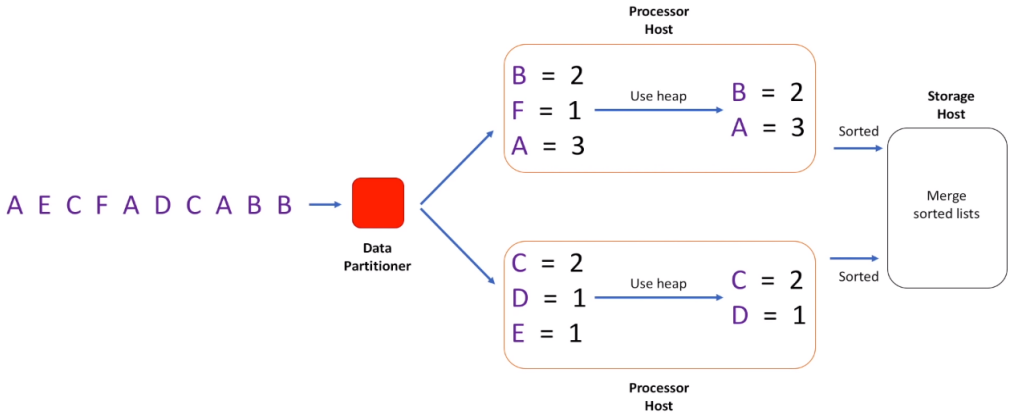
Design Parking Lot service
Code - **LeetCode**

Design a system that allows users to retrieve stock info as fast as possible.
Could be client push or server pull, When your app/website asks server for data then its client pull and reverse is server push.
Long/short Polling (client Pull)
Web-Sockets (Server Push)
Server Side events (Server Push)

So stock info can be retrieved by SSE and pushed into streaming queues like Apache Kafka or Spark. Or can be retrieved using TCP protocol.

How to transfer data from Bbg service to your service?
Bloomberg | Design a system to give prices of a stock
We can get the data by calling the external stock exchange API (on the top of TCP protocol) which will be called at the fixed interval using a cron job based on the user's requirement. Or we can use Web-sockets (which sits on top of TCP) or using SSE events

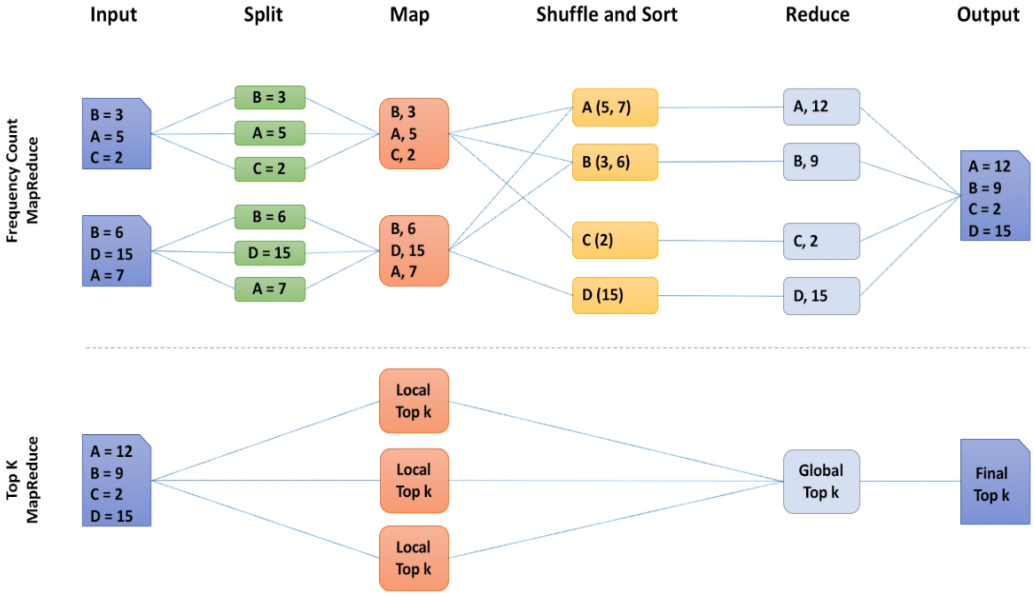
Design problem: implement k largest elements with a heap with P server



Storage host -> use merge N sorted list algo n retrieve K largest element
parallel processing is done through P computers => an array is given which is partitioned into p processor host using a data partitioner => in each host we can have a heap which will find top k element for that processor host and then at the end we will merge all the top K elements in each partition and obtain single list of all top k element.

Design problem: merge multiple incoming streams with a priority queue

1. Since we have an infinite number in each stream, we can assume the stream as a Linked List. Now to merge all incoming streams, we can use Heap, and put the starting pointer of the list in the heap and obtain the smallest value and remove it, then put the next element of that list into Heap.
Code - [Merge Multiple Incoming Stream using Priority Queue]
2. Map reduce can be used (In output side we can use priority queue) -
Merge(B1,B2) = res_1→ Merge(res_1,B3) = res_2 —> Merge(res_2,B4) = res_3;



How do you send data between two remote-systems ?

Using FTP server, File Transfer Protocol is a way to connect two computers and transfer files between them over a network like the Internet.Using an FTP client we can upload, download, delete, move, rename and copy the file on a server.
Anyone can upload files to a FTP server and others can connect to the server and download files via FTP protocol. Also, if you want then you can also configure the laptop to act as FTP server (In windows, it can be done using Internet Information Service Manager.)

You can access FTP server either via the Internet (just by typing ftp.example.com) or by using an FTP client application (like FileZilla).
SFTP (Secure File Transfer Protocol) - More secure than FTP since it transfers data in encrypted form and also adds authentication layers for users.
Both FTP and SFTP use TCP for file transfer so the delivery of the file is guaranteed.

How to proceed in the System design round?

1. Functional + Non-functional req (clear requirement, consistency vs availability or real time vs little latency? or how much to scale? or what happens in case the system goes down?)
2. API creation
3. Database Design [Table creation] [Data structure of table]
4. Logic to solve problem/Algorithm
5. System workflow
6. Load Balancer
7. Caching
8. Sharding
9. Indexing
10. Message queue
11. Hashing (Consistent Hashing)
12. LRU
13. Hadoop
14. Cassandra
15. Microservice

Non-Functional Req →

1. High Availability
2. Fault tolerance
3. Scalability
4. High Performance
5. Storage
6. Durability
7. Capital and operational cost
8. Security
9. Monitoring
10. Logging
11. Maintenance
12. Latency

Requirement clarification -

1. Users/customer
    a. Who will use the system?
    b. How will the system be used? (Help us understand what data will be stored in system)
2. Scale (read/write) (how much data is coming to the system and how much data is retrieved from the system.)
    a. How many read queries per second?
    b. How much data is queried per request?
    c. How many video views are processed per second?
    d. Can there be spikes in traffic?
3. Performance (Help us to evaluate different design options)
    a. What is expected to write to read data delay? (choose b/w batch/real time data processing)

    b. What is the expected latency for the read query? (data must already be aggregated)
4. Cost
    a. Shoule design minimize the cost of development? (Hint - use open source framework)
    b. Should design minimize cost of maintenance? (Hint - use public cloud service for storage)

Design TopK stocks
Stocks Msg are transferred from one service to another by using TCP.
Code - https://leetcode.com/playground/VGhBfPUM
Source - topK

Clarifying ques - Latency? Traffic? Accuracy/consistency? Availability?how large K can be(few thousands are okay, but ten thousands can cause performance issues)?, how many stocks do we have?

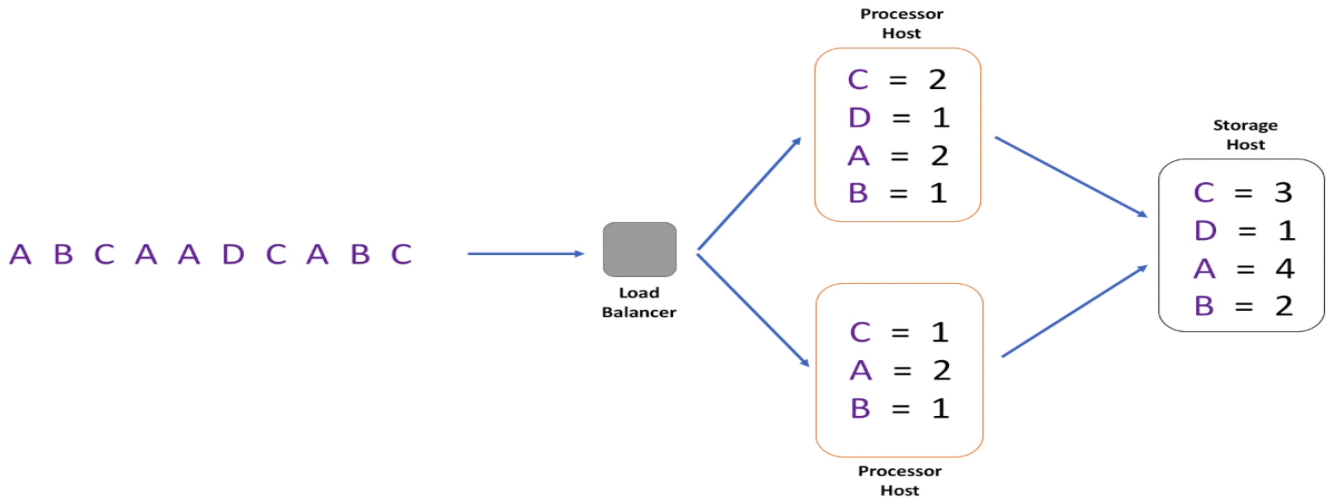
Say this first → **"**We can solve this using Kafka and some stream processing framework like Apache Spark (Although Spark internally uses the same approach which we used in this design i.e data partitioning and in-memory aggregation). Spark internally partitions the data and calculates topK for each partition using a heap and all these lists are merged together in reduced operation.

Functional - getTopK(k,startTime,endTime)
Non-Functional -
1. Scalable (scale together with increasing amount of data)
2. Highly Available (survives hardware/network failure, no single point of failure)
3. Highly Performance (few tens of ms to return top 100, hint - topK list should be pre-calculated)
4. Accurate (as accurate as we can get)

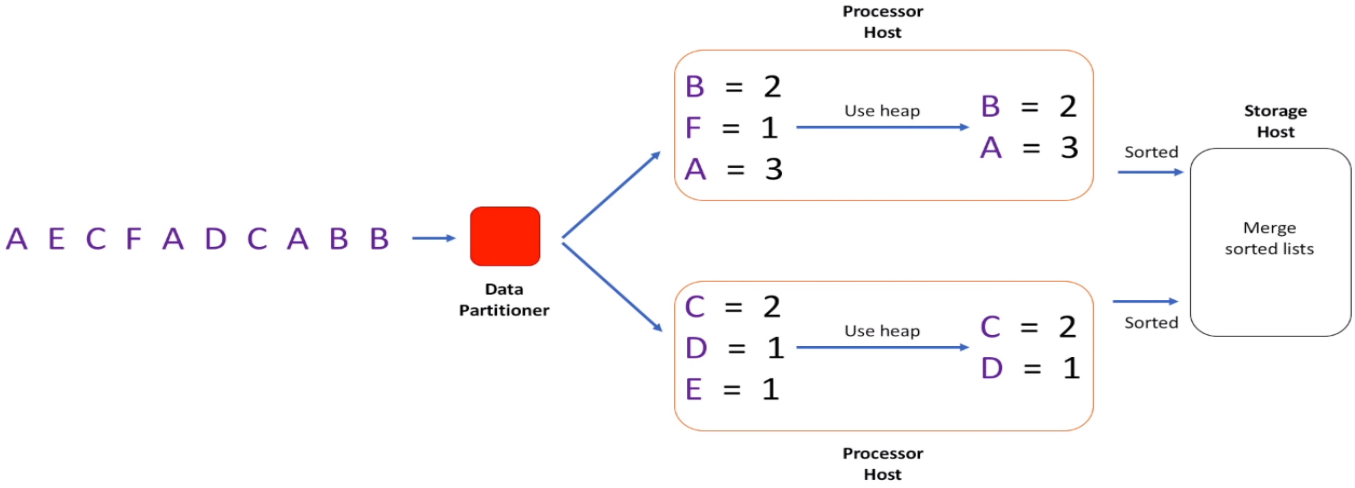
API's - getTopK(k,startTime,endTime), updateStockVolume(stock_id, vol_traded)

**Note -
1. Accuracy not imp → Fast Path
2. Accuracy is imp and result should be calculated in min → Slow Path (we need to partition the data and aggregate in memory)
3. If time is not an issue + Accuracy is imp + data set is big → Use Hadoop Mapreduce.

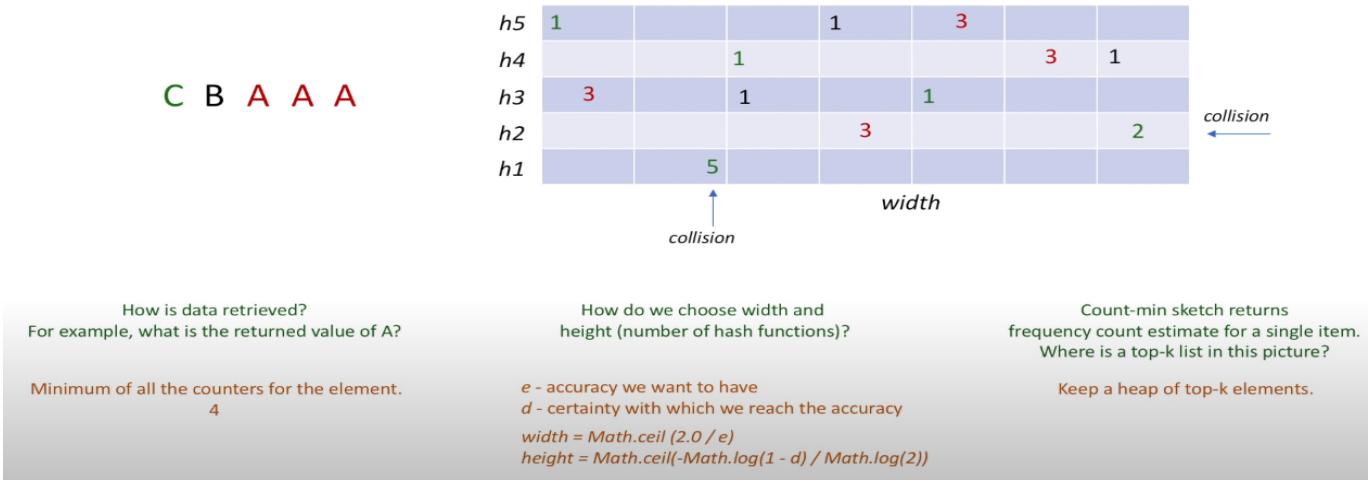
Steps :-
1. we use single host to store hashmap which has freq count of element and then sort it using heap (normal coding ques) {will have scalability and performance issue}
2. To solve above issues we use a Load balancer to distribute the load to different hosts {memory storage issues are there,if hash map size increases}



3. To solve above issues we used Data Partitioner is responsible for routing each individual entry to its own Processor host i.e range based partitioning of data which will have its own heap to get local top K elements, {problem is we assumed data is bounded but we have a stream of data}



Since we've streaming data, we need to process it in batches of a few minutes, say 1 min. Now to find topK every minute, we need data in batches of 1 min and calculate topK and then flush out data of non topK stocks since we can't afford it to store it in memory, so now to find topK in 1hr/day we need prev data of non topK hitters which we are flushing out due to memory shortage. So conflicting requirements either keep 1 day data or lose it to afford storage, so we store all the data in disk and use batch processing to find topK, so here Map reduce will be used.
Now to get the solution we need to sacrifice accuracy.



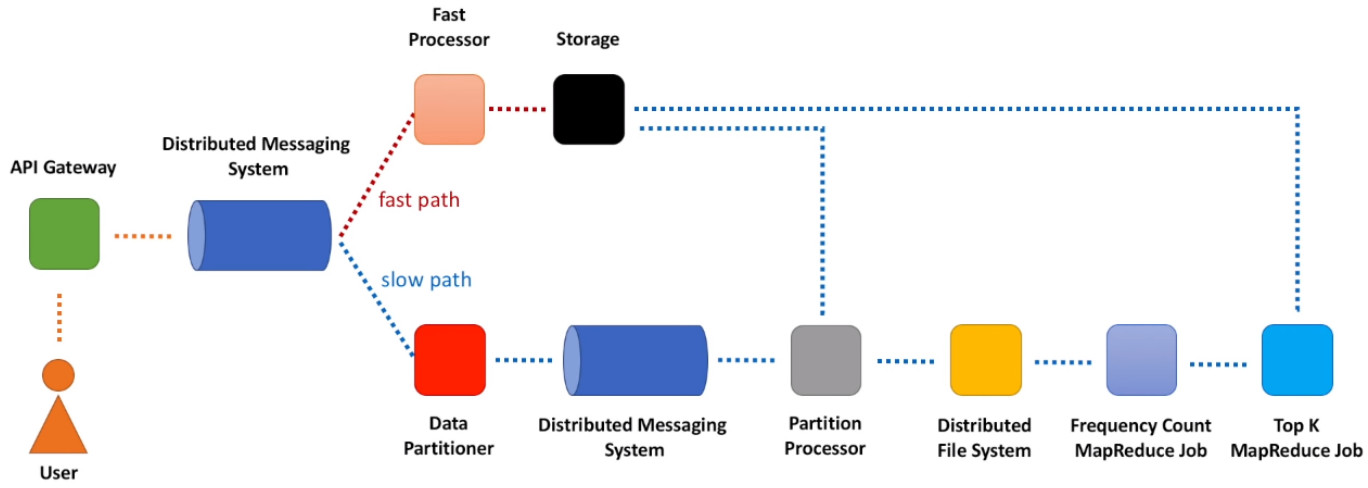
Api gateway → serialize the data into compact binary format to save network IO utilization if req rate is high and now this aggregated data is sent to Kafka. Also, does Authentication, SSL termination, rate limiting, request routing, response caching. In our case we're using API Gateway for log generation (to aggregate data) to know how many times a stock is traded for building freq count hash tables.

Fast Path → calculate topK result approximately and result will be available in seconds.
Slow Path → calculate topK result precisely, result available in minutes.

Fast Processor → Create *count-min sketch* for short time and aggregate data, and since memory required will be less now so no need of data partitioner, it directly process msg from kafka, for single point of failure we need to think about data replication, we can aggregate data in count-min sketch for a min but this would delay result so better to accumulate data for few sec (based on requirement).
Storage → store topK list for some time interval say 1 min

Data Partitioner → It reads batches of events from kafka and parses them into individual events. Partitioner will take each entry and send information to a correspondent partition in another Kafka cluster.

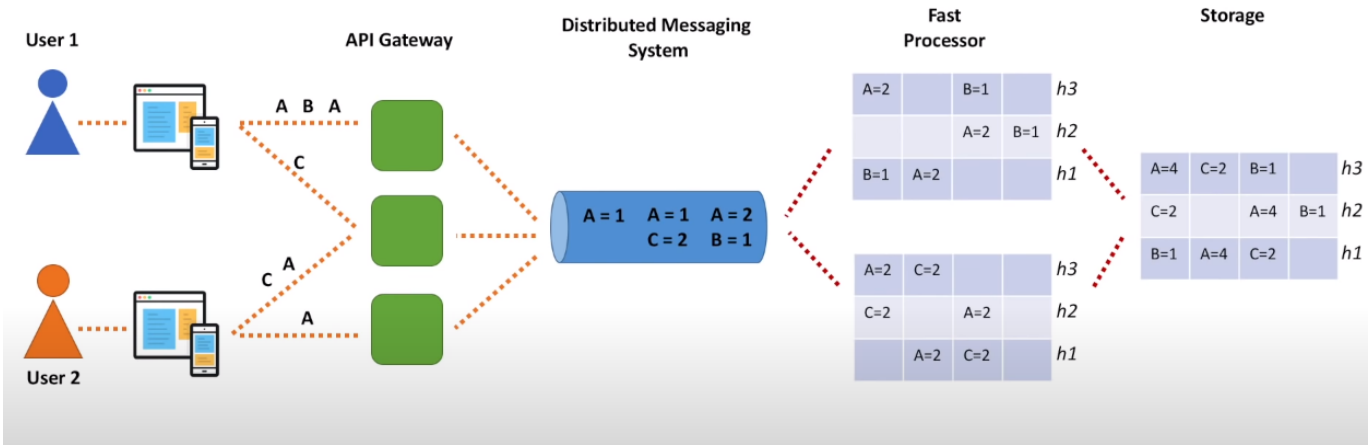
Partition Processor → It will aggregate data in memory over the course of several minutes, batch this information into files of the predefined size and send it to the distributed file system, where it will be further processed by MapReduce jobs. also send aggregated information to the Storage service.



Before sending data to Kafka we can pre-aggregate it on each host (API gateway) maybe for a few sec, which will reduce the number of msg that go to Kafka. Pre-aggregate means we create a hash table and when the size of the hash table reaches some limit we send data out and free the memory for the next hash table.

After a few seconds of data collection by Fast Processor it sends data to storage.

FAST PATH →



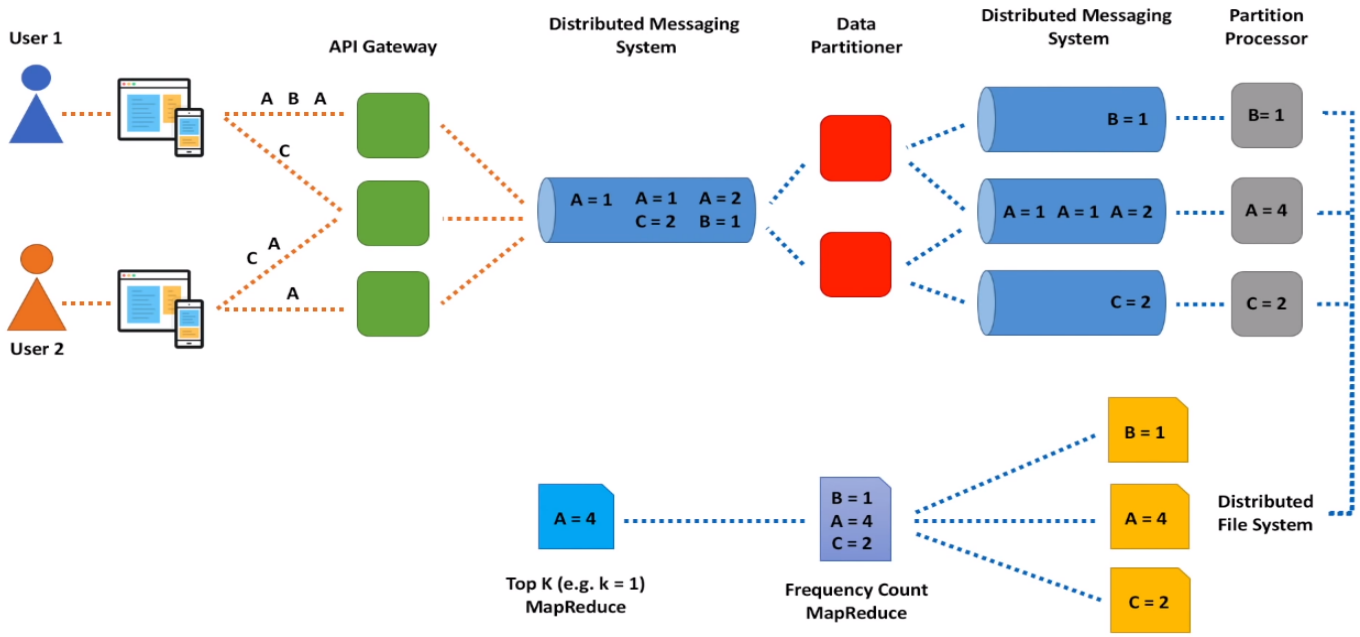
SLOW PATH →

Before sending data to Kafka we can pre-aggregate it on each host (API gateway) maybe for a few sec, which will reduce the number of msg that go to Kafka.

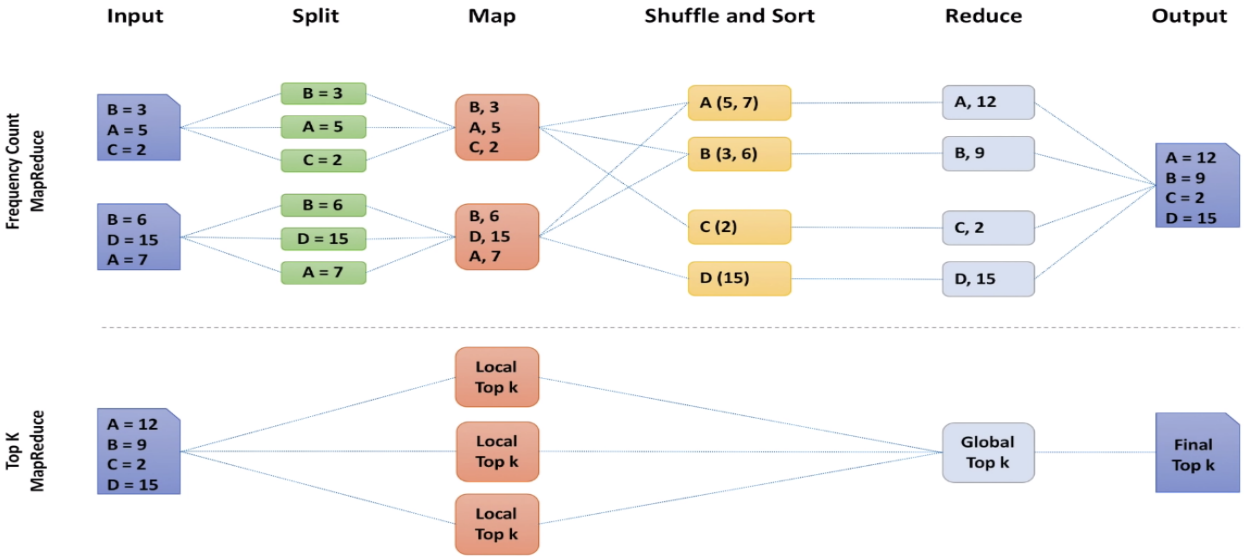
Data Partitioner, a component that reads each message and sends information about each video to its own partition.

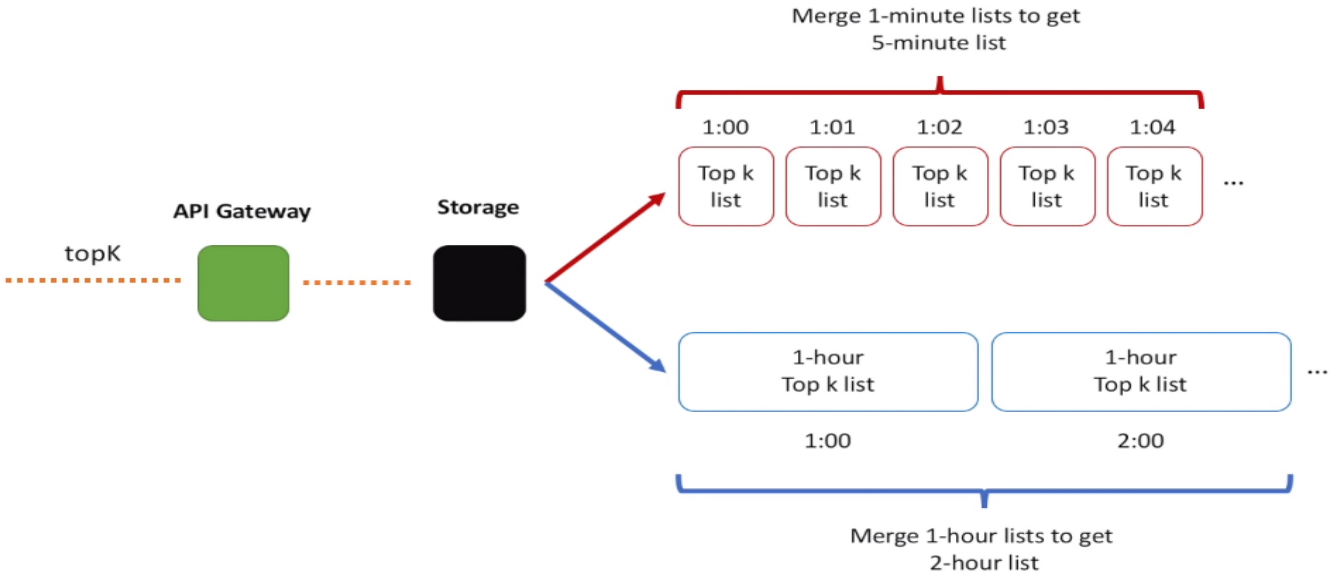
Partition Processors aggregate data for each partition. Each processor accumulates data in memory for let's say 5 minutes and writes this data to files, which are stored in the Distributed File System. Frequency Count MapReduce job reads all such 5-minute files and creates a final list per some time interval.

Top K MapReduce job then uses this information to calculate a list of k heavy hitters for that hour.



MapReduce jobs





Ques Interviewer can ask :-

1. Alternative to count-min sketch algo. - counter-based algo, Lossy counting algo, space saving and sticky sampling.
2. What if the API gateway is unable to aggregate data? → No issues, API gateway will generate log files which will be sended to a separate cluster, so we can run our aggregation on that cluster.

Design Notification Service.

Users want to receive notification for a stock whose price went above/below X% change within Y min.
Code - https://leetcode.com/playground/5W2FydqS

Clarifications :- How many stocks do we need to pull from exchange? Is there any time window for percentage change? How many users/consumers or Traffic? batch-processing or real-time processing ?

Functional Req →

1. CreateTopic(topicName)
2. Publish(topicName,message)
3. Subscribe (topicName,endpoint)

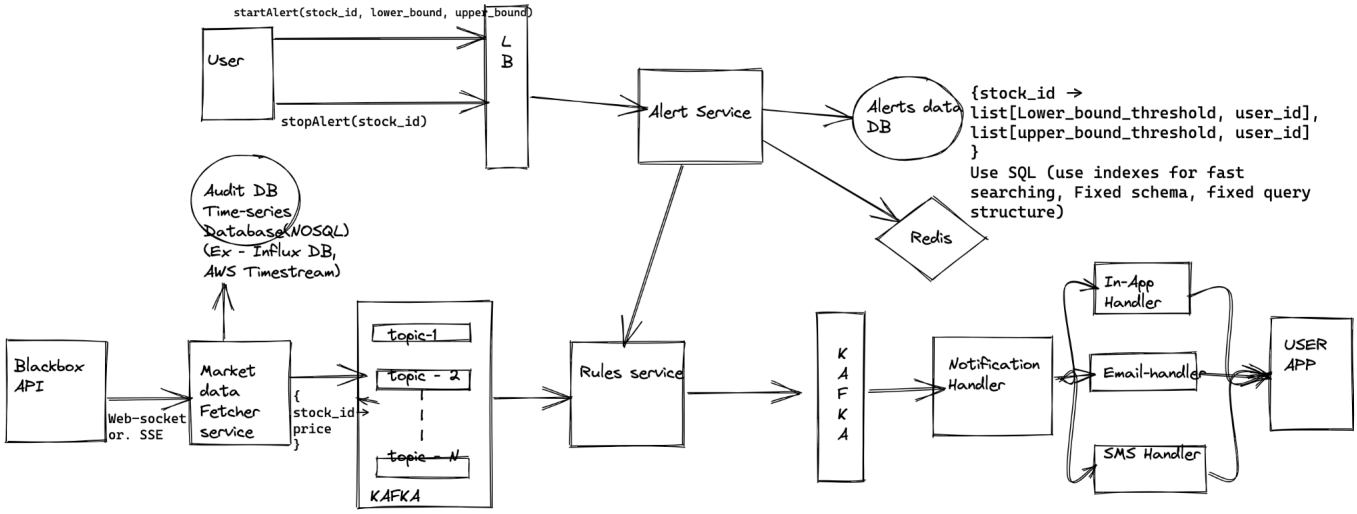
Non-Functional req →

1. scalable (support a big number of topics, publishers and subscribers)
2. highly available and survive hardware failures and network partitions
3. Fast (messages are delivered to subscribers as soon as possible)
4. Durable (messages are not lost and delivered to each subscriber at least once)
5. Security and operational cost

Case -

1. Few stocks + fixed threshold + no time constraint. -> below approach
2. Many stocks + fixed threshold + no time constraint. -> place LB after black-box API + multiple Kafka which will be sharded according to range (A-M + N-Z) + a DB and Redis cache
3. Few stocks + variable threshold
4. Many stocks + variable threshold

Architecture for Apple/Google stock notification :-
https://excalidraw.com/#json=ZwbadY9nZ69Ch9CpaKfSU,N8O2SWYjXSFL1LzWvpY4Xw

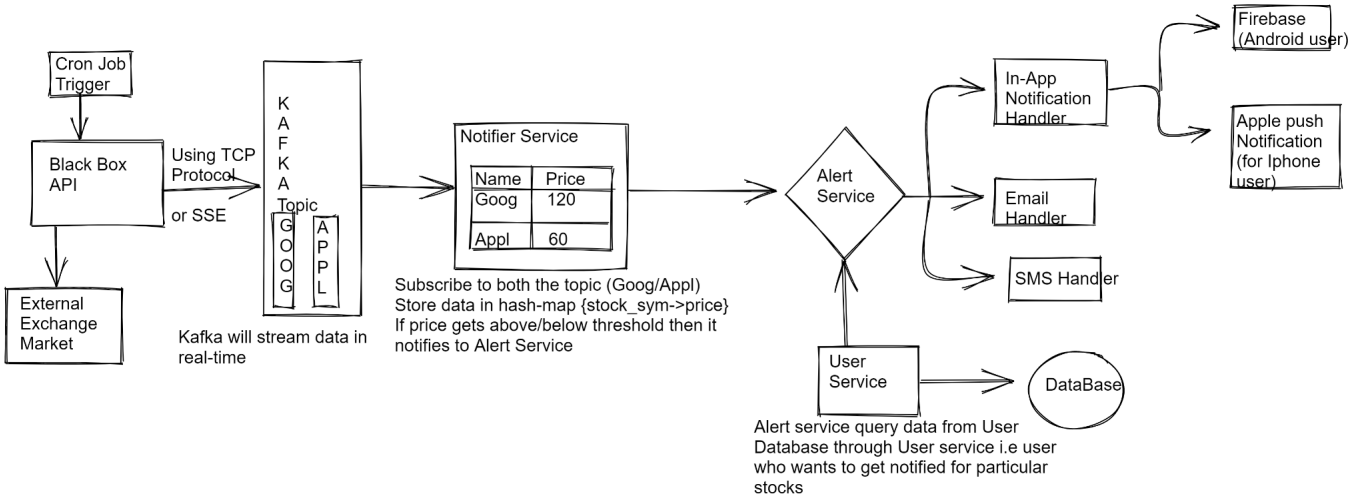


1. Market data Fetcher service - It fetches data from Blackbox API using Web-sockets (which sits on top of TCP) or using SSE and this data get audited in audit DB(Timeseries db like Influx Db or AWS timestream Db) in case if there is some mistake in calculation, so in future we can come and look in audit DB,
2. Kafka_1 - For each stock(apple/google) we will have seperate topic in Kafka.
3. Alert Service - This service has API through which user can create an alert for threshold price, which will be stored in Alert DB and redis cache.
4. Rule Service-Consumer to kafka and receive stock option which are sent by blackbox, also it fetch alerts from alert service and compare the prices, if prices are not under threshold then it takes the user_id and send an event to kafka, which is consumed by notification handler to publish details to user.

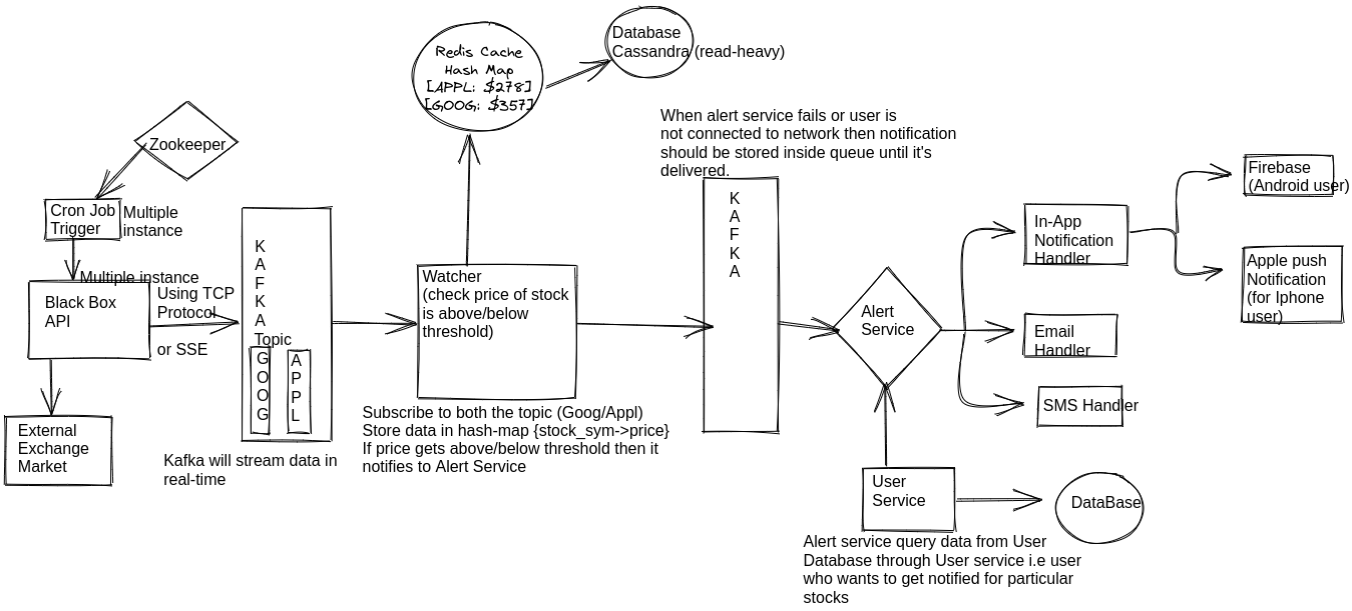
→To optimize comparision of prices from Alert service, we can use 2 list having upper and lower bound, and when a price of stock comes then use binary search and pick up the user from the eligible list, and send {user_id, stock_id, upper/lower bound} to kafka.
→For stopAlert, we can have another table {user_id → {stock_id, pair<lower_bound_iterator,upper_bound_iterator>}} (iterator is the pointer pointing in the list of Alert table), so that search time could reduce and then we can clear data of user from both table.
→In case of if we have Alert system for percentage changes for a stock in fixed time-window, then Alert DB {stock_id → list[%_change, user_id]}, and we'll have N sliding window, which will calculate % change in real time, and rule service will fetch user_id of all those user who crossed the threshold for this changes using binary search in AlertDB and send these ID's to Kafka.

Point of failure :-
1.If the data puller is down, zookeeper could replace it with another one, but we may need a way to let zookeeper know when it is down.
2.If the database is down, we can make it active-passive, failover to secondary database.
3.If the exchange server has the request limit control, you may talk about the rate limiter design a bit, like a sliding window, controlling the outgoing rps.
https://excalidraw.com/#json=9gqKFyngO7KyyMz8Sfdxn,pKXEyBHxbWPpcxVUx5l5xg

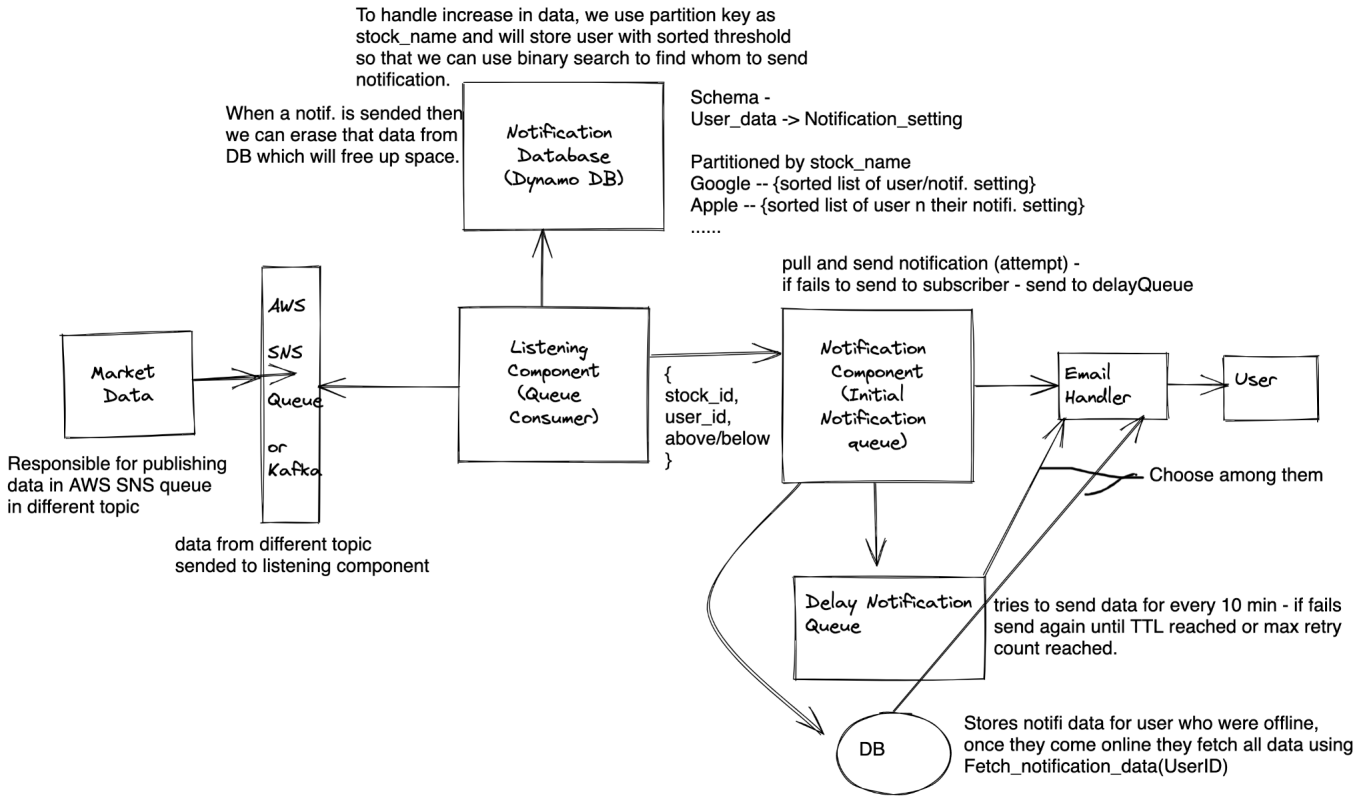
Solu 1



Solu 2



Zookeeper - The backend job component needs to be scaled, one job can periodically query 100 stocks, then we may need 1k such kinds of jobs, those jobs need to have high availability, so we may have redundancy jobs behind the coordinator like a zookeeper.



Partition key – Simply a primary key which DynamoDB uses as an input to an internal hash function which determines the partition in which the item will be stored.
Entities:

User:
- userId:GUID
- notificationSettings:List<NotificationSetting>
- email:Email
- phone:PhoneNumber

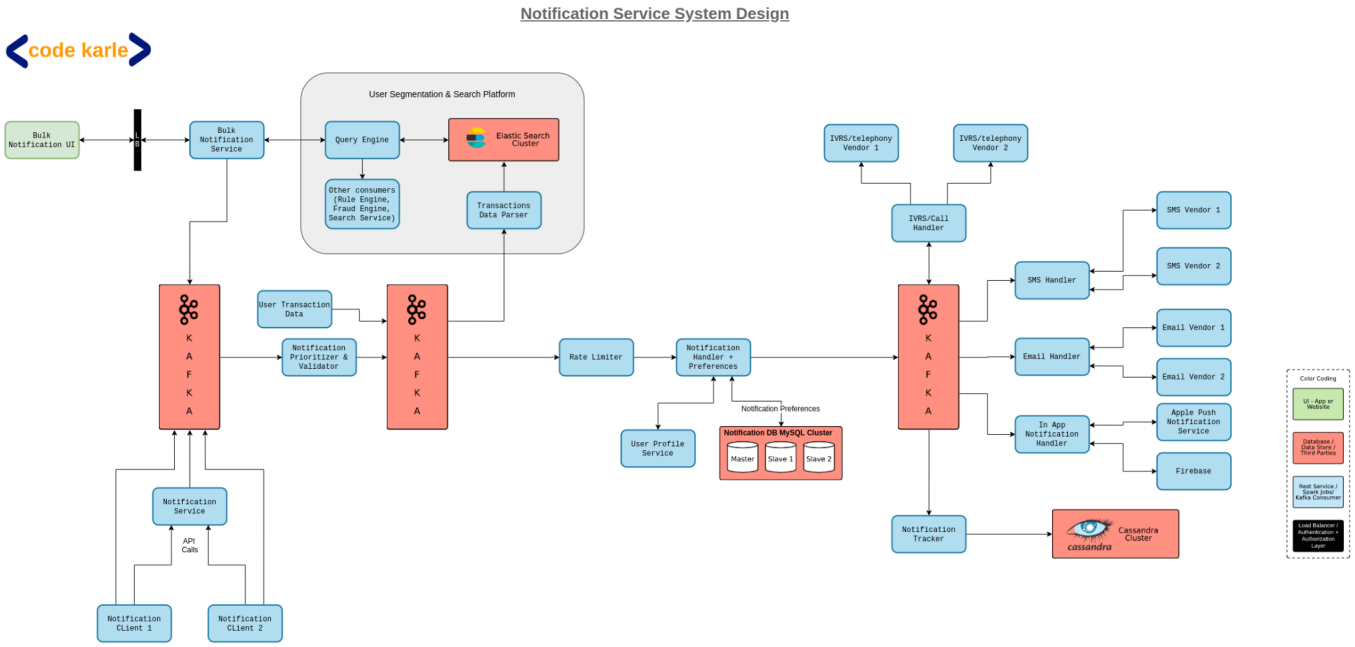
NotificationSetting
- id:GUID
- stockSymbol:StockSymbol
- threshold:Number

API - customer can set notification settings
- addNotificationSetting → add(userId:GUID, notificationSetting:NotificationSetting)
- editNotificationSetting → edit(userId:GUID, notificationSettingID:GUID, notificationSetting:NotificationSetting)
- deleteNotificationSetting → remove(userId:GUID, notificationSettingID:GUID)
-FetchNotificationData → fetch(userId:GUID)

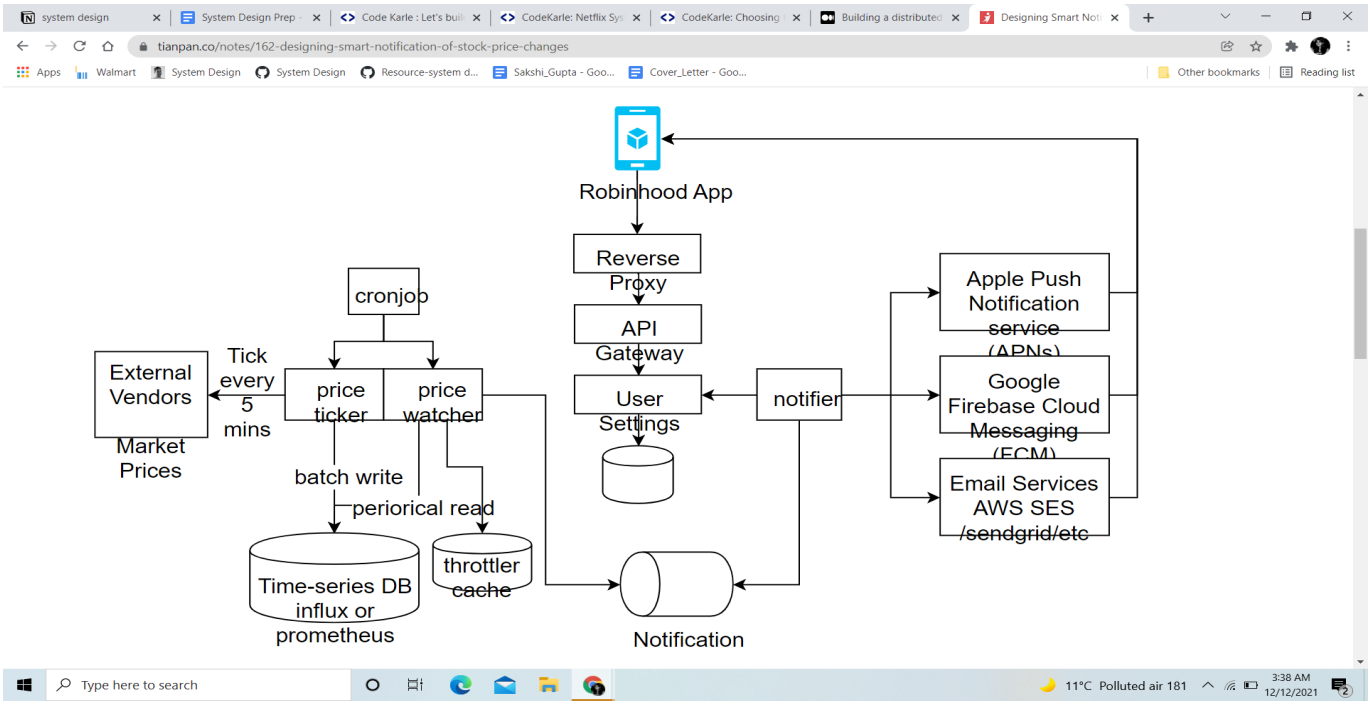
Customer Experience

UI

- Dropdown Stock Symbol
- Number Input field - for threshold
- Button save
- Display a list of their current settings - with edit button / delete button



https://tianpan.co/notes/162-designing-smart-notification-of-stock-price-changes



a

Design Video Streaming Platform

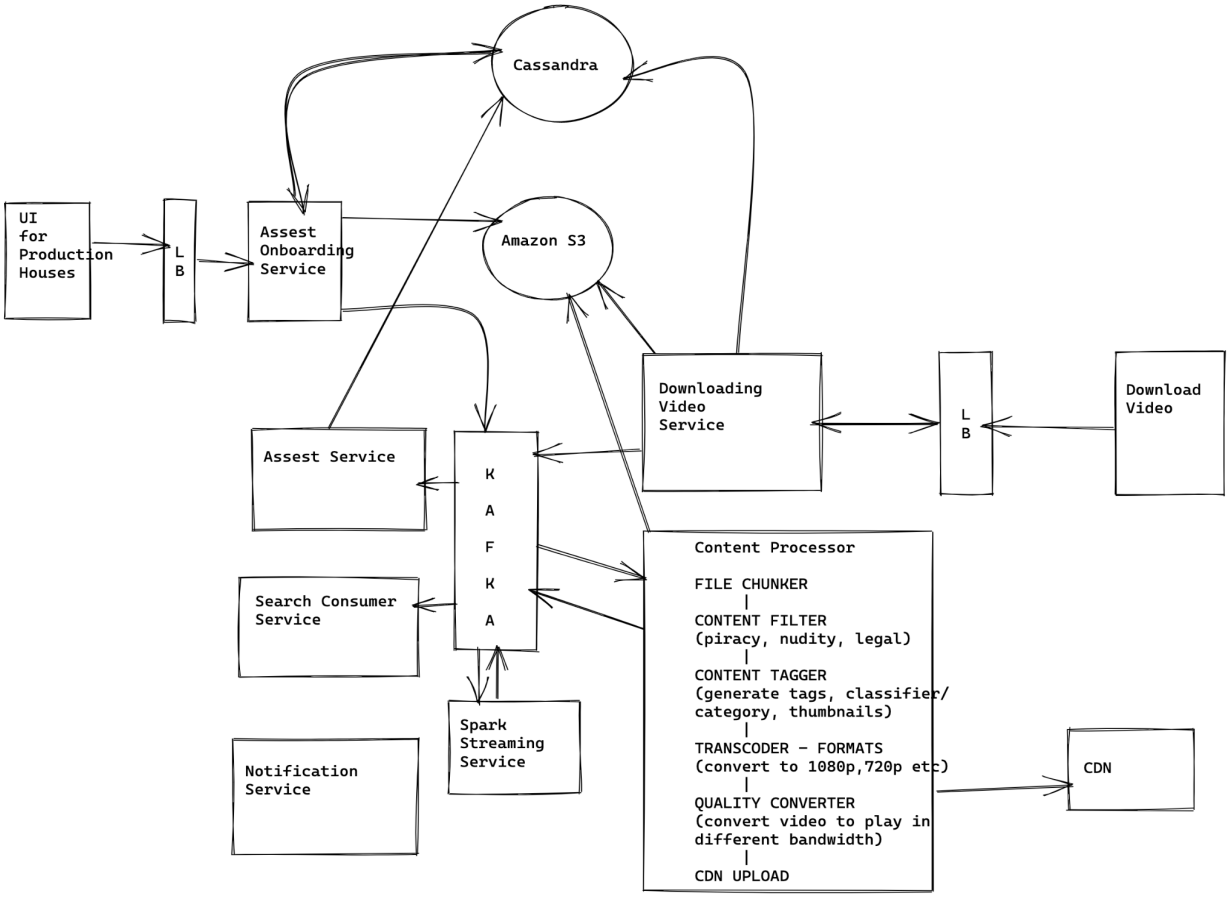
Clarifying ques - Traffic / how many users?How many videos?Can users upload videos?Can users give ratings?, is there any subscription?Do we need to build a streaming service also?

Functional Req :-
1. Download videos
2. Track user activity
3. Search videos
4. Recommend videos
5. Uploading Videos
6. Watch videos

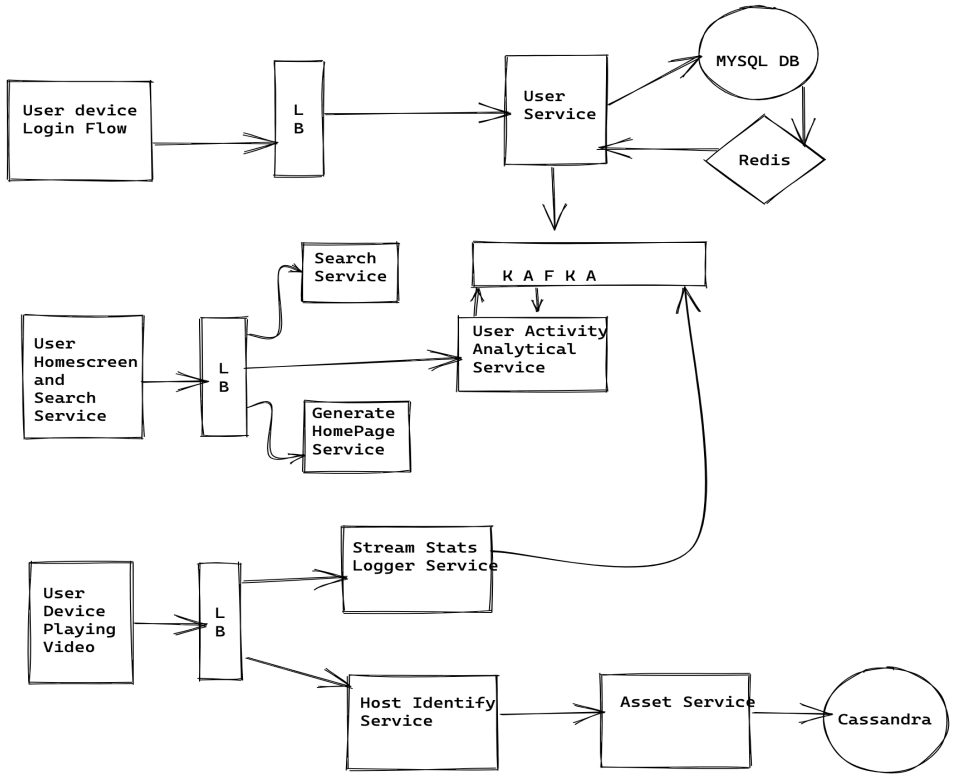
Non-Functional Req :-
1. Highly Available
2. Low latency or buffering
3. Highly performant
4. Highly scalable

API's :-
1. downloadVideo(video_id, user_id, format_type)
2. generateHomePage(user_id)
3. searchVideo(search_query)
4. uploadVideo(video_title, desc, tags, category, video_content)
5. watchVideo(video_id, user_id, fomart_type)

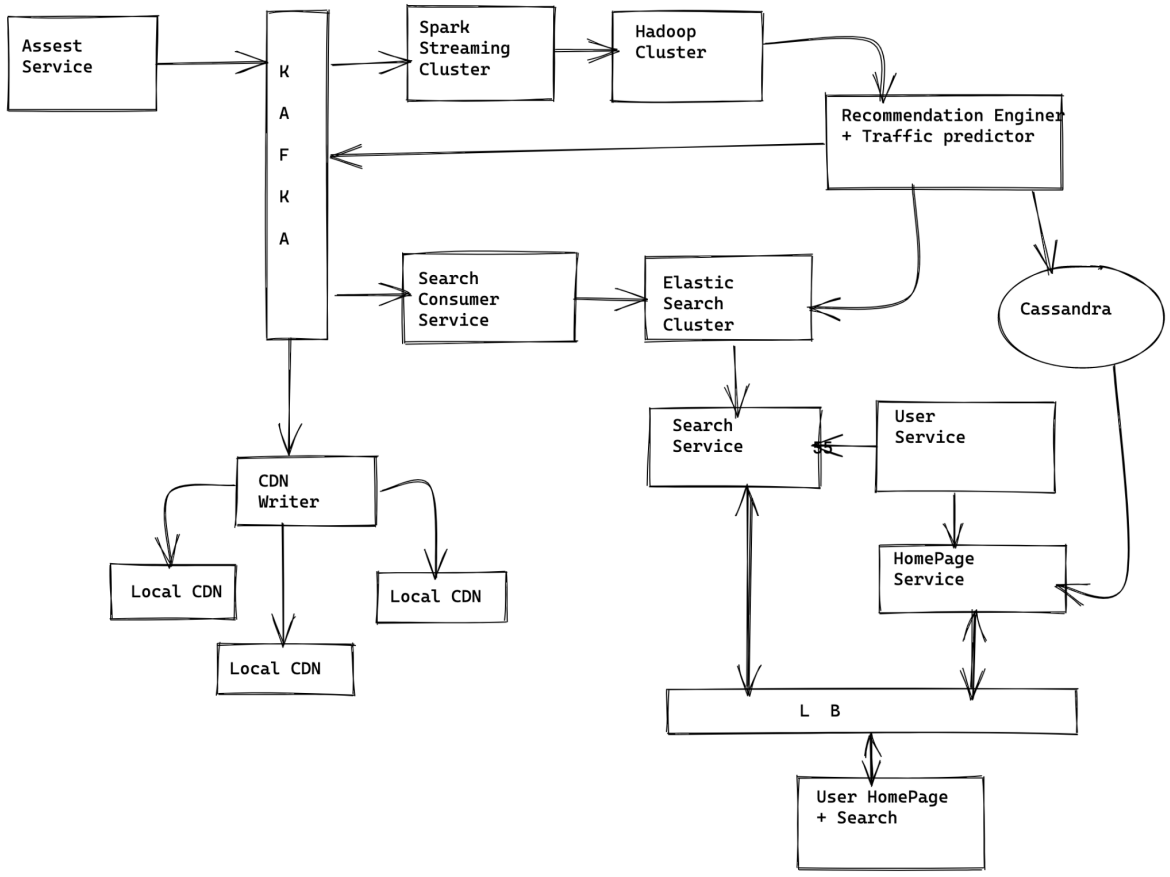
DB schema :-
1. User Table :- user_id, contact_info, subscription_type, list_watched_videos_id, last_access_time
2. Video Table :- video_id, meta_data(size,genere,age_limit), tags/attribute, thumbnail ,video_format/resolution, rating, access_URL, producer, images_attached



1. **UI for production houses -** for short video, a upload button is there, for big video since duration is long hence users need to wait for upload, so instead they give a link of SFTP server along with user_id/pass to Asset onboarding service for verification.
2. **Asset Onboarding Service -** It fetches the content from SFTP url and puts it in its own Amazon S3, and stores all details of video in cassandra. It also sends an event to kafka if the uploading job is complete.
3. **Cassandra -** store all the info/metadata of all videos i.e store video tables.
4. **Content Processor -** File chunker divides videos into chunks either based on size or scenes. Now file chunker puts an event to kafka saying that it's job is done and Content filter now filters those chunks having nudity, piracy or any legal issues, now each chunk goes to content tagger which attach relevant tags to each chunk also it suggest thumbnails for each chunk and put all this info in kafka. Now transcoders convert each chunk to different formats on the basis of quality and each format of a chunk gets converted to multiple formats based on different bandwidth by quality converter. Now all these converted chunks are stored in Amazon S3 and local CDN and an event is sended to kafka saying that the job is done.
5. **Download service -** when a download req comes then it will fetch the Amazon S3 access URL from Cassandra, and it will start downloading video in chunks from Amazon S3. In case of internet failure, it will push an event in kafka storing the info that video is downloaded till this chunk, and when internet comes then it will resume download from where it paused by fetching the event from kafka.
6. **Asset Service -** when the whole upload is complete then asset service will read all the final events from kafka and will store all the info of all top tags/thumbnails or any other metadata into cassandra and will put an event into kafka saying that job is done.
7. **Notification service -** when asset service uploads data then kafka sends an event to notification service to send notification to production house that upload is complete.
8. **Search consumer service -** consumes data from kafka and stores it in an elastic search cluster for generating search results.
9. **Spark streaming service -** All the info related to tag and thumbnail which is aggregated in kafka is sended to spark streaming service which runs machine learning algo to find top 10 appropriate tags/thumbnail for whole video.



1. **User Device Login Flow** - Login request comes from here and goes to LB.
2. **User Service** - used for authentication/authorization of login req, it sits on top of MYSQL DB and is a source of truth of all user info. Also, it sends an event to Kafka for performing analysis on how many users share cred or to find which country has the most subscriptions.
3. **User HomeScreen + Search Service** - Sends a req for generating home timeline and for searching anything.
4. **Search Service** - Gives results for searching something.
5. **Generate Homepage service -** Gives out results that would render as soon as the user opens the home screen i.e generating user timeline.
6. **User activity analytical service -** Keep a track of user search history and put this event in kafka and also analyze user data like user location, user sharing cred etc.
7. **User device playing video -** Video playing req comes from here.
8. **StreamStats logger service -** Track how many min users have watched a particular video, if many users watched 100% video => movie is really good and this data can be used for rating movies, also it tracks which scenes are watched most, this data will be used for deciding duration of chunks. All this data is sended to kafka for further analysis.
9. **Cassandra -** source of truth of info about video and in which CDN the video is stored.
10. **Host Identity Service -** It knows the user location and which video user wants to play, so it query Asset service and from cassandra it fetches location of all CDN where video is stored, and it returns location of 2 CDN, CDN optimized for local views and main CDN (nearest possible may be in other country).



1. **Asset Service -** It pushes an event in kafka when a new video is added.
2. **Spark Streaming Service -** Listen to all info related to tags/thumbnails and dump that info into hadoop for analysis, and it identifies top tags and thumbnail for the whole video and sends it to kafka as an event which is then consumed by asset service and stored in cassandra.
3. **Recommendation engine + traffic predictor -** it consumes info from Hadoop and runs various ML algo to predict recommendation for user and pre-compute all the recommendation for active user and

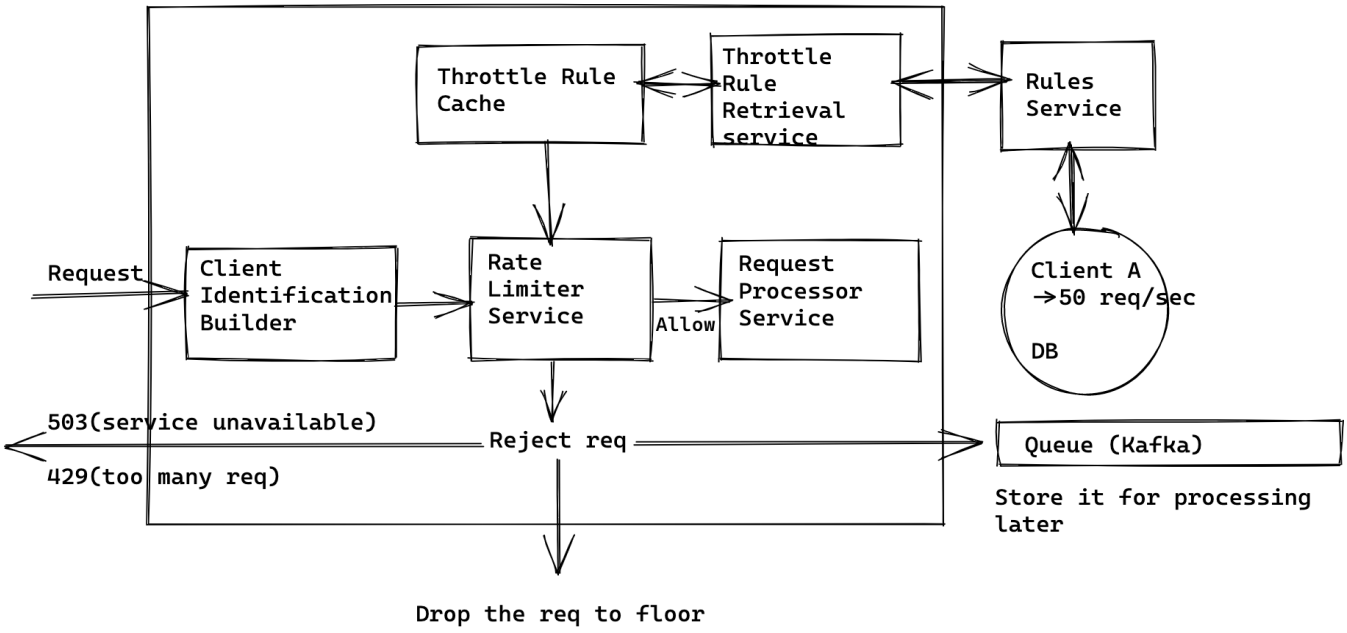
store it in cassandra. It also predicts which video will get more than usual traffic for the coming days. (IT takes into account User search history, user watch history, popular searches, popular watches, other similar user search and watch history)
4. **Cassandra -** we use it becoz it can handle lots of read and write queries, it's good when we have structured query or we have query by partition i.e key-value query like give all videos by producer xyz. It's bad when we want to do a random query.
5. **HomePage Service -** First it'll fetch details of the user from User Service and then from cassandra, it will fetch the recommendation for that user and say if the user age is less than 18, so it also needs to filter out the content whose age is >18. So it also filters the content according to user data.
6. **Search Consumer Service -** When a new video comes, then we need to make it available for the user to search, so it sends the video data (tags/thumbnail,title) in an elastic search cluster.
7. **ElasticSearch Cluster -** It stores search query data and it uses fuzzy search for searching and it also fetches data from recommendation engine to find all the popular search results for the query and then it combines everything and gives results.
8. **Search Service -** when user searches something then search service query elastic search cluster and get result from there and it also fetches data from user service to filter search result according to say user age.

Design Rate Limiting Service
Clarifying ques - do different users have different limits?, does different server have different limits?, does different API's have different limits? How much accuracy is required?Does the rate limiter throttle API requests based on IP, the user ID, or other properties?Is the rate limiter a separate service or should it be implemented in application code?

Functional Req - allowRequest(request)
Non-Functional Req -

1. Low Latency (make decision as soon as possible)
2. Accuracy
3. Scalability
4. Ease of integration with other services

**Note - If due to any failure, our service is not working and we don't know whether to throttle or not then by default we don't throttle.

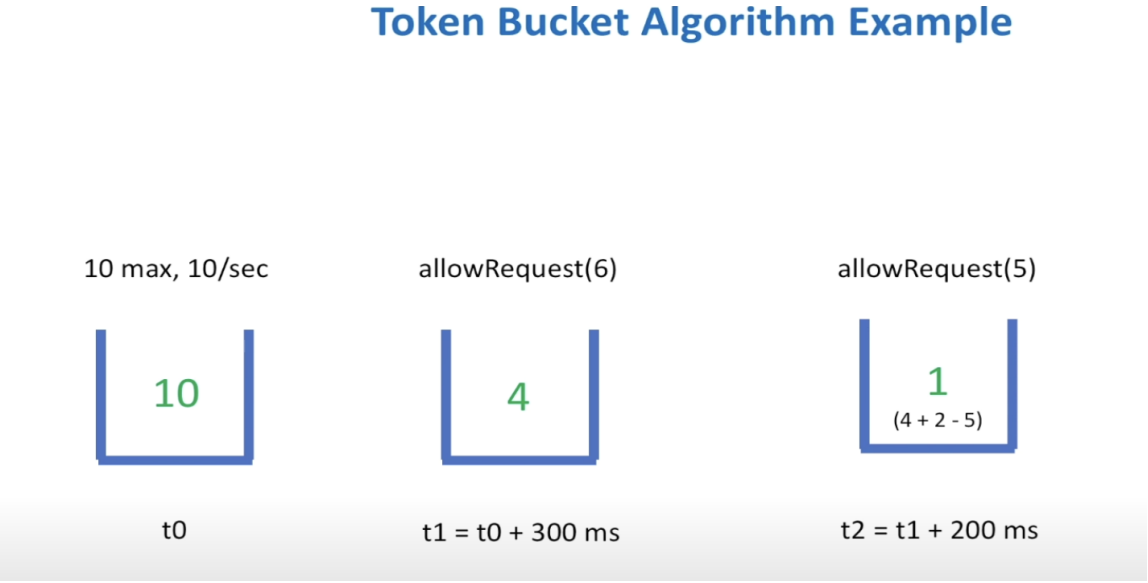


1. **Database -** In DB, we specify rules defined by service owner which states number of req allowed for a particular user.
2. **Rules Service -** A web service that manages all the operations with rules DB,
3. **Throttle Rule Retrieval service -** it's a background process that polls Rules service periodically to check if there are any new or modified rules i.e new or modified no of req for a client.
4. **Throttle Rule cache -** the info which Rule retrieval service pulls is stored in memory in cache, for faster data access.
5. **Client Identification Builder -** When a client logins, this service gets the unique key associated with the client and sends it to Rate limiter service.
6. **Rate Limiter Service -** It takes unique key and find the user in Role cache, if user is found then it check if number of req made so far is less than threshold or not, if yes then it allows req and send it for further processing to **Request Processor,** else the req is rejected, so we have 3 option here, either to drop req, or throw an error or store req in queue for processing later.

Now interview goes in 3 direction :-

1. Rate limiting algo
2. Object oriented design of rate limiter
3. Discuss distributed design and how hosts share data between each other.

1. Rate limiting Algo :-
Code Link - LeetCode



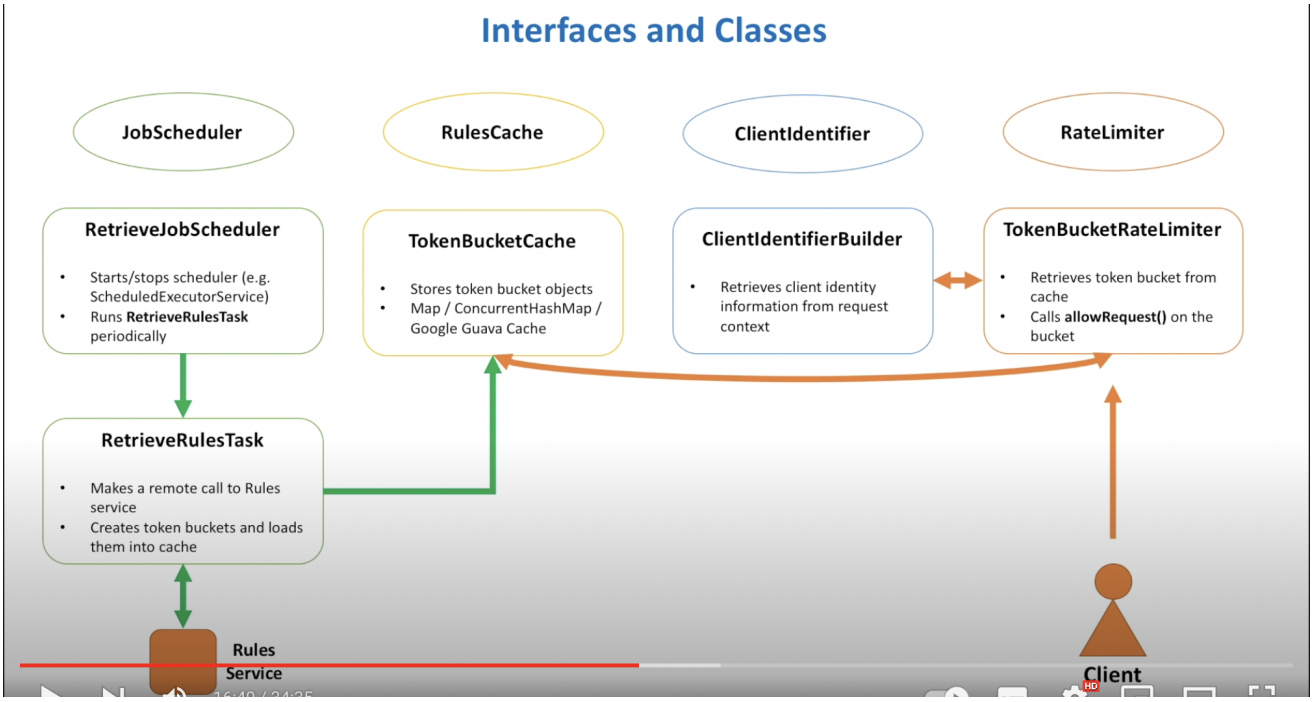
Assume we have a bucket, having capacity of 10 tokens, and rate of refilling bucket is 10 token/sec, now if 6 req comes at t=3ms then 4 tokens will be left, then at t=5ms another 5 req comes, and till now our bucket will also be refilled with 2 tokens (10 token → sec, 2ms → 2 token), so now we have 5 token left.

Algo - When we have a req then we will check if we have tokens available in bucket or not, if we have then we allow the req else not, also bucket has a feature of refilling itself at a constant rate, so that at any point of time we allow at max only 10 token in bucket.

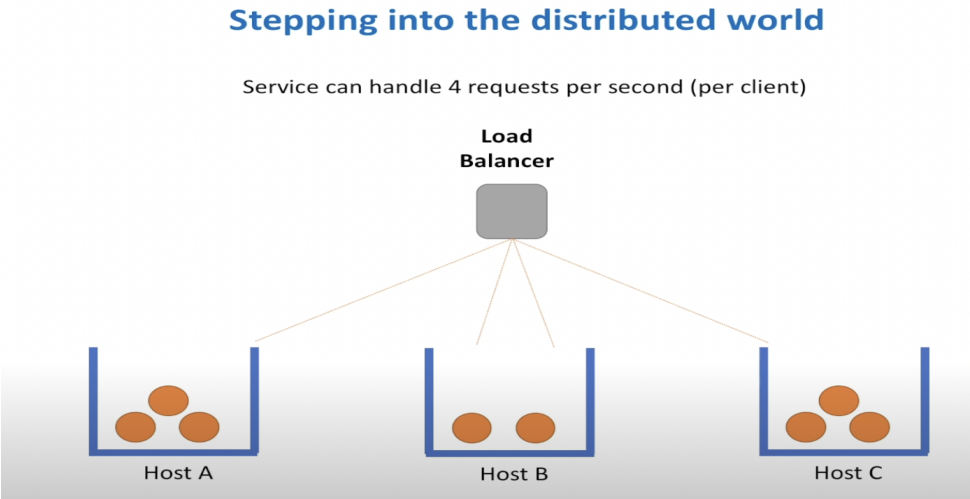
2. Object Oriented design of Rate Limiter
1. JobScheduler Interface - Responsible for scheduling a job that runs every several seconds and retrieving rules from Rules service.
2. RetrieveJobScheduler class - implements JobScheduler interface. It also instantiates, starts and stops the scheduler and runs retrieve rules tasks.
3. RuleCache Interface - Responsible for storing rules in memory.

4. TokenBucketCache - stores token buckets. We can use something simple, for example Map to store buckets.
5. ClientIdentifier - builds a key that uniquely identifies a client.
6. ClientIdentifierBuilder - Responsible for building a key based on user identity information like login or IP address.
7. RateLimiter is responsible for decision making.
8. TokenBucketRateLimiter class - implements RateLimiter interface and it calls the allowed req on the correspondent bucket for that client.
9. RetrieveRulesTask - It retrieves all the rules for this service.

How does it work?
→ Retrieve Job Scheduler runs RetrieveRuleTask which makes remote calls to Rule service, and then it creates token buckets and puts them in TokenBucketCache. When a client request comes to the host RateLimiter first makes a call to ClientIdentifierBuilder to build a unique identifier for the client, and it passes this key to cache and retrieves the bucket and checks if req should be allowed or not.

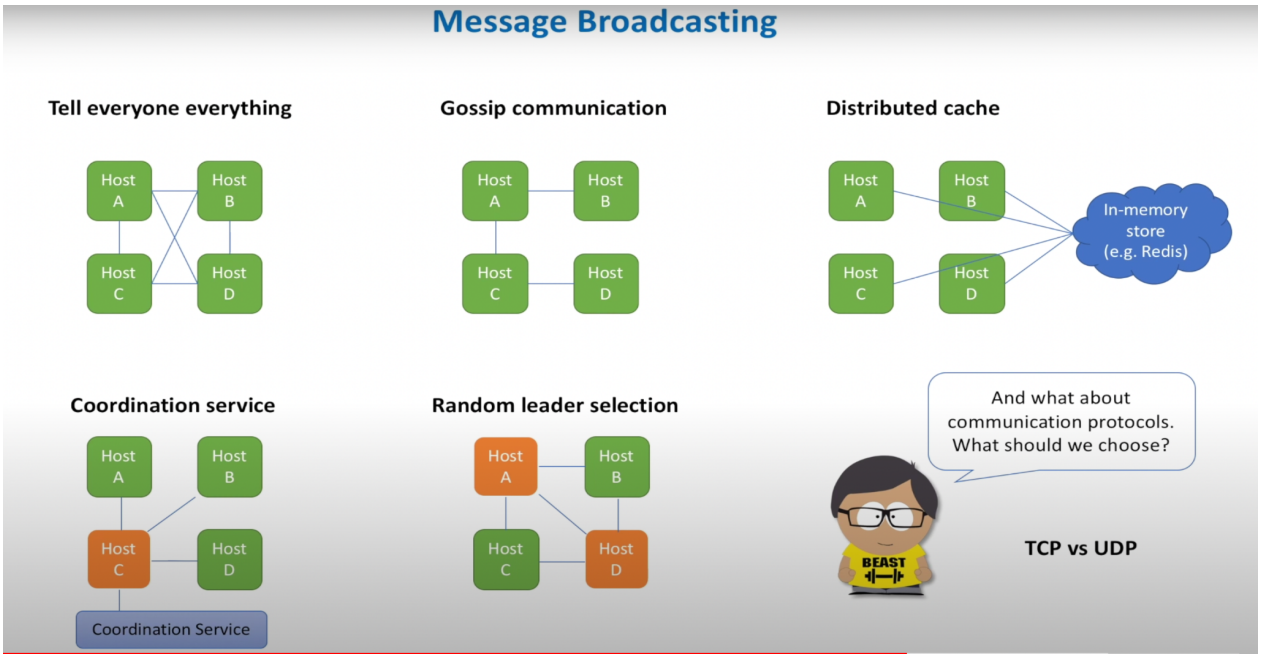


3. Distributed Design for Rate Limiting
Say limit for the req is 4 req/sec and we have 3 servers, now at point of time every host needs to keep track of how many tokens are consumed by another host and need to subtract from its token, so that only 4 req are processed each sec.



In above scene :-
Bucket 1 = 3 - (2 [Host 2] + 1 [Host 3]) = 0
Bucket 2 = 2 - (1 [Host 1] + 1 [Host 3]) = 0
Bucket 3 = 3 - (1 [Host 1] + 2 [Host 2]) = 0
Now, we have 0 token in each bucket and all the req will be rejected until the time limit is reached.

Message broadcasting between each host :-
We've few cases for this :-



Case 1 - Tell everyone everything
This is known as full mesh. In this case every host knows about every other host and shares msg with them. We use a 3-rd party service to keep track of the heartbeat of the host and keep track of the newly added host as well. When a new host is added every other host query 3rd party service to get the list of all connected hosts. But this approach is good in small clusters and it's difficult to scale since lots of msg are broadcasted in clusters.

Case 2 - Using Gossip protocol
Random peer selection is done by each machine and shares info with randomly picked machines.
Cons is data inconsistency

Case 3 - Distributed cache
We've a common cache, in which each host writes and reads data. Benefit is that the cache is very small so faster read and write query also can scale independently.

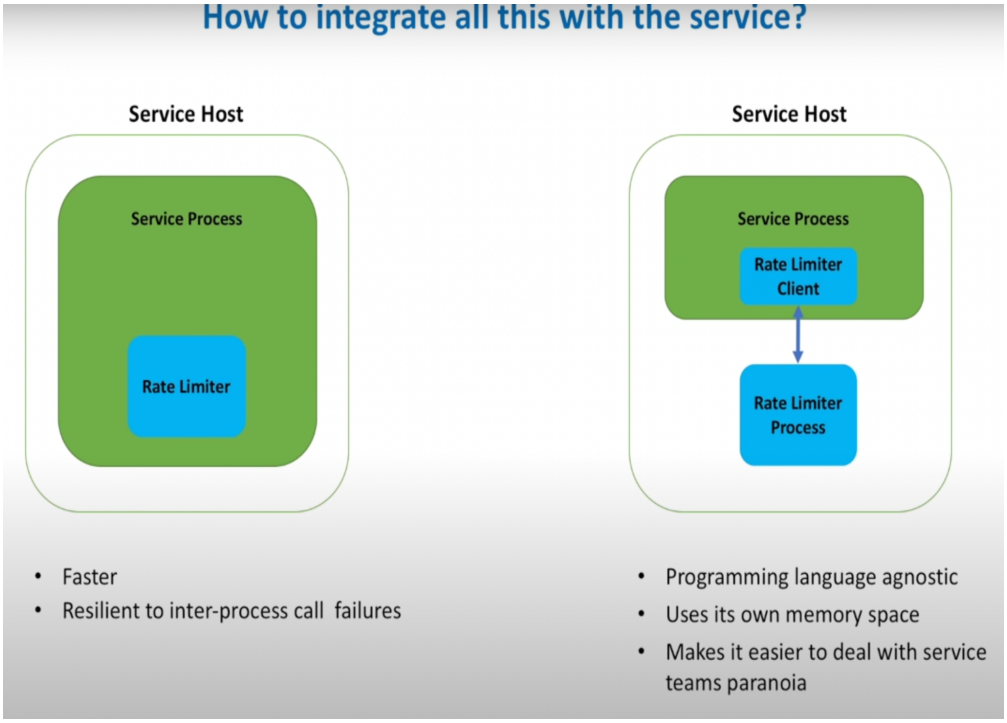
Case 4 - Co-ordination service
We've a coordination service that will choose a leader to whom everyone will share data and read data from. Consensus algo like Paxos and raft are used to implement coordination service.
Drawback is coordination service is a single point of failure and it's difficult to maintain it.

Case 5 - Random leader selection
We can have a simple algo for selecting leaders randomly, the problem is that multiple leaders can get selected, but that won't be an issue if we're not expecting 100% accuracy.

Communication Protocol
TCP - it guarantee delivery of data also guarantee the order of delivery (that it would be same as they were sent)
UDP - it doesn't guarantee that you're getting all the data, also delivery order is not guaranteed, but it's faster than TCP since it throws all the checking stuff out.

If we want more accuracy and can sacrifice a bit of performance → TCP
If we want more performance and can sacrifice accuracy → UDP

How to integrate all this with the service?

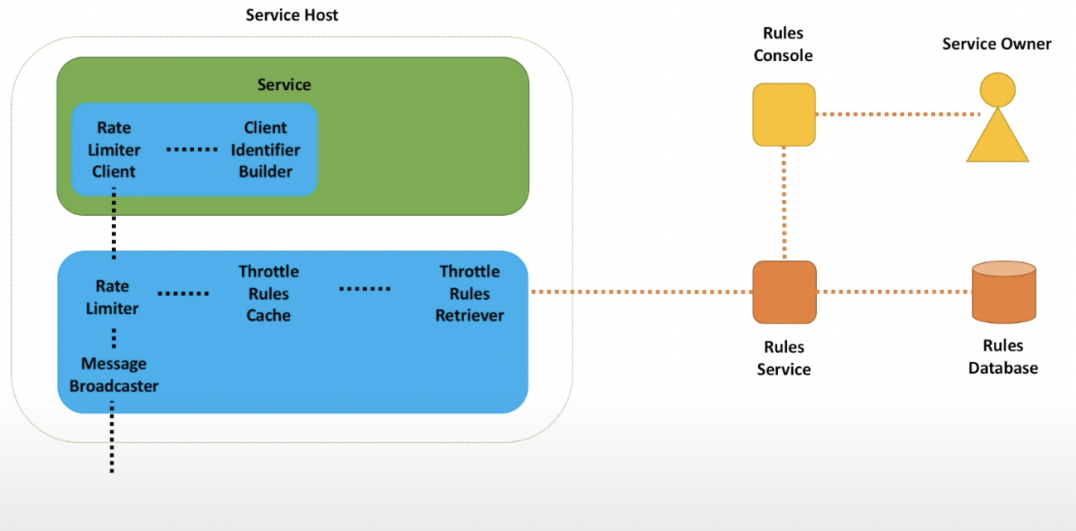


Option 1 - Run Rate limiter as a part of Service process i.e we have 1 library which needs to be integrated with service code.
Pro - 1. Faster since no call for inter-process communication
2. Resilient to inter-process call failure since there are no such calls.
Option 2 - Run Rate Limiter as its own process, i.e we have 2 libraries → the Rate limiter client(integrated with service process) and Rate limiter process (Daemon). Here we'll have interprocess communication between service process and daemon.
Pro - 1. Daemon uses its own memory space, so no need to allocate extra memory
2. Daemon can have different programming language so no need to integrate it with service prices

So which option is better? →option 2 is more popular, it's widely used to implement auto-discovery of service hosts, when hosts in a cluster identify each other.

What happens to requests which are throttled?
Clients may push those in queue for retrying but in a smart way using "**Backoff and jitter",** i.e we retry requests several times but wait a bit longer after every retry attempt, that time period is decided by jitter to spread out the load. If we don't add jitter backoff algo will retry requests after the same interval.

Final Design -
Service Owner - Uses *Rules console* for rule management or storing Rules in DB
Service host - We've *Rules retriever* that store retrieved on *Rules cache,* when req comes then *Rate limiter client* builds client identifier and passes it to the *Rate limite*r for decision making. Rate limiter communicates with other hosts using *Message broadcaster*.



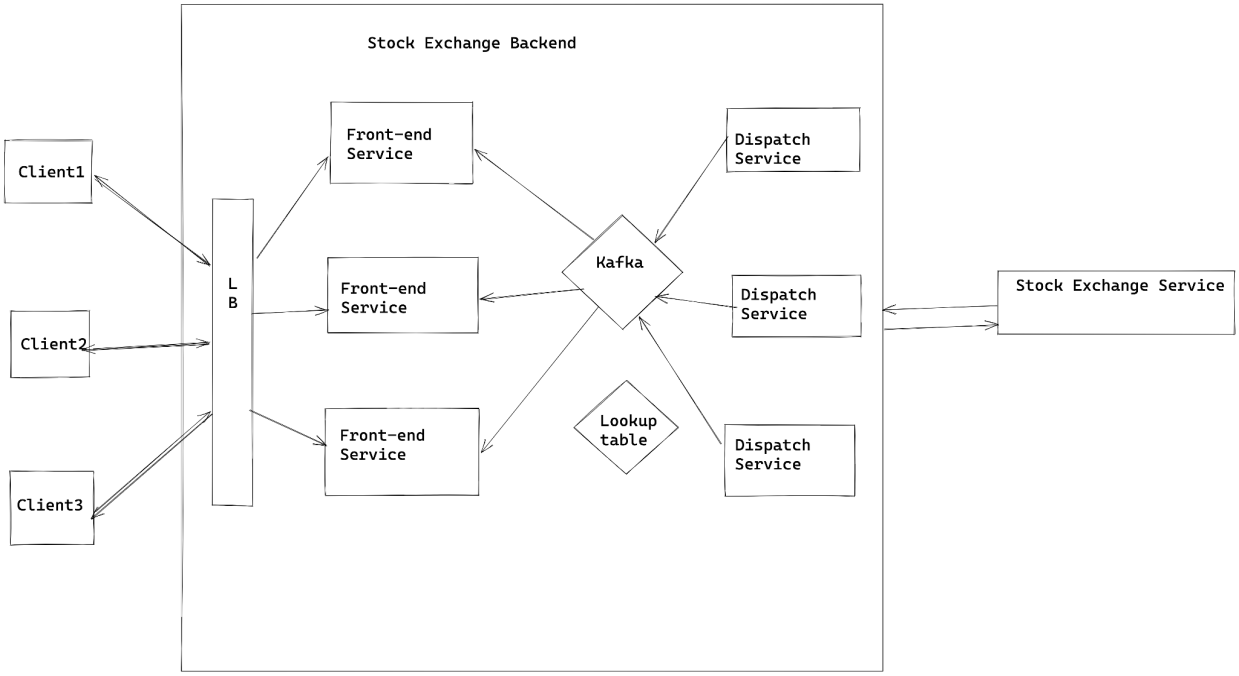
Design Stock Exchange Service (Receive Real-time Stock Price Updates)
Clarification Ques :-
1.Volume of requests the system receives or how many queries can users make in a min/day/month?
2.Real-time or little bit latency is accepted?
3. Does the user need to see the history as well (history of price of stock)?
4. Do all stocks exist on all exchanges? If yes, then which price we will take, will the user give us the exchange_id?
5.Does all the exchanges have the same open/close/holiday times and open hours?
6.Do we need to sort the obtained data by price or volume?
7.Notification feature for stock price up/down?
8.In case of network partition what to choose? Consistency or availability?

Functional Req :- Depends on the answer of above question
Non-Functional Req :-
1.Scalable - as users, stocks, exchanges increase we need our system to scale.
2.Availability - otherwise financial institutes will face major money loss.
3.Performance

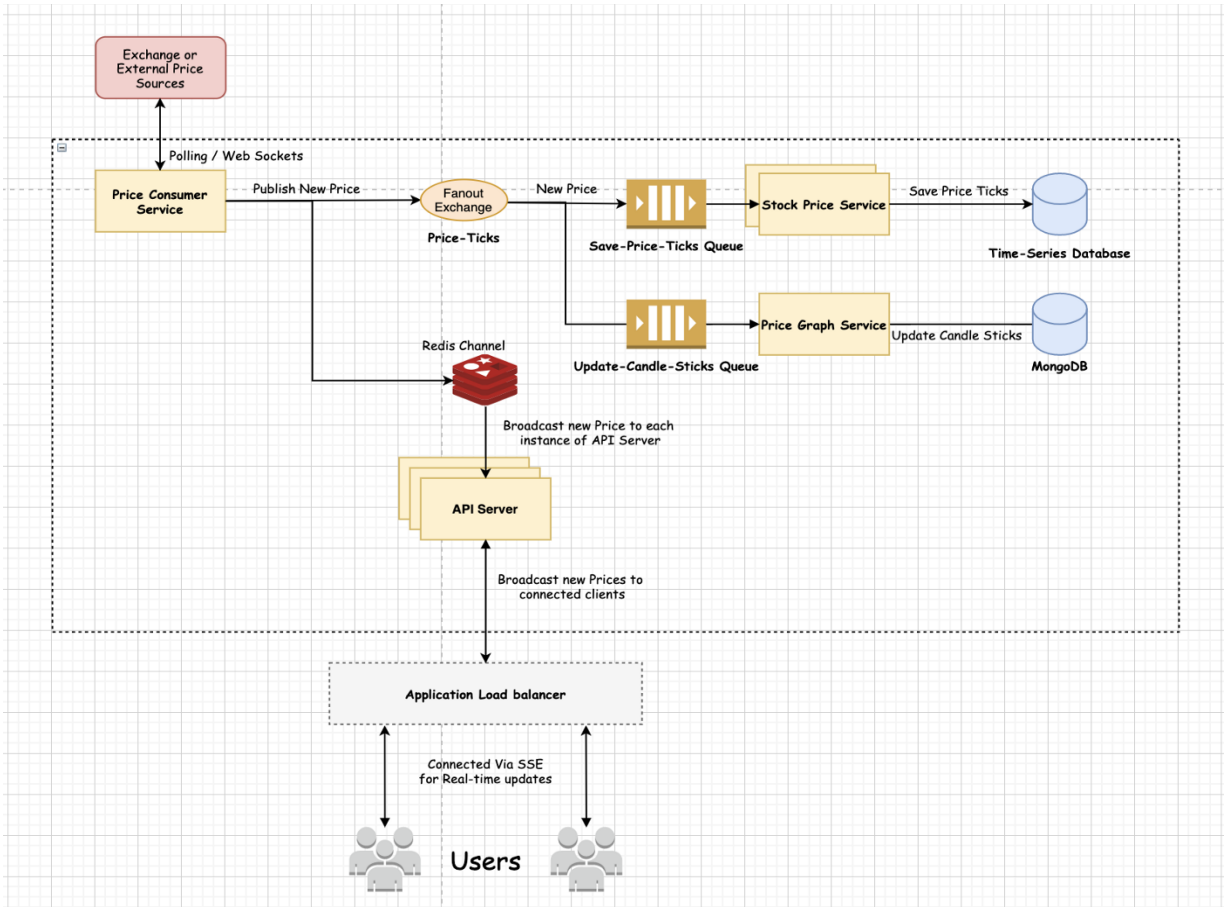
METHOD - 1
This communication can happens by various ways:-
1. TCP connection with custom protocol
2. HTTP polling - Backend continuously sends req to server to receive updates.
3. SSE - When backend establish a connection with server it can mention all the stock whose data it needs, so when there is a update, server sends the data to backend
4. Web-sockets
5. GRPC

The way of communicating is decided by the server, not the backend. We can use either SSE or web-sockets or gRPC

1. **Front-end Service -** holds the client connection upto 100k per machine, so if we have 10M connection then we need 100 machines. Each machine will contain a lookup table storing the info {stock_id → list_client_id}.
2. **Dispatch Service -** for 100 front-end services we need 3-4 dispatch services, and we can distribute 5k stocks among 3-4 dispatch services. Each service will listen to a unique list of stock from the stock server.
3. **Kafka -** we use a multi-consumer queue. Whenever there is change in stock price then dispatch service puts an event in a topic(may be a topic per stock) in kafka, and all the consumer of kafka (front-end service) will consume that event and will check in their lookup table if they need data of that stock or not, if they need then send that data to the users who asked for it.
4. Global cache - Instead of queue we can use a global cache having info {stock_id → list_front_end_service_id}, whenever dispatch service receive update for change in price then it will check who are the consumer for that info in cache and sends info to those Front-end service.



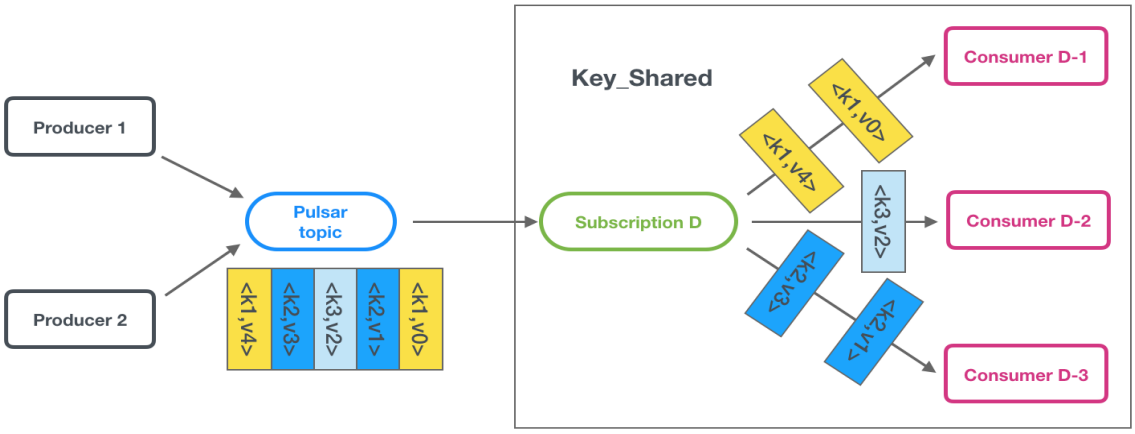
METHOD - 2
Design Real-Time Stock Price Update System

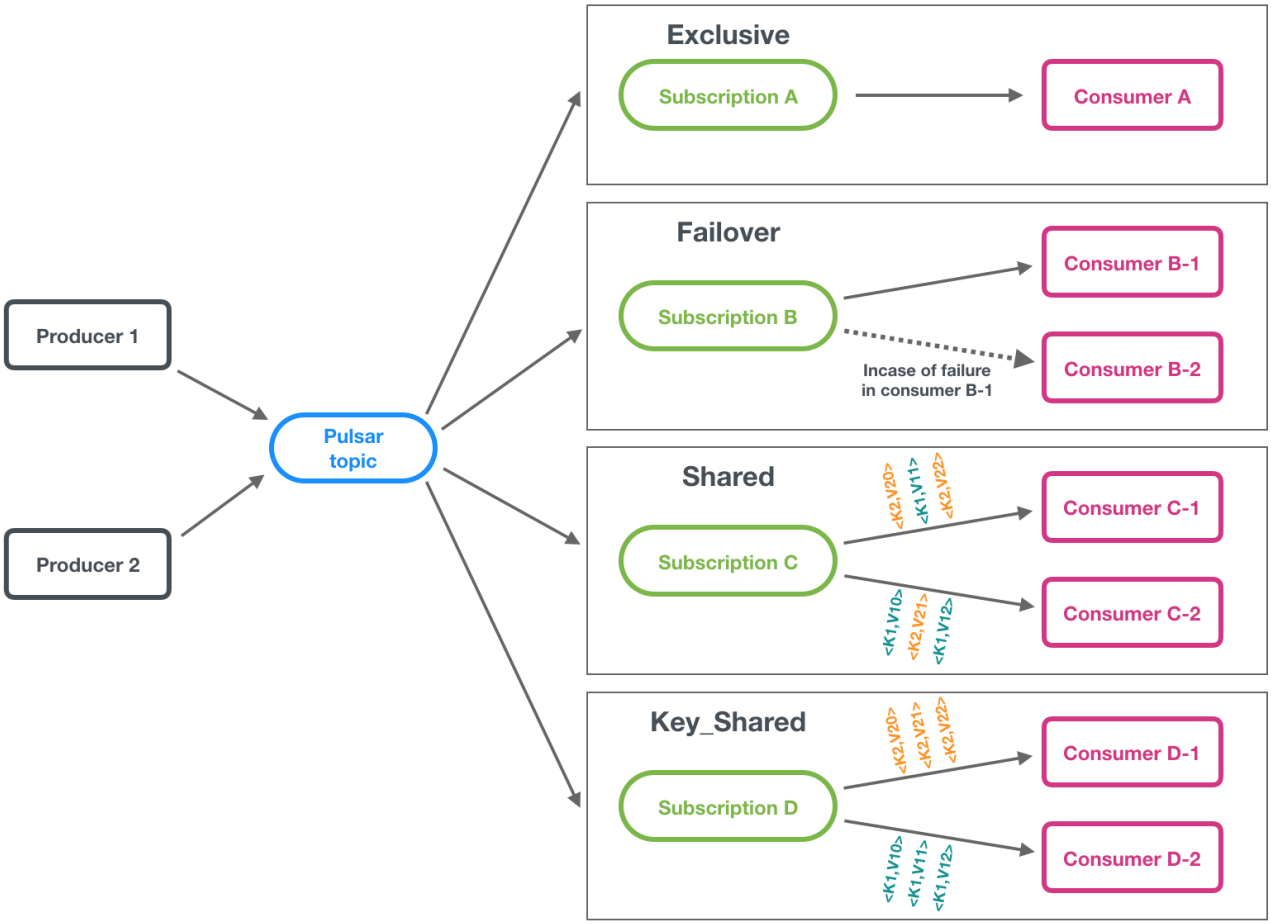


1. Redis Pub/Sub - Since the price ticks are **short-lived** and few ticks loss is tolerable so we use Redis cache, also it provides extremely fast service and in-memory capabilities which helps to send price updates in **minimum latency**. Also, there won't be any objection to sending stale prices to clients therefore persistence of data is not needed.
Cons - It doesn't guarantee the order of processing of msg or order of delivery of msg to subs since it works on a broadcasting model. So we used Kafka for updating the candlesticks.
2. **External Price Source** - It provides us with a new price of stocks and sends it via SSE(Push Mechanism) or web-sockets or we could fetch prices from them at regular intervals (Pull Mechanism).
3.**Price Consumer Service -** Consumes updated price of stock from external sources via SSE. It also adds currency conversion features or validation on price ticks before allowing them in the system. It then publishes new prices to Redis and Fanout Exchange.
We can either run one instance of price-consumer service for each stock, or combine multiple stock for one instance, depending on use case.
4.**API Server -** It provide REST APIs to client and also event stream API (SSE). Each instance of this service subscribe to Redis for price update. When Redis gets updated each api-server instance gets notified about updates via events, then it pushes the update to the client via SSE.
5.**Stock Price Service -** It consumes events from **save-price-ticks message queue**(added for data persistence in case of failure) and saves them in time-series DB like Timescale or InfluxDB for analytics.
6.**Price Graph Service -** It consumes events from **update-candle-sticks message queue**(added for data persistence in case of failure and to ensure correct ordering of which msg are processed) and update different candles in cache.

METHOD - 3
Design a pub/sub architecture for real-time stock price
We can consider **Apache Pulsar's Key_shared subscription** to pre-sort the data by ticker symbol. Each publisher could then publish their results to a ticker symbol specific topic. Clients would then need

to subscribe to only those topics that they are interested in and consume a subset of that data.
All ticker symbols ----> (500 symbol-specific topics). <---- Client subscribes to a subset of these.
Apache Pulsar - It's a distributed, open source pub-sub messaging and streaming platform for real-time workloads, managing hundreds of billions of events per day



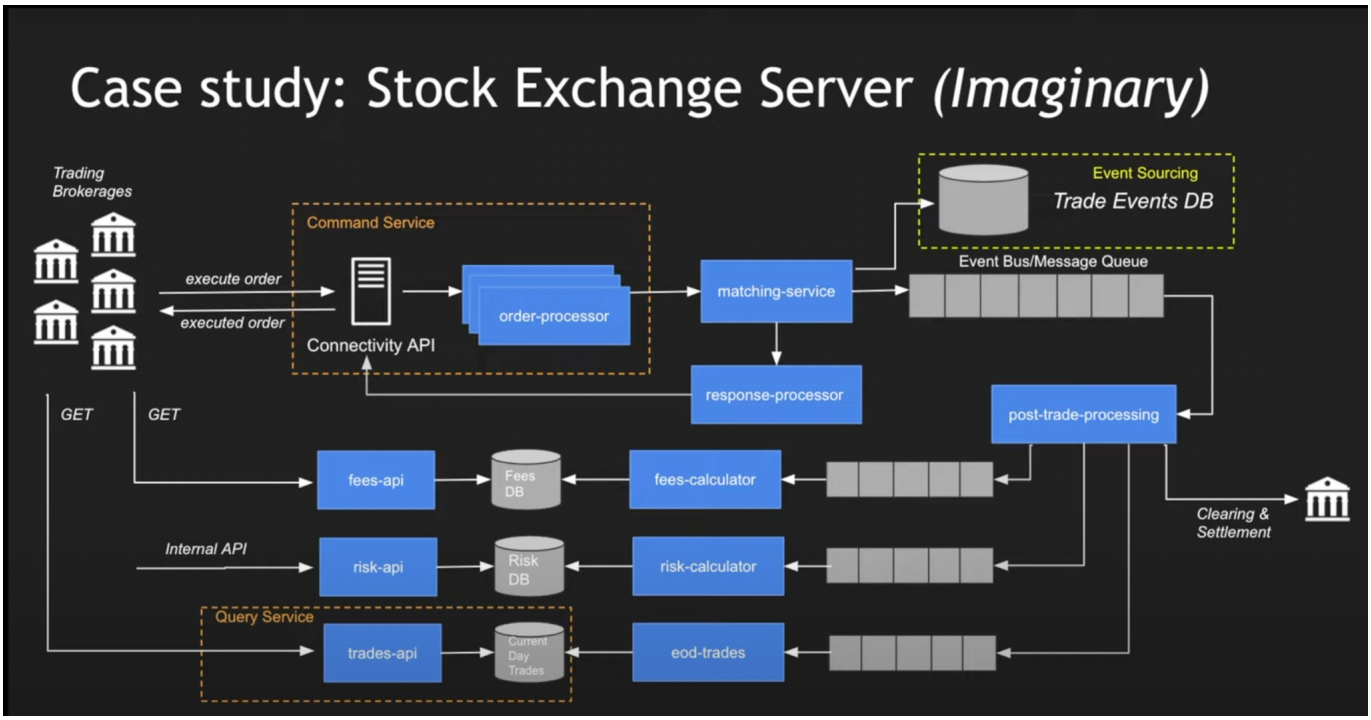


Pulsar Subscription type :-
1. Exclusive Subscription - Only one consumer can subscribe to a topic, if multiple consumers subscribe to a topic using the same subscription then an error is thrown.
2. Failover subscription - Multiple consumers can get attached to the same subscription, and a master consumer is picked which receives all msg, when master consumer disconnects then msg is delivered to the next consumer in line.
3. Shared Subscription - Multiple consumer can get attached to same subscription, and msg are

delivered in round robin manner (a msg is delivered to only one user), when a consumer disconnects then all the msg which were sent to it but not acknowledged are rescheduled for sending to other consumer.
4. Key_shared subscription - Multiple consumers can get attached to the same subscription. Msg with the same key or same ordering are delivered to only one consumer who subscribed for it, no matter how many times msg is re-delivered, it's delivered to the same consumer.

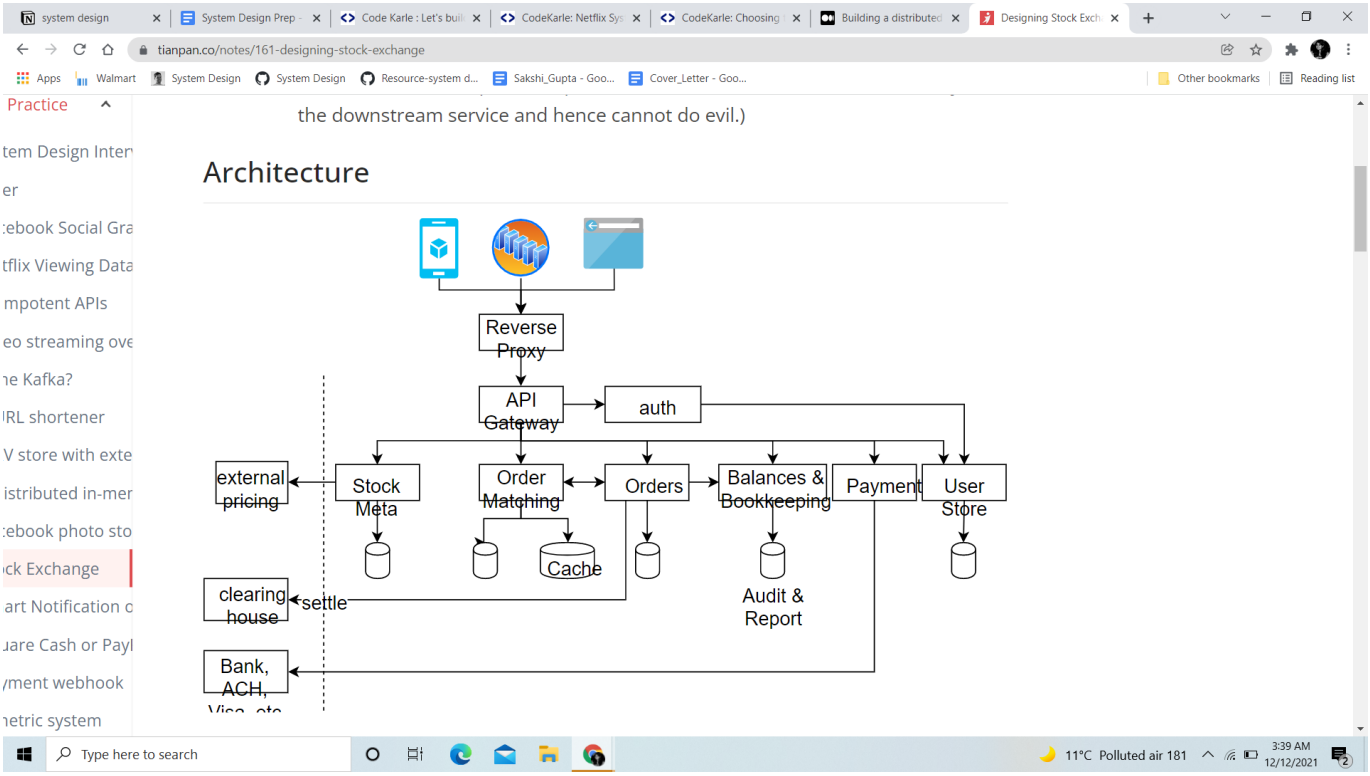
Kafka vs Pulsar :- Pulsar is used when we require both queuing and event streaming in the same system and Kafka used for event streaming use cases that require high throughput, scalability, and permanent message storage.

METHOD - 4
Designing Stock Exchange Service



There are many trading brokerages which try to execute orders using connectivity API into the stock exchange system.
1. **Order Processor -** Fetches data of stock order for processing from Trading brokers using connectivity API.
2. **Matching service -** Orders from different brokerages are sent to matching servers, which runs a few algo to match the buyers and sellers and make an execution for them. This service will store the executed buy and sell orders in Trade Event DB. Also it sends a copy to the response processor and it also publishes the message into a Kafka so that it can be processed by some post trade processing service.
3. **Trade Event DB -** The executed buy or sell orders are stored in this DB for further processing.
4. Response processor - A copy of response from matching service regarding matching of buy or sell orders is sended to response processor which is sended to trading brokerage via connectivity API.
5. **Post trade processing -** It pushes msg into different queues which will do different calculation and analysis.Also, it sends msg to clearing and settlement platforms.
6. **Fees calculator -** It calculates fee based on the trades (GST, SGST, taxes) or how much fee does the user needs to pay to broker and store this fee info into Fee DB which can be accessed by brokers using fee-apis

7. **Risk calculator -** It calculates risk associated with the trade/risk with the system and stores this info in Risk DB and an internal API is exposed for performing analysis.
8. **End trades -** We get all the trades of a day stored in current day trade DB, and it will get refreshed each day. Also brokers can access this data through a get API.



Source of below pic - [Designing Stock Exchange]

Design Logging System.
As we know Microservice runs on multiple hosts and to fulfill business requirement these services need to talk to multiple services running on different machines so the logs generated by micro-services are distributed across multiple hosts so while troubleshooting any issue we need to keep a track of logs from different microservice and to handle situations when a request originated from S1 and goes to S2 and then there comes an error at S2 so we should be able to tracback its origin and that's why we need a central logging capability. Hence logging right info is essential to build meaningful metrics and monitor the health and performance of the system.

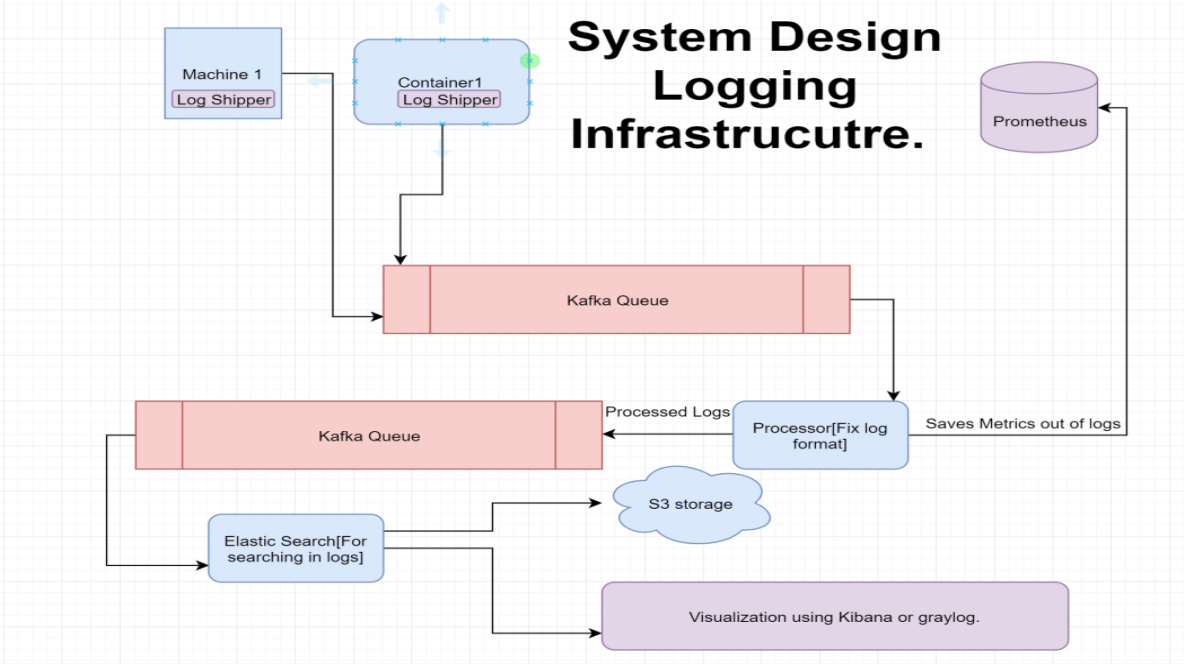
Log shipper - Logstash, Flume, Fluentd
Log storage - Old data - S3, Amazon Glacier / few days or month - Cassandra, MongoDB, HDFS / few hours - Redis
Log Analysis - Kibana or grafana
Alerting service - Sentry or Honeybadger

METHOD-1
3 architectural problem in designing logging system :-

- How to ship logs from the machine - a log shipper (e.g. Logstash(written in Ruby),Flume (written in Java),Fluentd(written in Ruby),rsyslog) which reads the log from files/directories/containers and ships them to Kafka.
- How to process and save it - log processing should be async since we can't process them in real-time so we need to store them and process later. For processing logs are pushed into Kafka and then for any algorithmic processing/analyzing/transformation of data they are pushed to

processor, then it either push log data to Amazon S3(AWS) for auditing/future use or pushes it to next Kafka

- Show or visualize or analyze it - To present logs into human readable format we use Kibana or graylog to visualize it and ElasticSearch for searching logs according to correlationId.



METHOD-2

Requirements

- Centralized logging capability: means all logs should be store at a single place, purpose is to make log info available easily at one place (can be AWS)
- Centralized tracing: means that there should be a single place to traceback any error
- Generate aggregate report, like how many times in past 24Hr or past 1 week 404/ 503 error has occurred
- Send alerts to the oncall team when a particular error happens, maybe use a chatbot or send a mail/ phone call

Corner Cases

- Since it's a distributed architecture, it could be possible that a single service like User Registration service (S1) will be having multiple instances running and which are discoverable by a load balancer, so we should be able to identify that error has happened at which instance.

Log Data Structure

For troubleshooting purposes log data is converted to JSON to make log msg more readable, can be done using Logstash aggregation tool and inside Logstash there are encoders you can configure to output JSON log msg.

Correlation IDs

Generate a correlation ID (using Spring Sleuth + Zipkin if microservice is developed using Spring Cloud) when the first microservice is called then pass the same ID to downstream services. Log the correlation ID across all microservice calls so that you can use the correlation ID coming from the response to trace out the logs.

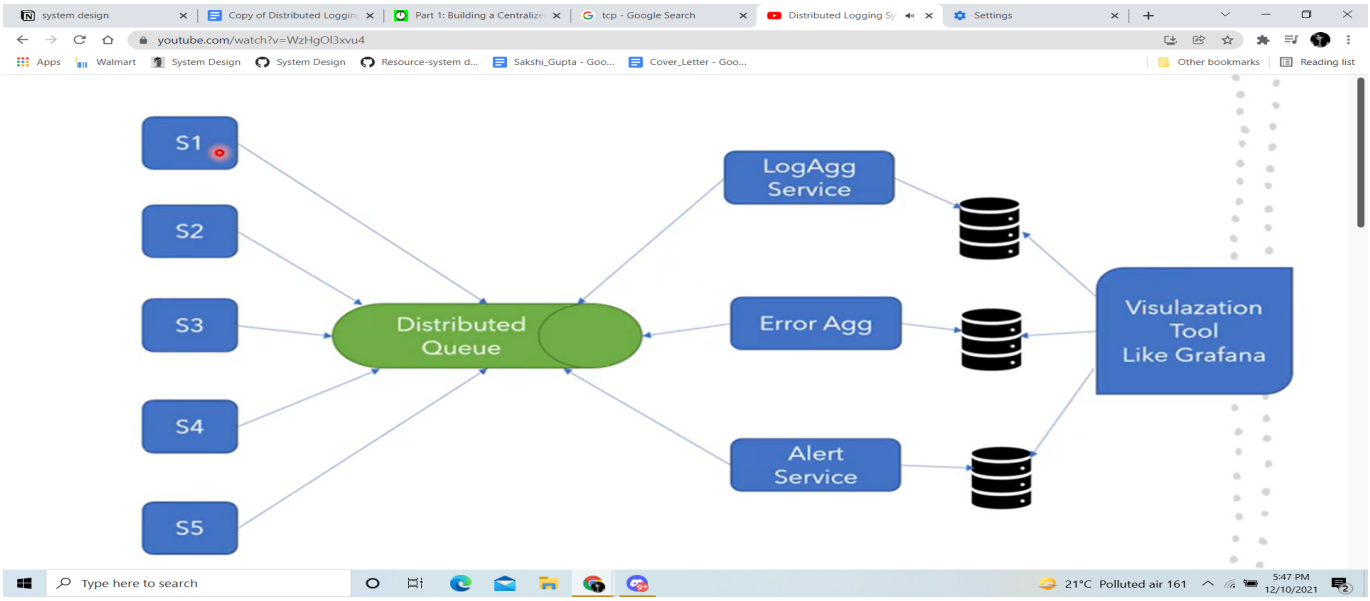
What data to log?

Store the corelationID through which we can resolve the history of that request (request context) and also store some other metadata from that request such as what kind of request it was like if it's a HTTP

request so whether it's a GET/ POST or DELETE. But **don't** log any sensitive information because it may lead to PII (Personally Identifiable Information) issue as a security concern.

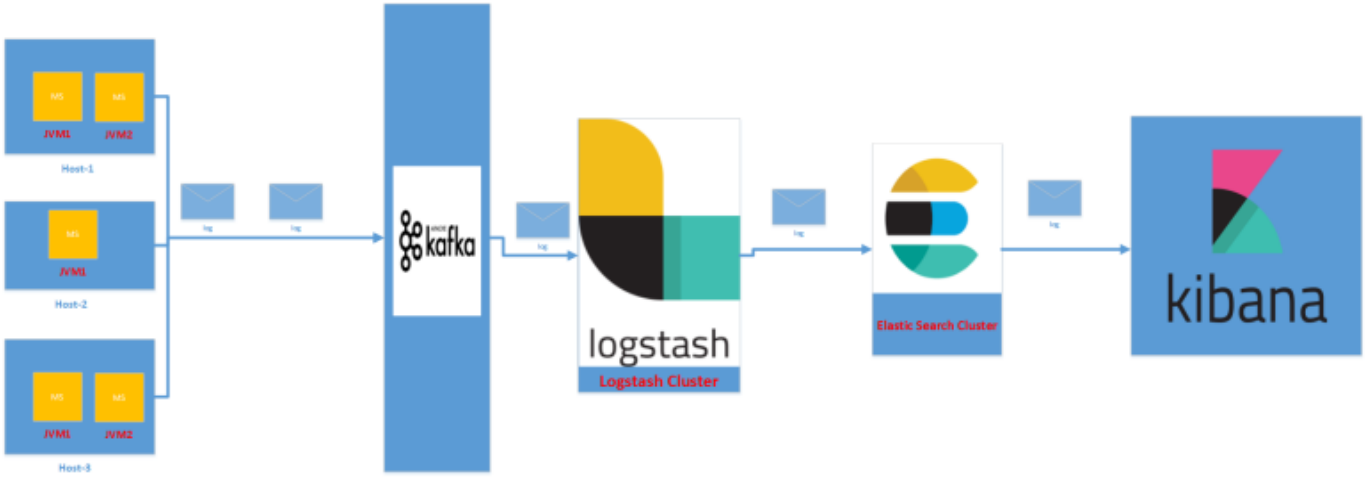
Components

- Message queue service like Kafka or RabbitMQ
- Aggregator service will subscribe to a particular type of msg from Kafka.
- Log Aggregator service - whenever a new log is being written to the messaging queue it should read that and tries to resolve that log by doing some kind of filtration and based on that it can structure that log and store in a DB like mySQl for few times and after which it can perform a ***log rotation***
- Error Aggregation service should read all the error related logs and then structure that data into a DB, later at the end of day/ month we can aggregate that data and generate a report like how many times X type of error has occurred in the Y time.
- Alert Service - Listen to msg queue and if there is a fatal error then it will immediately send a notification through either a chat bot, or ring a call to the person on-call or if there's a UI then notify there, we can also maintain a DB for this as per requirement.
- Have a visualization tool like Grafana, or Kibana and since we have already structured our data so now it can read that data to provide insights in form a Charts or Graphs.

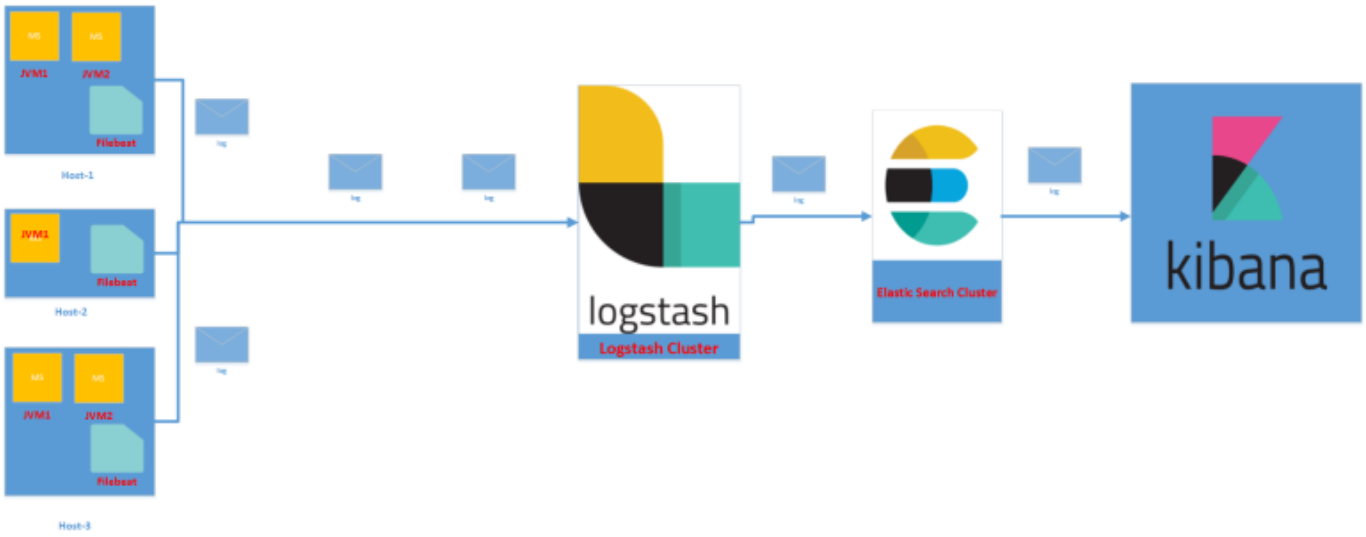


METHOD - 3

OR other way is to use a ELK stack(Elastic Search, Logstash, and Kibana) where you can write all logs in a time series database like Cassandra and then Logstash can be used to process that, and after processing we can use ElasticSearch to index logs and Kibana can be used to perform fuzzy search and provide visual representation through charta, filters, pre-build aggregations.
we configured the Kafka log appender to output the log messages to a Kafka cluster (RabbitMQ/Active MQ can also be used) -> message ingested by Logstash (used for collecting log data from kafka and transform the information if required) -> output sended to Elasticsearch -> search indexed log using Kibana visualization tool



write log msg using a Logstash appender to the file(Filebeat agent) on the host machines → The Filebeat agent will watch the log files and ingest the log information to the Logstash cluster → output sended to Elasticsearch → search indexed log using Kibana visualization tool



First approach is good coz :-

- When a system is large and it needs autoscaling then instances of kafka (which is used in the first approach) can be created and destroyed based on need.
- If in the second design, Filebeat agent stopped working then we may lose the logs from that machine, while in the first Kafka log appender is responsible for transferring logs from host to Kafka cluster.

METHOD - 4

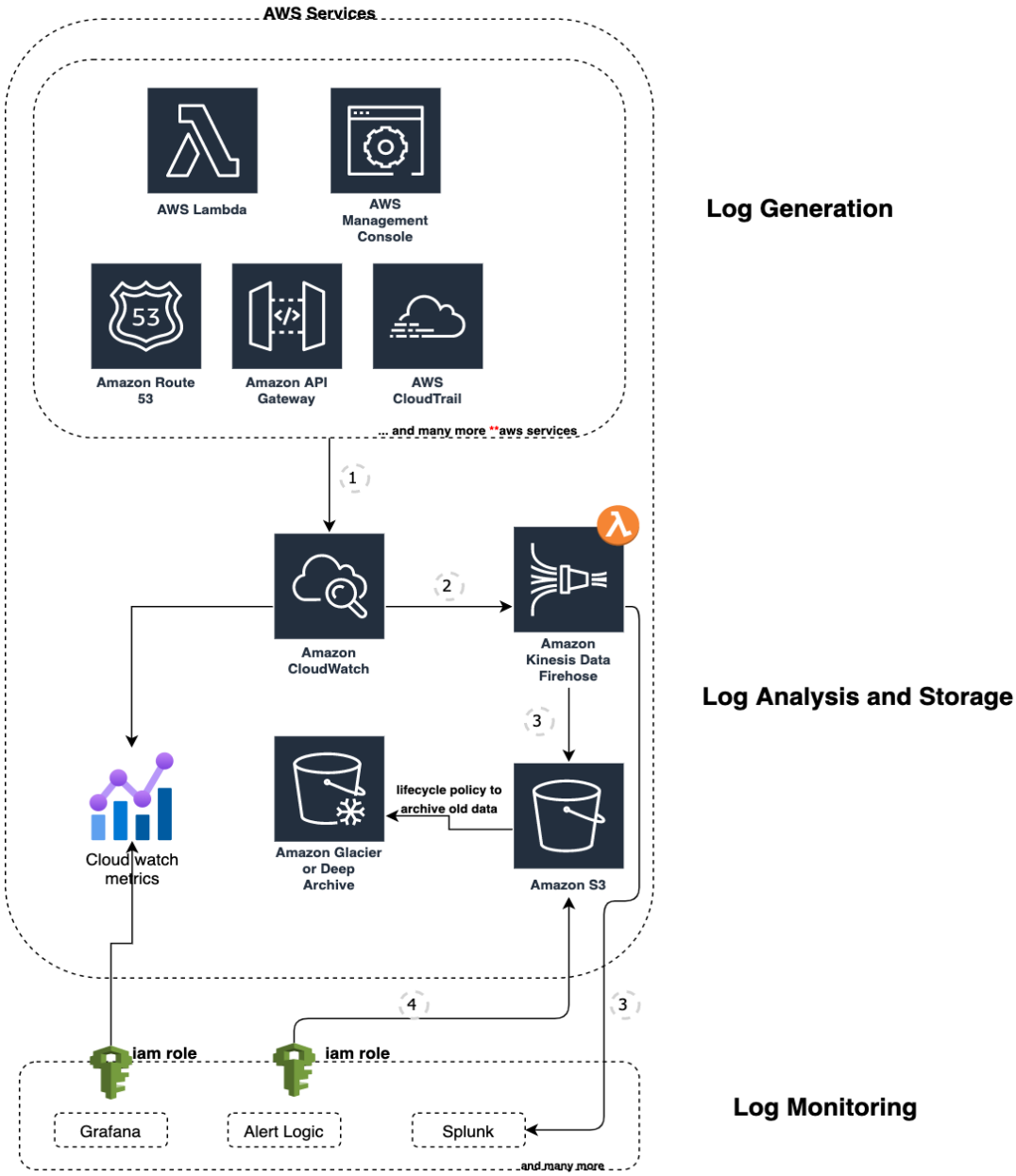
Several AWS service publishes log in JSON form directly to cloud watch which parses JSON and send the stream data into Amazon Kinesis Firehose,

cloud watch metrics make metrics for analysis using cloud watch data and these metrics are used by grafana for visualization

Kinesis firehose then pushes data into S3 bucket and Alert logic can access that data to send urgent notification in case of failure to users.

The extremely old data is archived to Amazon Glacier from amazon S3.

We also tag each log msg with a context_id or correlation_id which make them easier to find lter.



Design Audit System

It is used in org to manage data changes so that it can easily be tracked who has changed the data and what was the existing data and what has changed, this is done to maintain consistency in the system if something goes down.

Clarifying ques → can we afford to loose some data?

Non-Functional req –

1. Highly Available to audit every single info since we can't lose any data.
2. Horizontally Scalable
3. Works in isolated manner, if something goes down then we still want things to work in expected manner
4. Fault tolerant.
5. Access should be fast i.e low latency

To find differences between old and new data i.e what data has been changed, we can use "Apache common" library which finds the differences b/w 2 data also it allows us to define the field we want to

audit/find difference for chosen attributes of a given object so that we don't need to get difference for all attributes.

Where should we place an audit data service?
We can place it at every microservice and then persist this data into DB. But writing the same logic/placing the same service everywhere is duplication of work and also difficult to manage and if we wanted to modify some logic in the audit service then we'll have to write it everywhere. So instead we can build a single audit service which will handle all the audit data from different microservice.

How to read audit data?
We can build an API which takes search parameters/filters and its one endpoint is exposed to the client side. We can call this API from the client side and the audit service will fetch the relevant data from elastic search and give it back to the client.

How will other microservice communicate with the audit service/how will they send their data to the audit service?
It can be sync or async. If we do it sync then other microservice need to wait until current microservice haven't entered all it's data so response time increases and also let's say if audit service is down then we'll loose all the incoming data from microservice and it'll impact other service as well, also since we cann't afford to lose any data so we need something to store data in case audit service is down and we also needs to make audit service loosely coupled i.e work in isolated manner.
If we go for async communication, we can pass audit data to kafka topic which will make sure the delivery of data irrespective of availability. Another advantage of using kafka will be we can come up with multiple partitions based on required throughput i.e it is a horizontally scalable system.

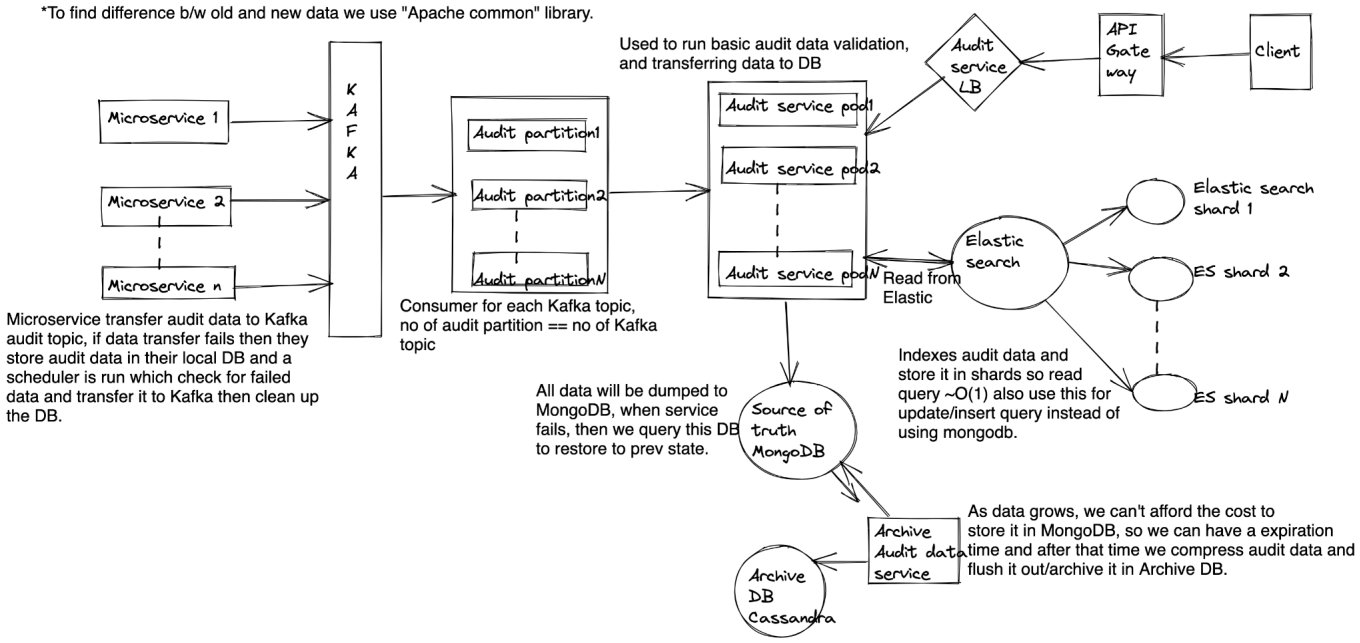
How to process data at audit service/how to respond to read/write requests from audit service?
Since data will be growing exponentially as time passes, responding to read requests (which will include some criteria/filters according to the client) from huge data would be costly. So we can use elastic search (which will index the data and store it,so that the read query becomes ~O(1)) to store data for read requests. We will be running multiple nodes (audit partition) based on no of topics in kafka which will act as consumers of kafka.
Audit service pod will be used to run basic validation of audit data and will transfer audit data to primary data store (source of truth) ie mongoDB (for write heavy we can use cassandra/dynamo as well) and also used to transfer audit data to elastic search for indexing and performing update/insert queries on elastic. When our data grows more then we use sharding to manage data.

What if microservice fails to push audit data to Kafka?
This is a very rare case, but since we can't afford to lose any audit data, we need to take care of this case as well. As soon as microservice generates audit data then store it into DB of that service and then pass it to Kafka audit topic, once passed then clean it up from microservice DB. In case when microservice fails to send audit data then we have failure data stored in DB and we can run a scheduler using cron job which will read failed data from DB and pass it into Kafka and clean it up form DB.

How will I see this data?
We can have an API gateway who will talk to audit service and read_API endpoint, so UI will talk to API gateway which internally fetches audit data and passes it back.

When should we archive audit data?
As data will grow fast, we can't afford to store it in DB since data storage comes with a cost . Also it will make slow read requests. After a certain amount of time decided according to business requirement we

can compress audit data and flush it out and store(archive) it in a data warehouse i.e Hadoop



Design Index retrieval service

You have to design a system to deliver “indexes”, and index is an object with an id and a set of attributes. Team X notify you that all indexes are ready then you can read them from their database. They are ready each day so they notify you each day that "ALL" indexes are ready, and you can start delivering them. Clients show interest in your system into a specified index chosen by them and moreover a set of specified attributes for each index.
The client also specifies a format to receive the file on such as XML, CSV, and so on and a preferred way such as email or FTP.

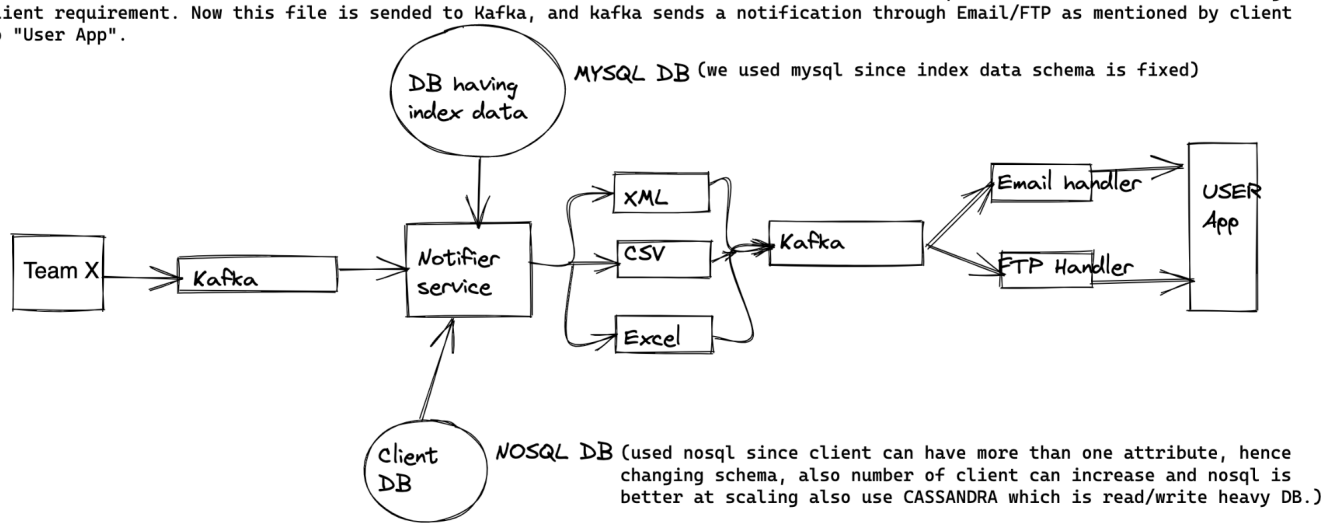
indexes ~ 20000 clients
order of thousands attributes ~ 5000 for each index

=> solution
Function req -
sending notifications,
converting files to different formats,
searching query by specific attributes.

Non-Functional req -
1. Highly Available
2. Low Latency
3. Scalable

API - get_indexes(user_id,index_id,list<attributes>,format,tranfer_mode)

Team X sends an event in Kafka saying that it has done it's work and other can continue, and "Notifier service" consume this event and it fetches data of all client form "Client DB" and make query from "Index DB" according to the required attributes like "SELECT * from INDEX where status is active" and transfer this result to respective format handler according to client requirement. Now this file is sended to Kafka, and kafka sends a notification through Email/FTP as mentioned by client to "User App".



Design a system that takes data from stock market and display it in BBG terminal

Design a system that reads book reviews from other sources and display on your online book store

→
Functional Req → 1. Search feature (author name, book name, category)
2. View Rating
3. View Review from various sites
Non-Fucntional Req :-
Highly Scalable
Highly Available
Low latency

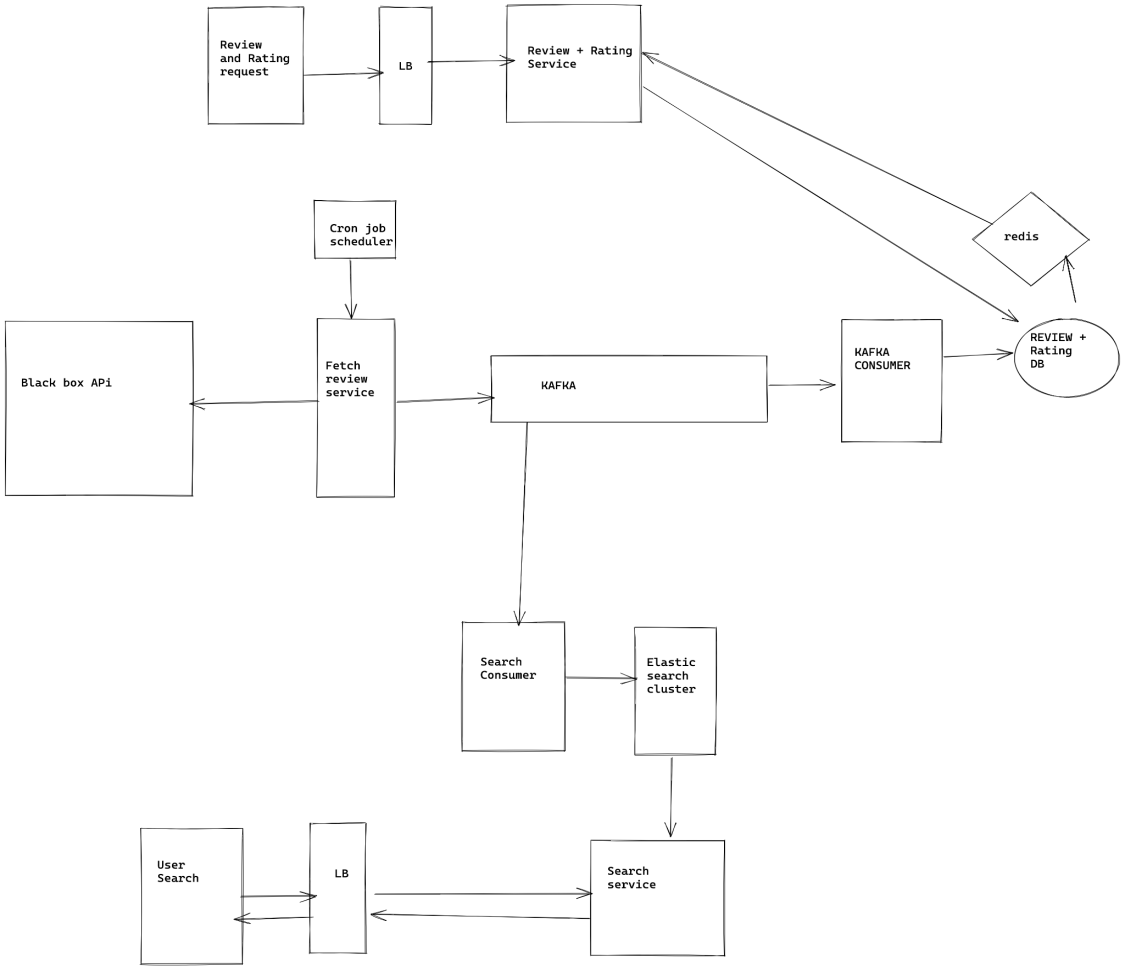
API's :- 1. searchBook(query)
2. fetchReviews(book_id, website_id, positive, negative)
3. fetchRating(book_id, website_id)

/v1/book
GET - search
POST - adding a book
PUT - update a book
DELETE - delete a book

Database :-
Books Table - book_id, author_name, category (MYSQL)
Rating and review Table - book_id, website_id,

Key (book_id) → list[website_id, rating, list_positive_review, list_negative_reviews] (Cassandra)

DESIGN SYSTEM THAT TAKES DATA FROM STOCK MARKET AND DISPLAY IT IN BBG TERMINAL



Design URL Shortener

[CodeKarle: TinyURL System Design | Bit.ly System Design]

B62 Algo code - https://leetcode.com/playground/Ngw7Gmxn

Functional req →

1. Get shortUrl from longUrl i getShortUrl(string longUrl)
2. Redirect to longUrl i.e getLongUrl(string shortUrl)

Non-Functional req →

1. High Availability
2. Low latency

Amount of data we need to store →

Long URL – 2 Kb (at max allowed)
Short URL – 17 byte (7 random char + www.bitly/)
Created_at – 7 byte (7 char)
Expire_at – 7 byte (7 char)
Total — 2.031 Kb

30M user/month => 60.74 GB/month => 0.7 Tb/year => 3.6 TB/year

If we get X req/sec => to support these URL for 10 years => we need A unique URL

$$Y = X * 60 * 60 * 24 * 365 * 10$$

We have possible 62 char => if length of URL is "*L*" => $Y = 62^{L}$

If we take $L = 7$, then our requirement will be met.

To generate random 7 char for short Url, we use B62 and MD5 hashing :-
B62 → input will be integer/long int and o/p 62 char string containing 0-9 + A-Z + a-z
MD5 → input is string and o/p is 62 char string containing 0-9 + A-Z + a-z

Since we need 7 char, so we can just extract the first 7 char string from MD5 algo, but that could result in collision. If collision occurs then you can check for another int/string as input and re-generate the hash and check if collision is still occurring or not. Or else since B62 algo will always generate a unique URL if a unique number is passed, so if we pass a unique number every time then our problem will be solved.

So we can just keep a counter to generate unique number which will be passed to the service containing B62 algo => a unique random string generated => attached to long URL and stored in DB

Now this counter could be a single point of failure, so to avoid this we can use 2 counters which will assign number one by one and a zookeeper to coordinate, but complexity increases and wastage of numbers if one counter fails.

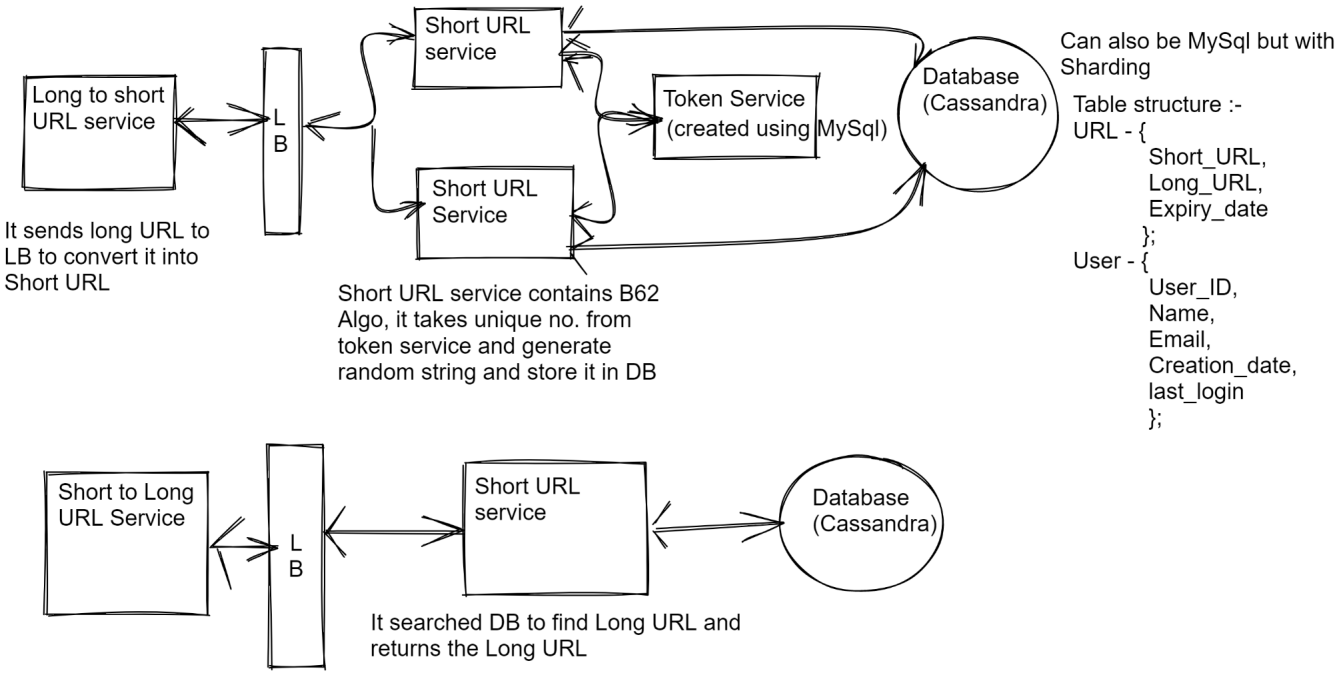
Token service -
It set a range for each service, and to make sure each service has a different range we will use something called Token Service
The token service will run on a single-threaded model and cater to only one machine at a time so that each machine has a different range. Our services will only interact with this token service on startup and when they are about to run out of their range, so token service can be something simple like a MySQL service as it will be dealing with a very minimal load. We will of course make sure that this MySQL service is distributed across geographies to reduce latency and also to make sure it is not a single point of failure.

How do we scale it?
we can either spin up multiple instances of the MySQL instances and distribute them across the map, as mentioned previously, or we could simply increase the length of our range. That would mean that machines will approach the token service at a much lower frequency.

Missing Ranges
But what if one of the services that haven't used up the complete range shuts down? How will we track what part of its range was left unused? We won't keep a track, A few thousand numbers are not significant enough to complicate our system and possibly compromise the performance. So we will just let them go and when the service starts back up we will assign it a new range.

Database
Instead of cassandra, we can use MySQL as well, but it'll require sharding also we need to store billion of rows and there is no relationship between object so we use cassandra, also cassandra is easy to scale.

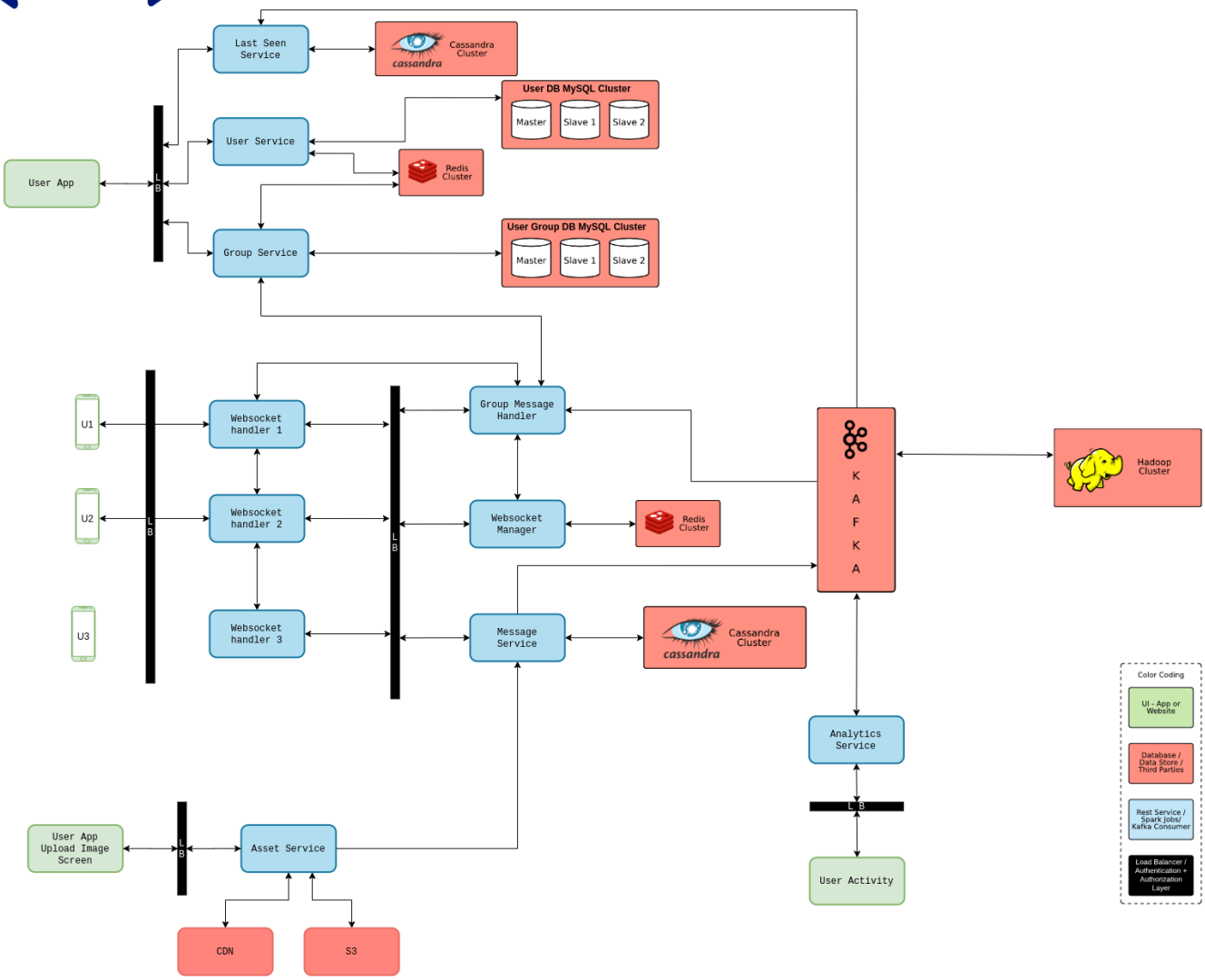
Analysis
For sending data for analysis we can aggregate data locally in a queue, and at regular time interval flush



Design WhatApp

[CodeKarle: WhatsApp System Design | Slack/FB Messenger System Design]

Chat Application System Design - Whatsapp, FB Messenger, etc



Design Twitter

[CodeKarle: Twitter System Design]

Twitter System Design