**Paxos**

分basic paxos 和multi （？）paxos， 角色有proposer，acceptor，notifier

**Raft / Etcd** 的实现用了raft， <http://thesecretlivesofdata.com/raft/>

* node有三种状态：leader， follower 和 candidate （中间状态，等待promote 成leader）
* 两种操作-选leader和commit
* 两种操作对应两种timeout - leader timeout （heartbeat to follower）和commit timeout
* leader 发heartbeat 给follower，如果timeout，则任何follower可以自己promote自己成candidate，同时发信给其他follower，要求选自己当leader
* vote 结果有多数决定，总node个数一定是奇数个
* 每个leader的任期叫term
* Partition之后，数据仍然能consistent，由最新的leader（看term number）决定更新和rollback

**ZAB / Zookeeper** 用了ZAB

和Raft 类似，有几个区别

* heartbeat由follower发给leader
* leader的任期叫Epoch， 不叫term

**Distributed ID generator，** <https://mp.weixin.qq.com/s/8CGN6aeMy9UuI58ZWlUGEg>

**Snowflake** - 64 bit, 最大问题 - 用zookeeper 协调生成

* 1 bit 正负，
* 41 bit timestamp，可用69年，
* 10 bit machine id，
* 12 bit 单机每毫秒内可生产的总数4k - 4\*1024

**Instagram** 的ID生成和Snowflake类似，差别是41+13（shards）+ 10 bit （单机毫秒内产1k id）

**Nosql** - 4种， key-value， document，graph，column

* **Key-value** stores \*\* are the simplest. Every item in the database is stored as an attribute name (or "key") together with its value. Riak, Voldemort, and Redis are the most well-known in this category.
* **Wide-column** stores \*\* store data together as columns instead of rows and are optimized for queries over large datasets. The most popular are Cassandra and HBase.
* **Document** databases \*\* pair each key with a complex data structure known as a document. Documents can contain many different key-value pairs, or key-array pairs, or even nested documents. MongoDB is the most popular of these databases.
* **Graph** databases \*\* are used to store information about networks, such as social connections. Examples are Neo4J and HyperGraphDB.

**API Gateway -** <https://www.youtube.com/watch?v=vHQqQBYJtLI>

**can be used as LB, and other service routing - authenticating, public/private API etc, but it has 10k RPS limit (per AWS), and ALB is 100k+**

An API gateway takes all API calls from clients, then routes them to the appropriate microservice with request routing, composition, and protocol translation. Typically it handles a request by invoking multiple microservices and aggregating the results, to determine the best path. It can translate between web protocols and web‑unfriendly protocols that are used internally.

**API Gateway can manage and balance out network traffic just as a Load Balancer,** just in a different way. Instead of distributing requests evenly to a set of backend resources (e.g. a cluster of servers), an API Gateway can be configured to direct requests to specific resources based on the endpoints being requested.

It plays an important role in microservices architectures, for example. Multiple services can be connected to the Gateway and mapped to particular HTTP endpoint representations. The Gateway is responsible for routing each request, on-demand, to the appropriate backend service.

**1. General Data Processing Design （见下图）**

<https://www.youtube.com/watch?v=bUHFg8CZFws>

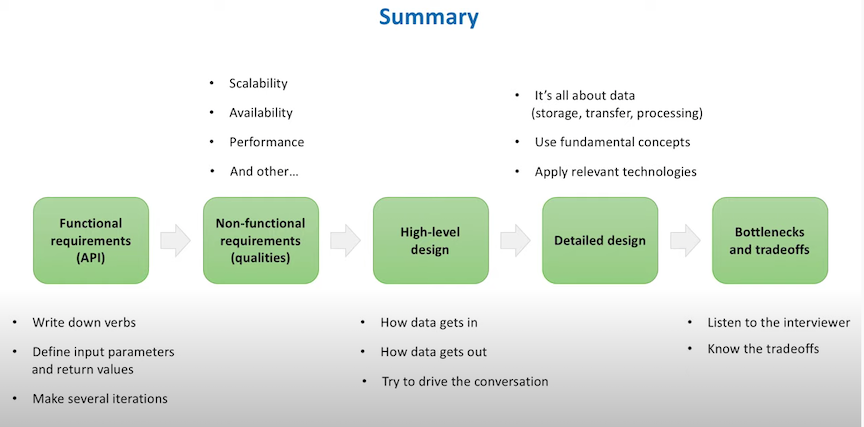
**Functional requirement - API**

问functional requirement 同时可以设计API，列出需求，参数，不断扩大范围，generalize

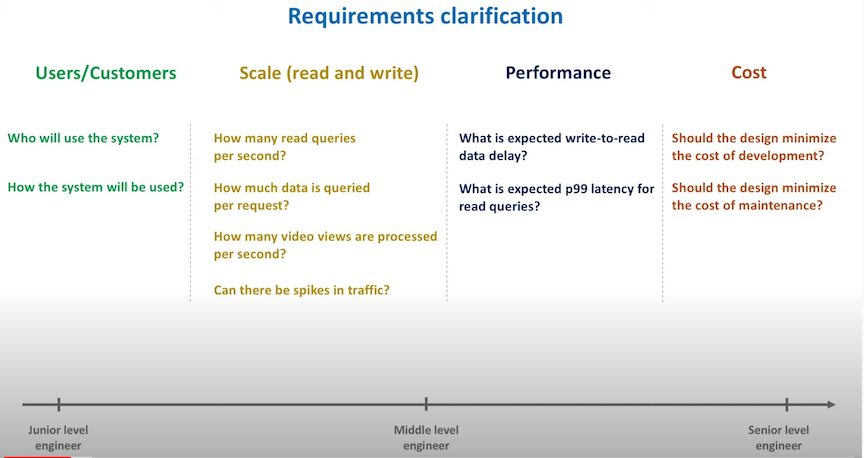
**Non-Functional**

Normally means quality - security, reliability, HA

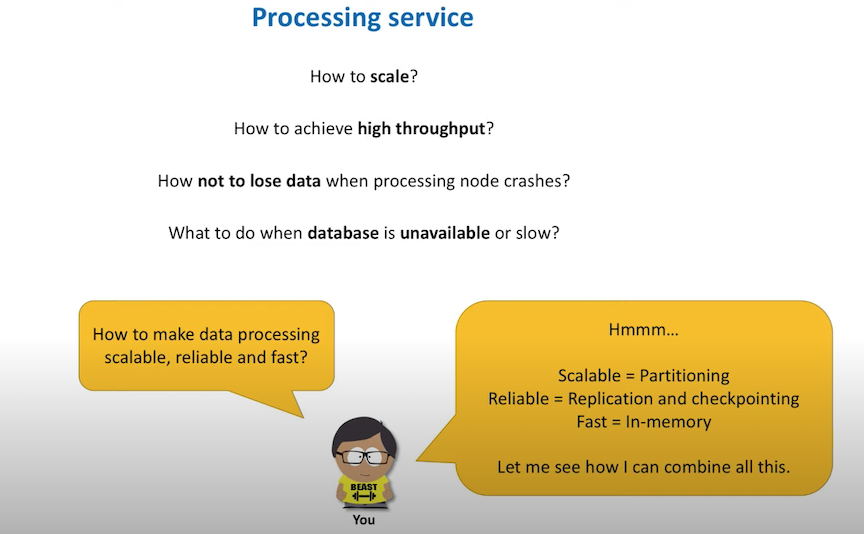
**Summary**



**Requirement Clarification**



**Processing Data in Real Time**

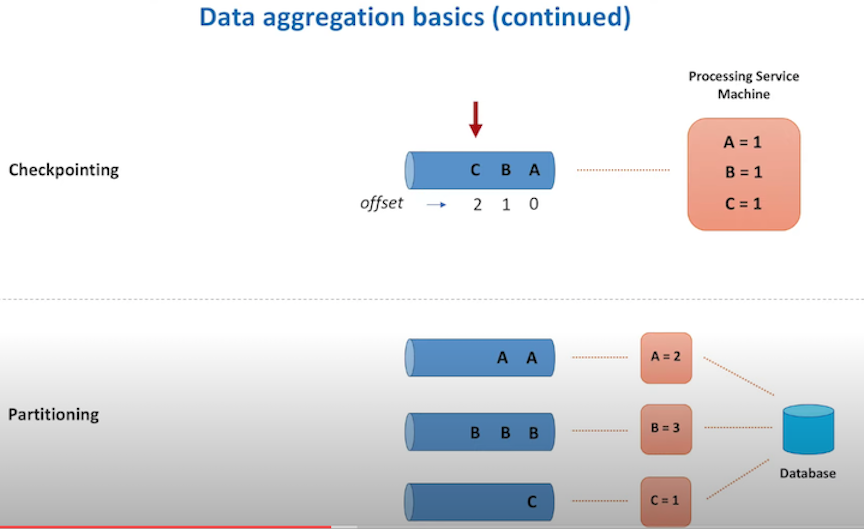


**Checkpointing**

After some data (events) consumed from MQ, persist MQ’s offset (checkpoint) to log. Hence any machine can pickup the checkpointing from log to avoid data loss.

**Partition**

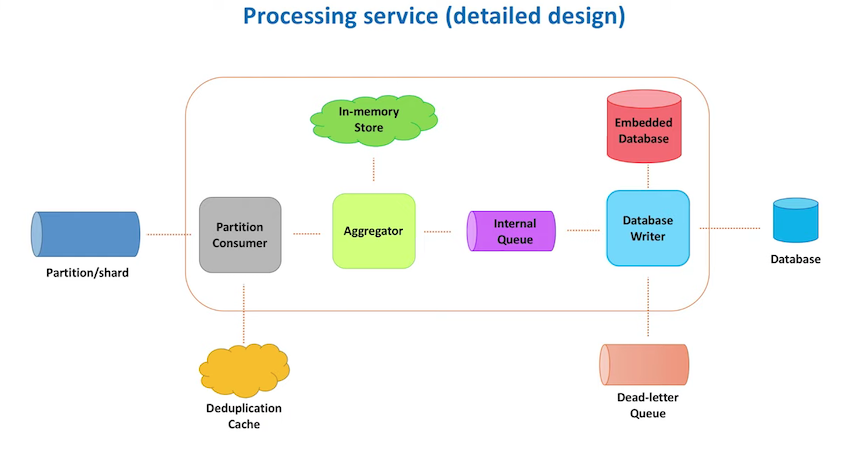
MQ partition, 就是把Queue 分成shards，例如Kinesis的shards



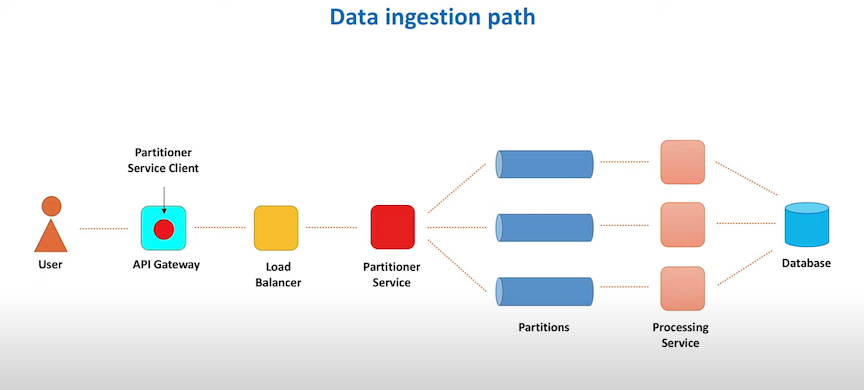
**Data processing component**

Embedded DB 是用作cache，存放metadata，避免API call一类的latency

DB writer可以看作后期的持续处理

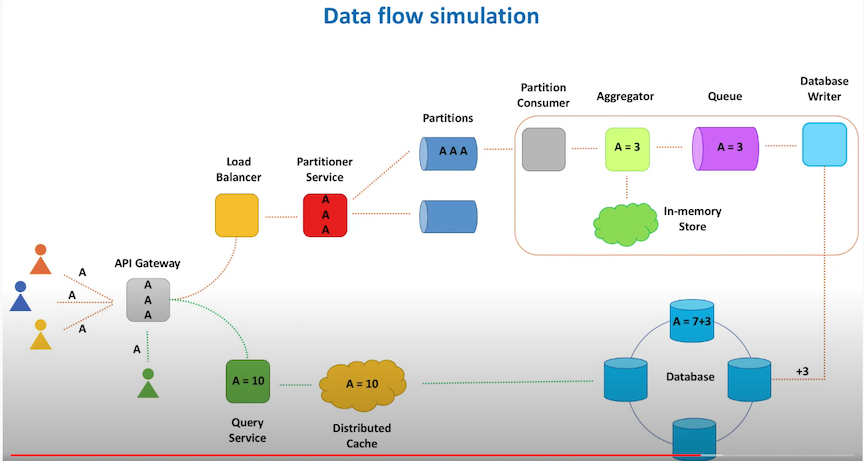


**Architect of data processing**



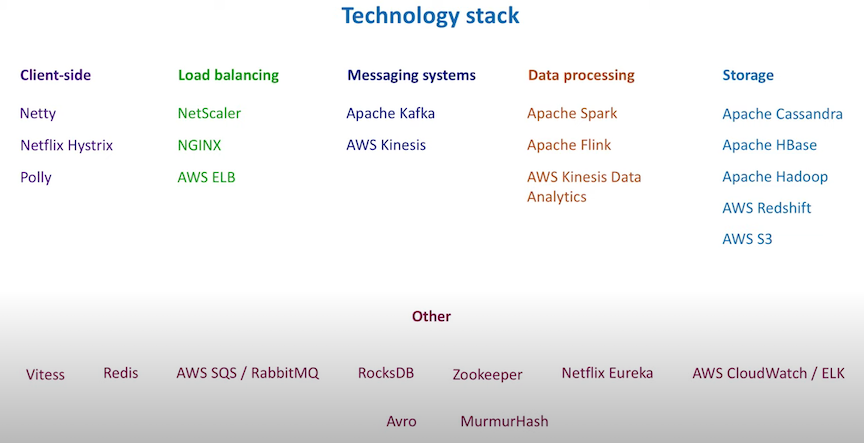
**Data Flow simulation sample - to count how many views**

API Gateway - AWS 有



**Tech stack**

NetScaler - hardware LB



**2. Design Distributed cache**

<https://www.youtube.com/watch?v=iuqZvajTOyA>

**要点**

1. LRU的算法要熟-Doubly Linked List + Hash = HashHeap
2. 配置有cache cluster和colocated 两种 （在local host 上）
3. Cache server 列表要存在DB里 - 每个service的cache client 通过cache server列表query cache cluster/cacher server
4. cache server health monitor - 用ZK heartbeat

Functional

* put (key, value)
* get(key)

Non functional

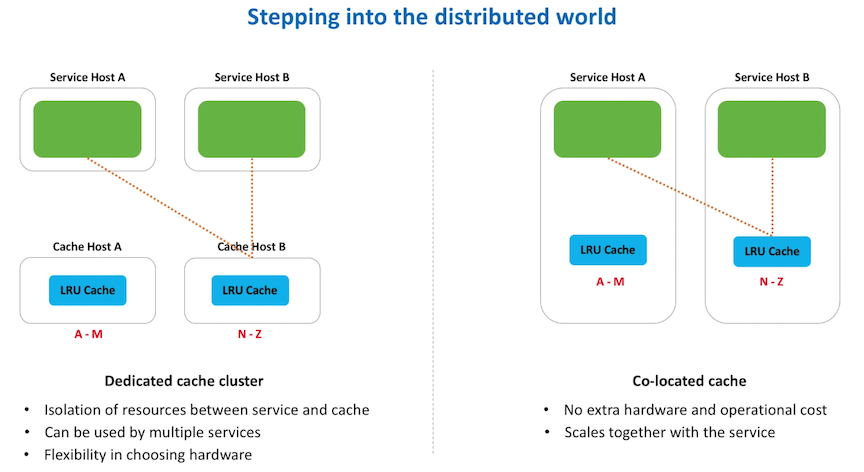
* Scalable (scales out easily together with increasing number of requests and data)
* HA
* Highly Performance
* Reliability -?

Design

Start from local cache, simple -> complex, step by step

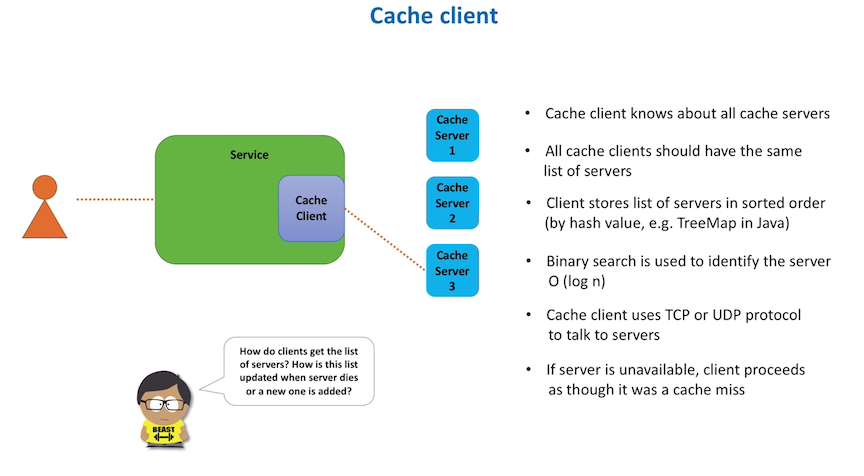
LRU cache

Two architectures - cluster vs co-located cache



**Shards of cache - use consistent hashing**

**Cache client resides in service**

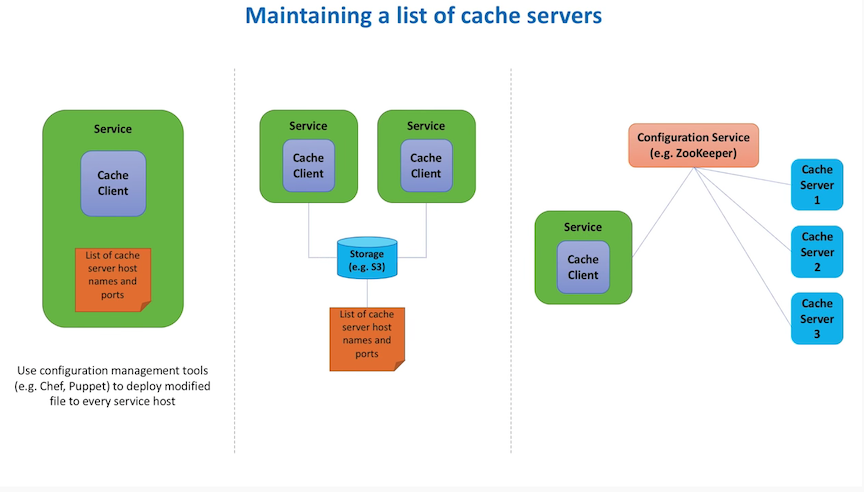


怎么存cache server info 呢？ 有三种方式

1. 直接存在file里， 放到每个service host上，用puppet deploy过去。好处是简单，坏处是一个host变了， 要重新改code + deploy file

2. 用central db.，所有service 都从这里读cache server的info。好处是不用更新多个文件，害处是不能知道即时的server down

3. 用Zookeeper作为Config manager，维护cache server的信息，heartbeat停了，就知道某个server down了。坏处：dependent on zookeeper



**Issue - solve HA and Hot shard problem?**

考虑给cache shards加leader和follower / replica （不同DC）。这里有了consistency 问题：

1. Eventual consistency - Gossip protocol

2. Strong consistency - Raft - 很自然的就想到了用Zookeeper来维护cache shards （node）， 另外Redis Sentinel也可以做这个事

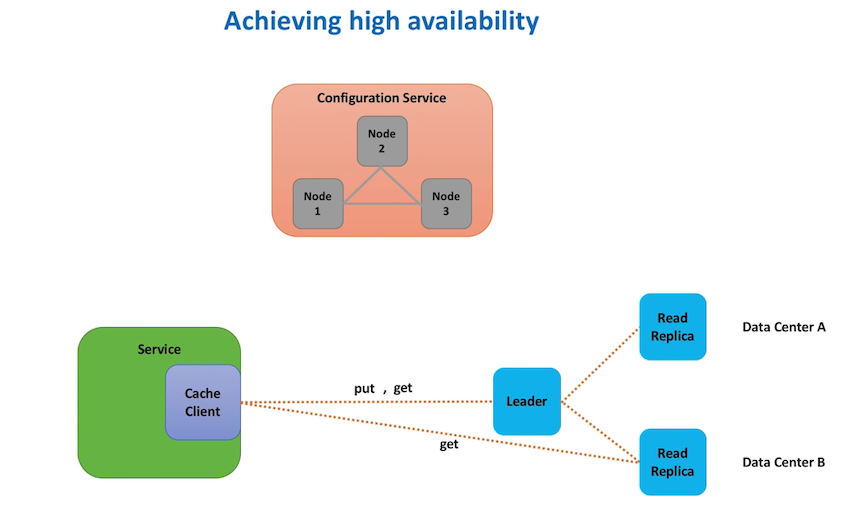
注意

1. 如果有丢数据，是可以接受的，cache就是要快，cache miss ok

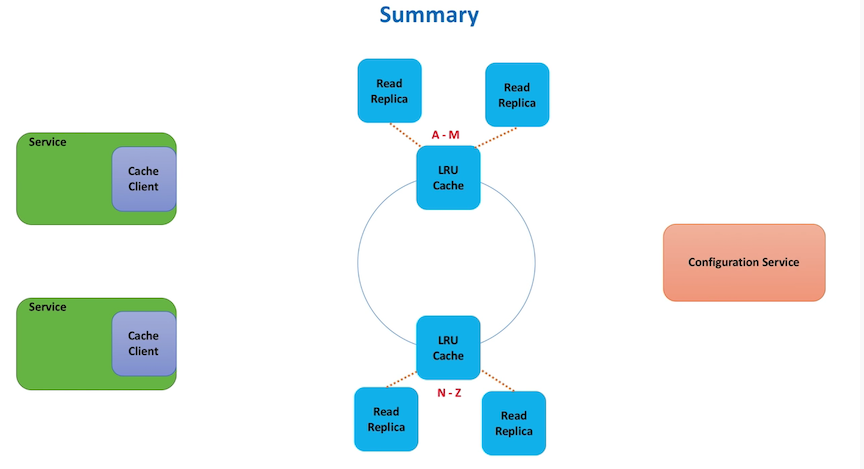
2. 这套系统prefer HA than Consistency。

**额外问题**

* Consistency
* Data Expiration
* Local and remote cache - 先local再remote
* Security - not exposed to internet， should be firewall for whitelisting
* Monitoring and logging
  + Latency
  + Number of misses
  + Network IO
  + CPU
* Cache client - proxy



**Overall Structure**



**3. Top K Heavy Hitter Design**

<https://www.youtube.com/watch?v=kx-XDoPjoHw>

**要点**

1. stream 有accuracy的问题
2. 分需求计算 fast & slow 两种aggregator （对应分秒级和hour以上的agg）
3. fast 计算是second 级别的，要用Count-Min Sketch概率算法，aggregate到minutes report
4. slow 计算要用Map-reduce，是hour级别的，aggregate到day report

**Difficulty**

* data comes in stream of 100K+, cache not enough to handle. Map Reduce has long latency, alone M-R is not enough
* 静态数据可以partition再用K-way merge，但是streaming怎么处理？

**Function**

topK(k, startTime, endTime)

**Non-Function**

* Scalable
* HA (survive hardware network failure, no SPOF)
* High performance (few tens of ms to return top 100 list)
* Accurate (as accurate as we can get)

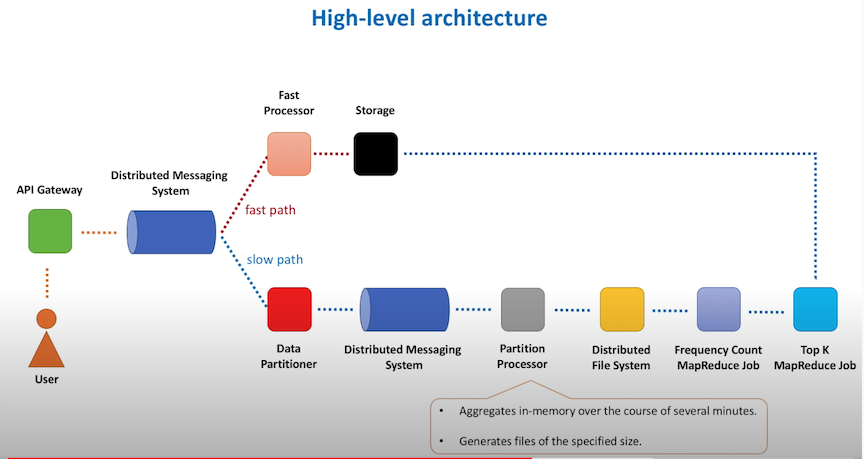
事实上，100%准确的top K对streaming不存在，我们只能逼近，方法一，count-min sketch, 就是bloom filter的另一种应用 <https://www.youtube.com/watch?v=ibxXO-b14j4>

**API Gateway**

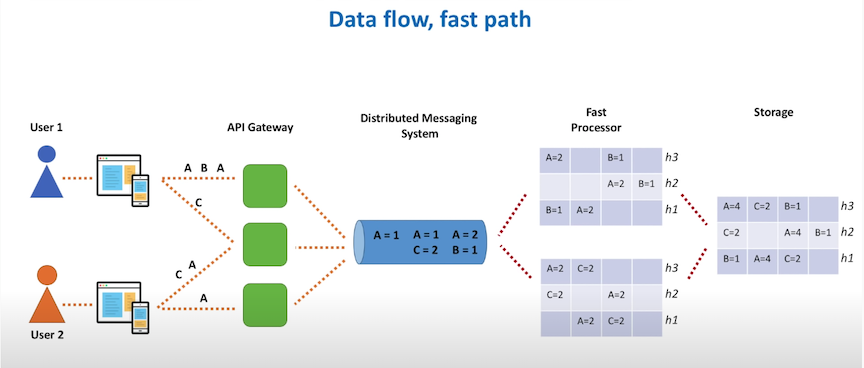
* Single entry point for all clients
* Aggregates data on the fly or via background process that processes logs. Data is flushed based on either time or size
* Serializes data in a compact binary format (eg Apache Avro)

Pipeline divides into fast path and slow path. Fast path get data in seconds

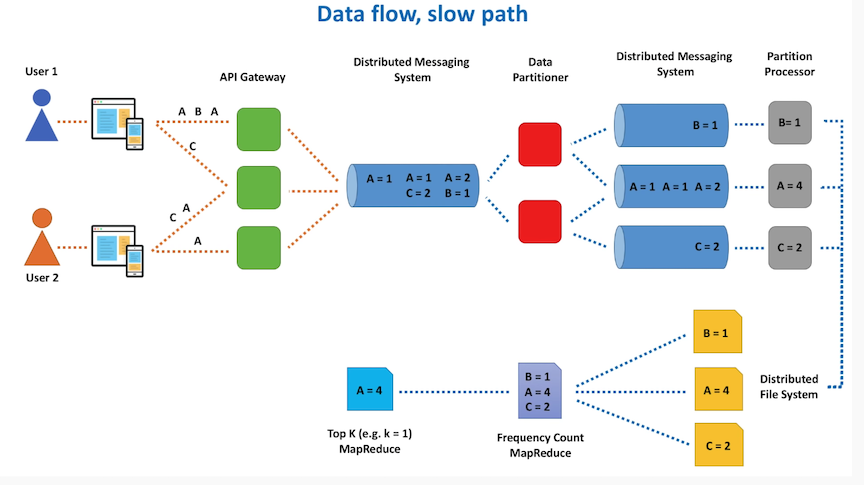
**Overall Architect**



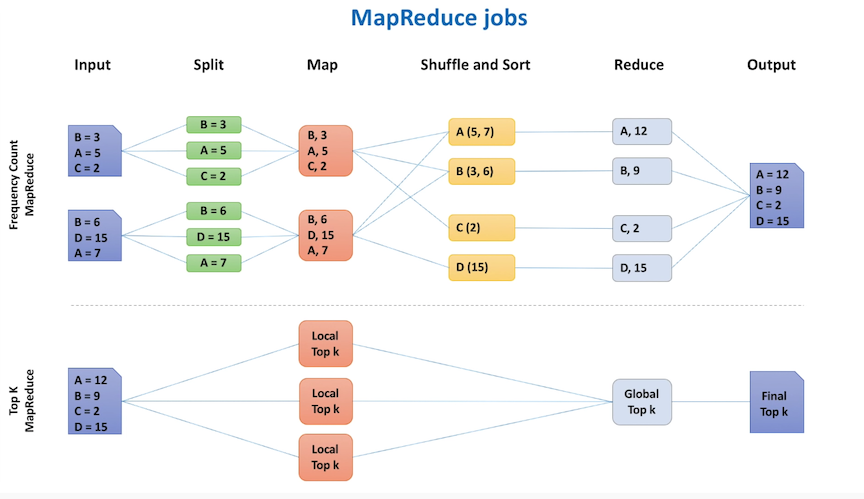
**Fast Path**



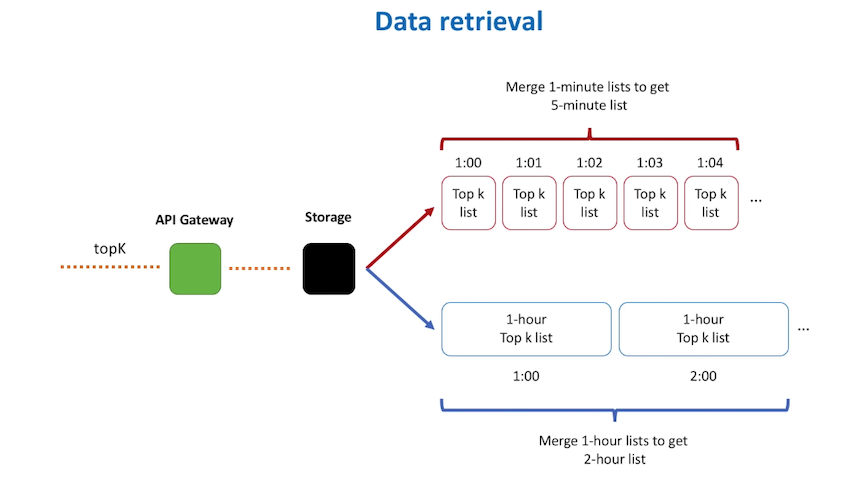
**Slow Path (accurately count from GFS)**



**Map reduce job**



**Data Retrieval - aggregation is an approximation**



4. **Rate limiter Design**

[https://www.youtube.com/watch?v=FU4WlwfS3G0](https://www.youtube.com/watch?v=FU4WlwfS3G0&t=6s)

**要点**

1. 为了快速查询，rate limiter一般放在service的相同host上（甚至可以在同一service process内），不能把Rate Limiter做成service （rule 可以是service）！
2. rule based - 主要metadata （config）的维护， 单用一个rule service
3. cluster 内要sync 当前总的req 数 （ZK或Gossip）

**Functional**

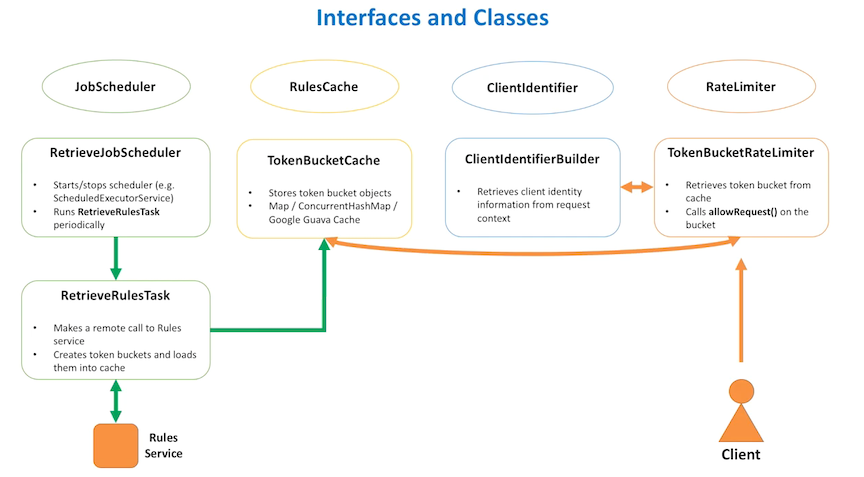
allowRequest(request)

**Non-Functional**

* Low latency
* Accurate
* Scalable (supports large number of services)

**有三方面可以考察面试：**

* Algorithm - Token bucket
* Scalability
* OOP design rate limiter class - JobSchduler, RulesCache, ClinetIdenfitier, RateLimiter,



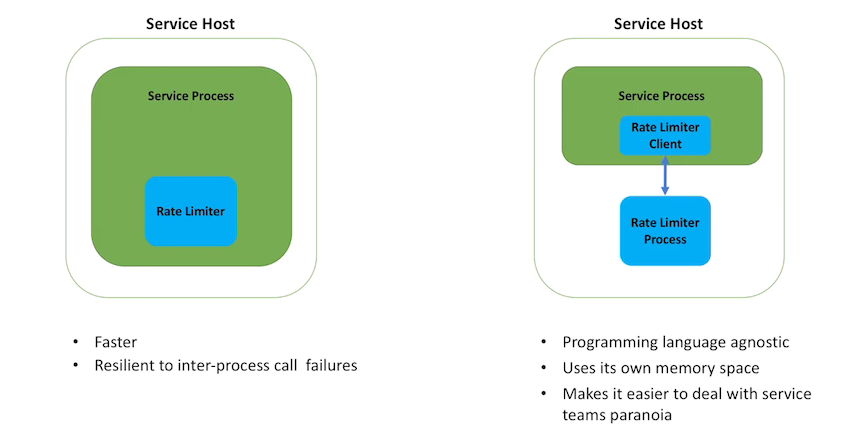
**Question 1** - how to distribute limits among multiple limiter hosts?

答案：比如4rps total， 每个bucket放4个token。Request进来了，正常加减，然后hots 之间要sync 当前的总req 数，允许负数, 可用 Gossip, ZK, etc

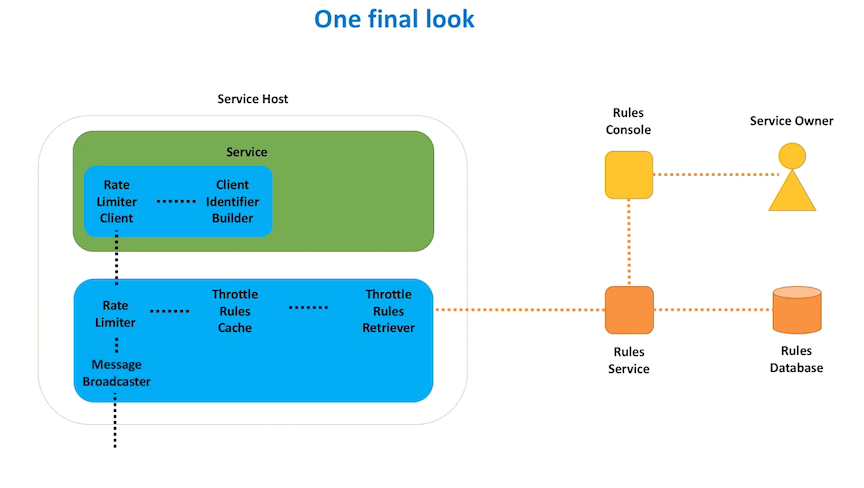
**Question 2** - rate limiter in service or out of service ? Both are good

答案：都可以，看实际需求

**Question 3 -** how does client do being throttled? Answer: exponential backoff



**Overall architect**



**5. Design Distributed Message Queue**

<https://www.youtube.com/watch?v=iJLL-KPqBpM>

**要点**

1. 怎么partition queue - in cluster （ZK） & out cluster
2. Message有Delete的问题-一般读后不删除，有的需要user自己维护（kafka），有的读完不再visible （AWS SQS）
3. FIFO 和 exactly once mode 是很难的问题，如果要实现，一般会降低throughput，比如SQS

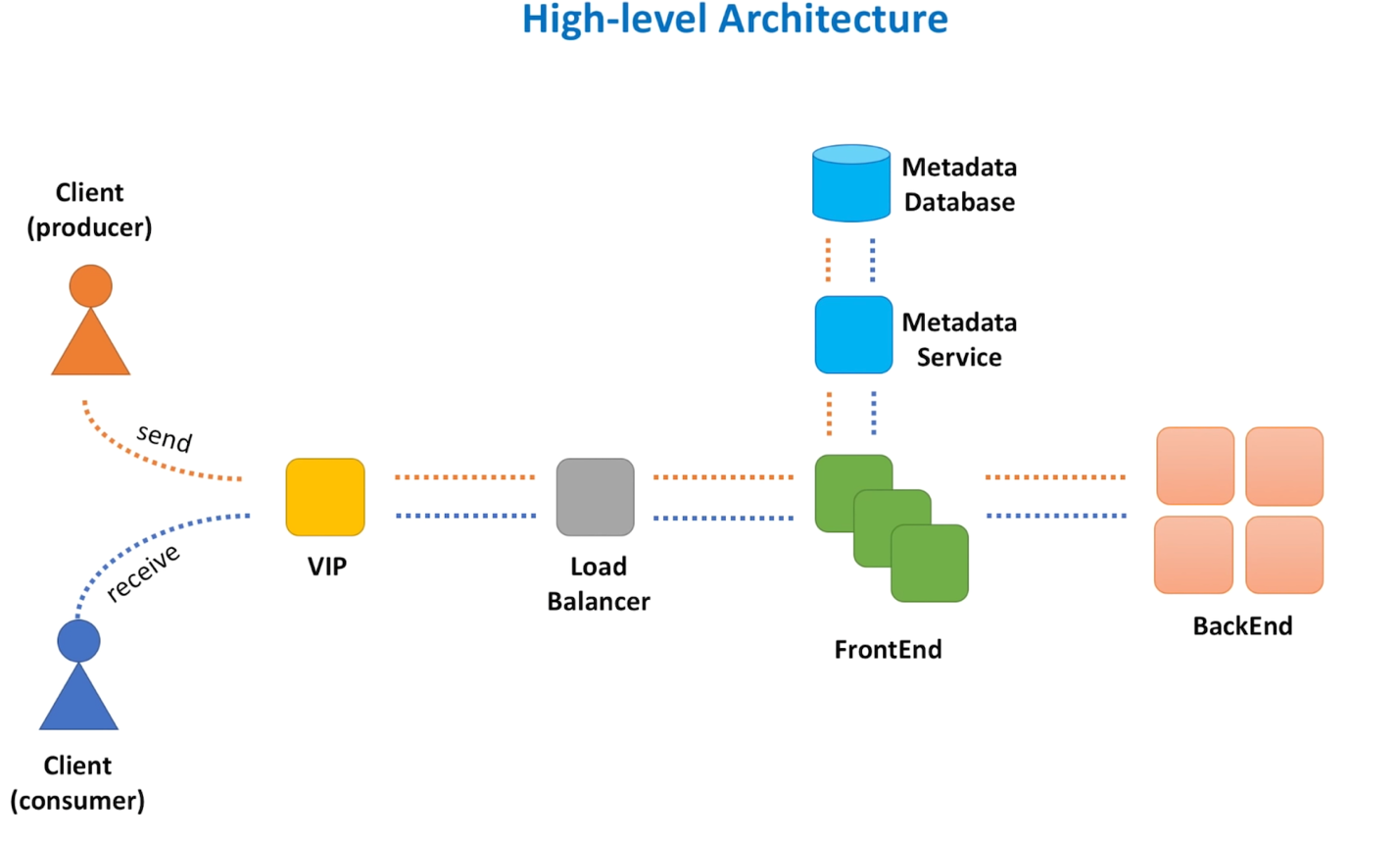
**Functional**

* sendMessage(message\_body)
* receiveMessage

**Non- functional**

* Scalability
* HA
* High Performance

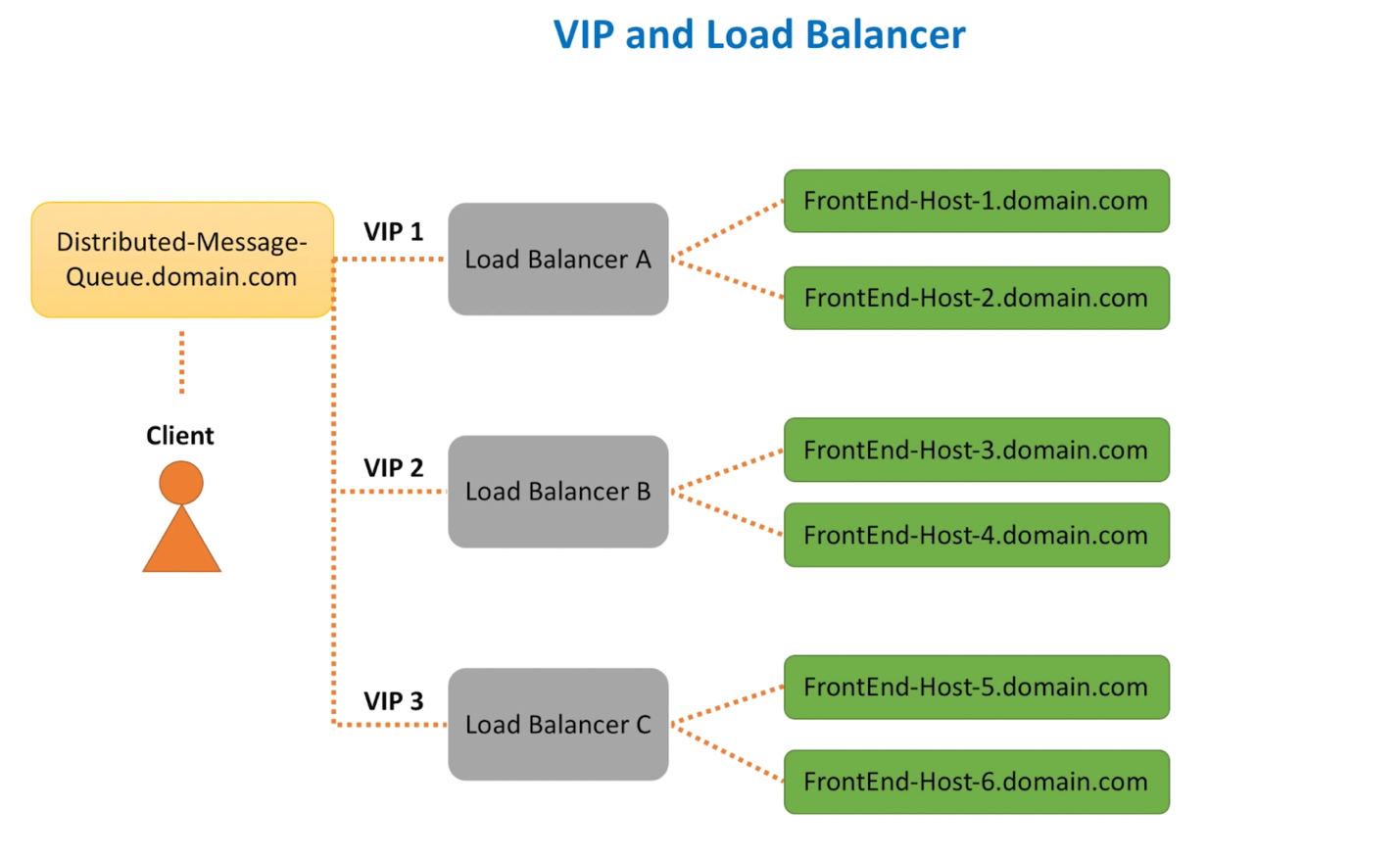
**Overall Architecture**



**VIP and LB**

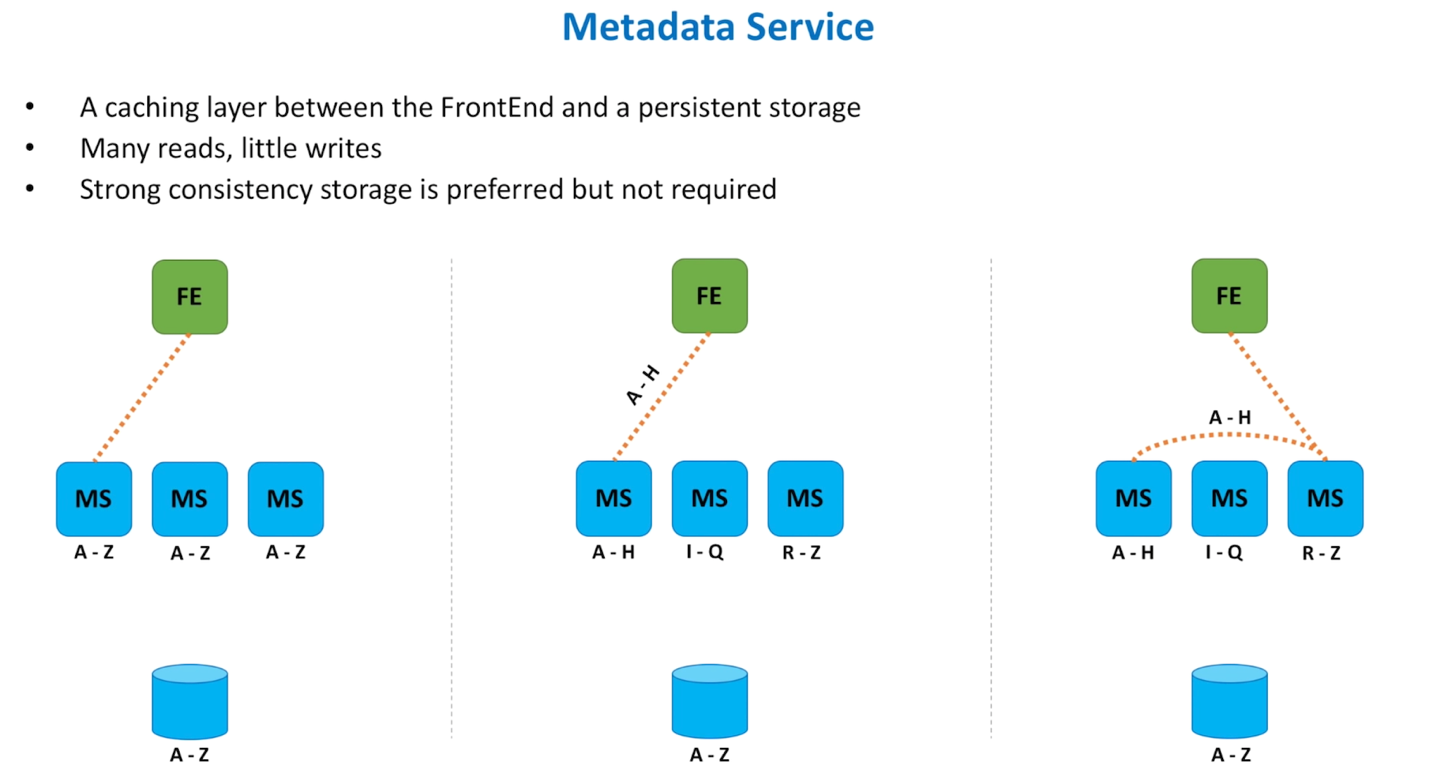
LB could have master and slave setup

VIP can scale out too



**Metadata**

Is a cache layer between FE and Meta DB



**Where do we store messages:**

answer: RAM and local host a backend service

**How do we replicate data?**

Replicate within a group of hosts - cluster

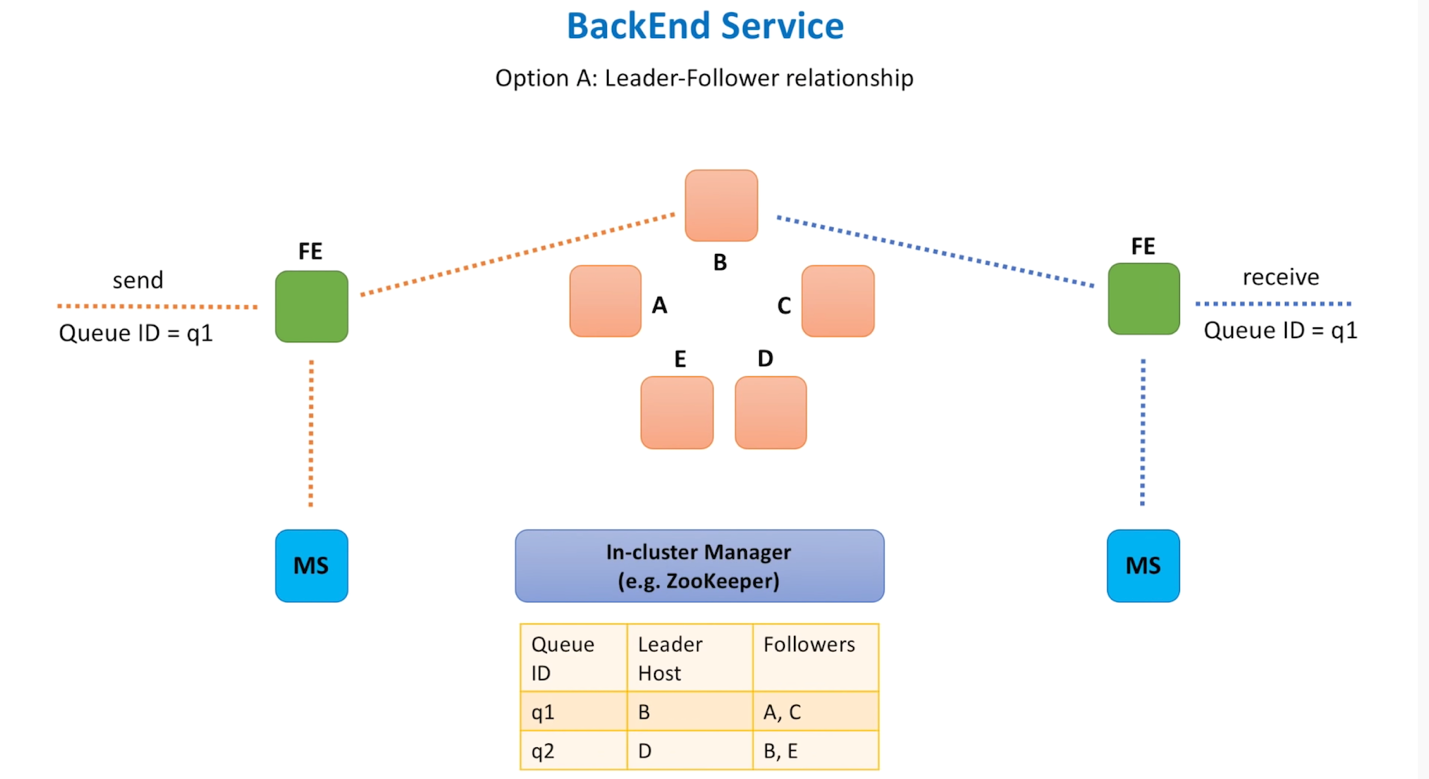
**How does FE select a BE host to send data to? How does FE know where to retrieve data from?**

Answer: meta service

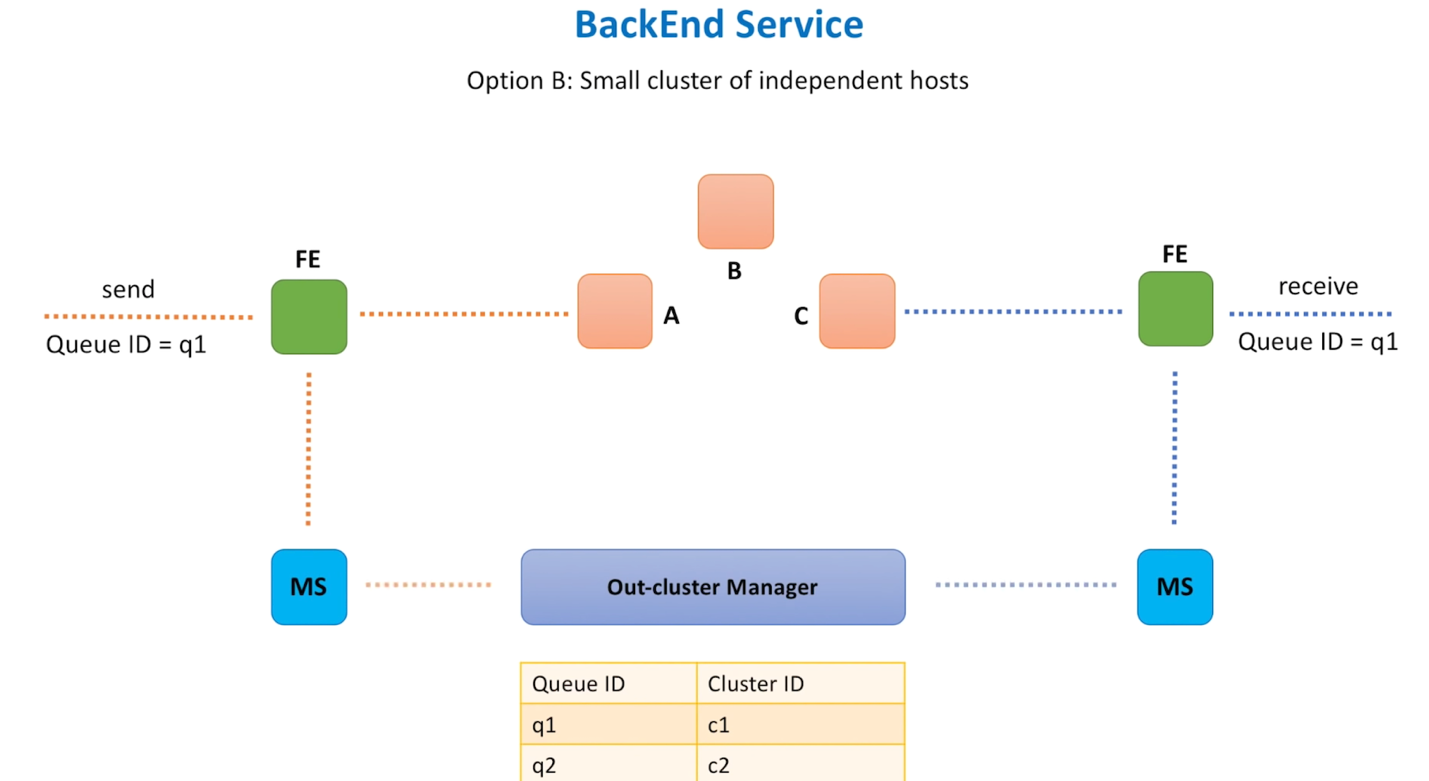
**Backend Architecture**

Zookeeper maintains the leader-follower relationship of node clusters. Metadata service tells FE where to route/retrieve the message to / from

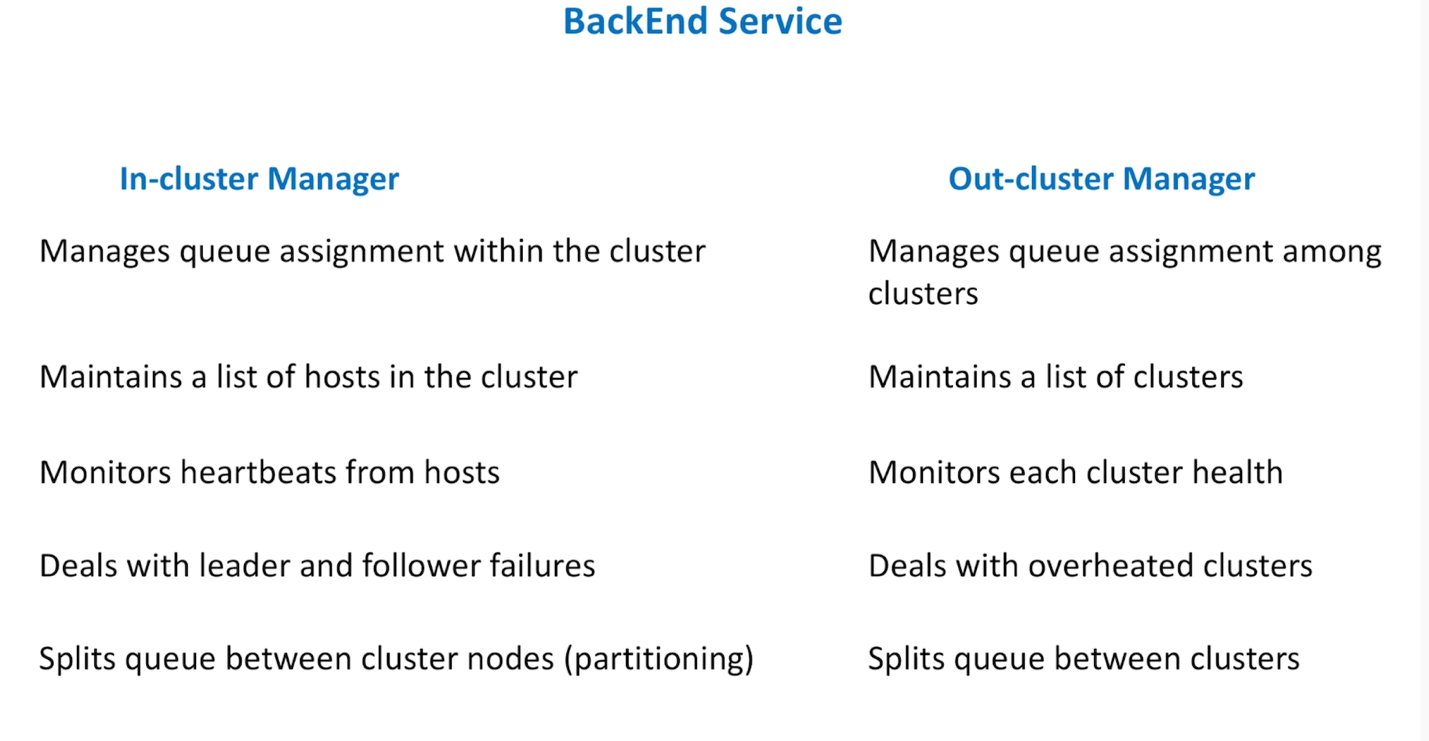
Option A - ZK with leader follower relationship



Option B - no leader/follower, all nodes are equal, random select one to process



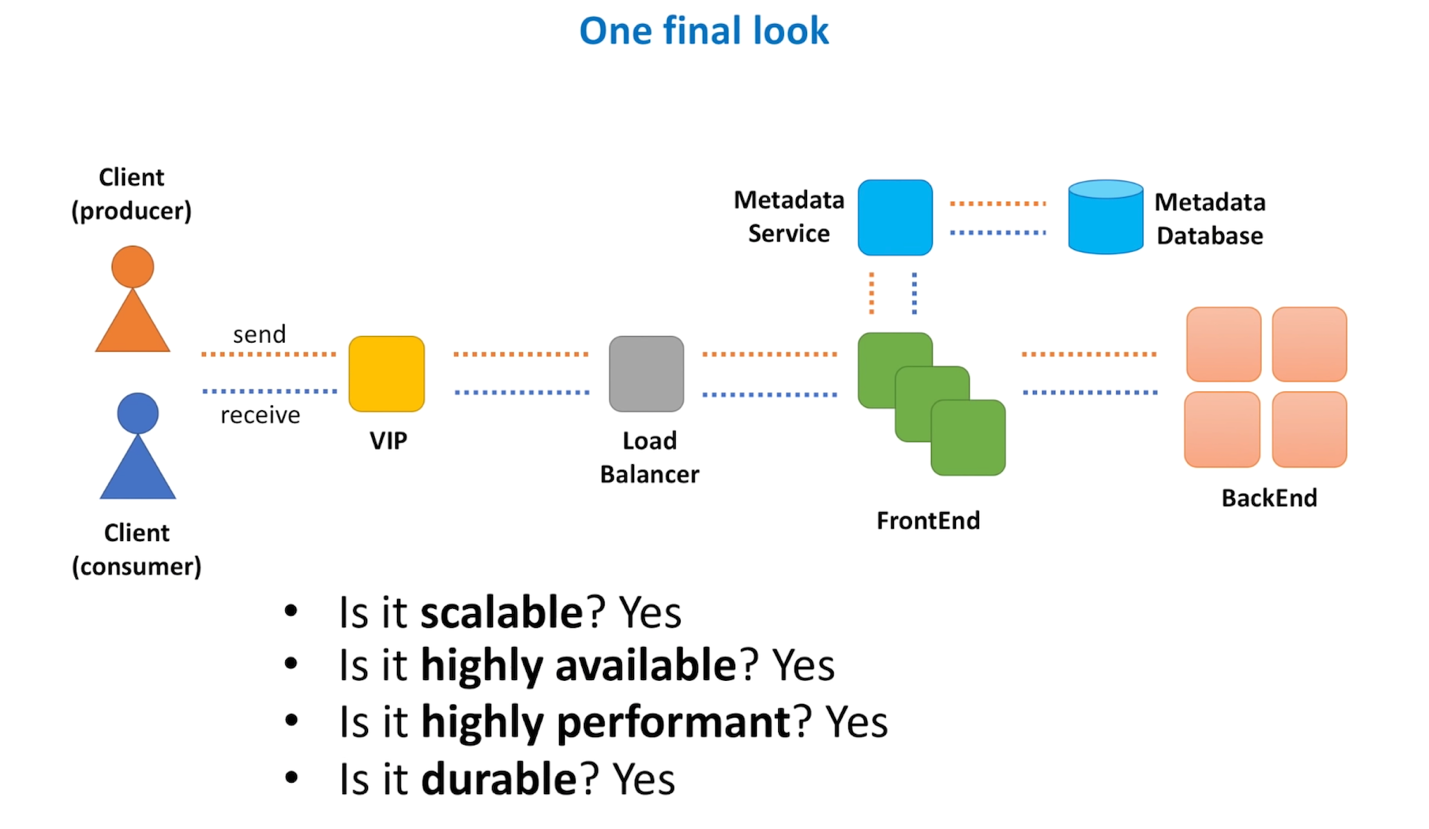
Compare in cluster and out cluster manager



**Questions**

* Queue creation and deletion
* Message deletion, a. Not delete the messages; b. Message is not deleted and not Visible (AWS SQS)
* Message replica, async is preferred, but doesn’t guarantee survive failure
* Message delivery semantics, at least once, exactly once, at most once etc
* Push vs Pull, pull is easy to implement but push is preferred by users
* FIFO, generally FIFO drags down performance
* Security
* Monitoring

**Final Look of the design**



**6. Design Distributed Notification**

<https://www.youtube.com/watch?v=bBTPZ9NdSk8>

**要点**

1. HA 系统，message不能丢，一定要考虑retention period，比如Kakfa/SQS，要存7天
2. temp message store 用那个？Kafka或者Cassandra
3. Sender 的处理速度会是bottleneck，做两个优化-每个message一个job，每个job用micro service进一步细分，提高throughput

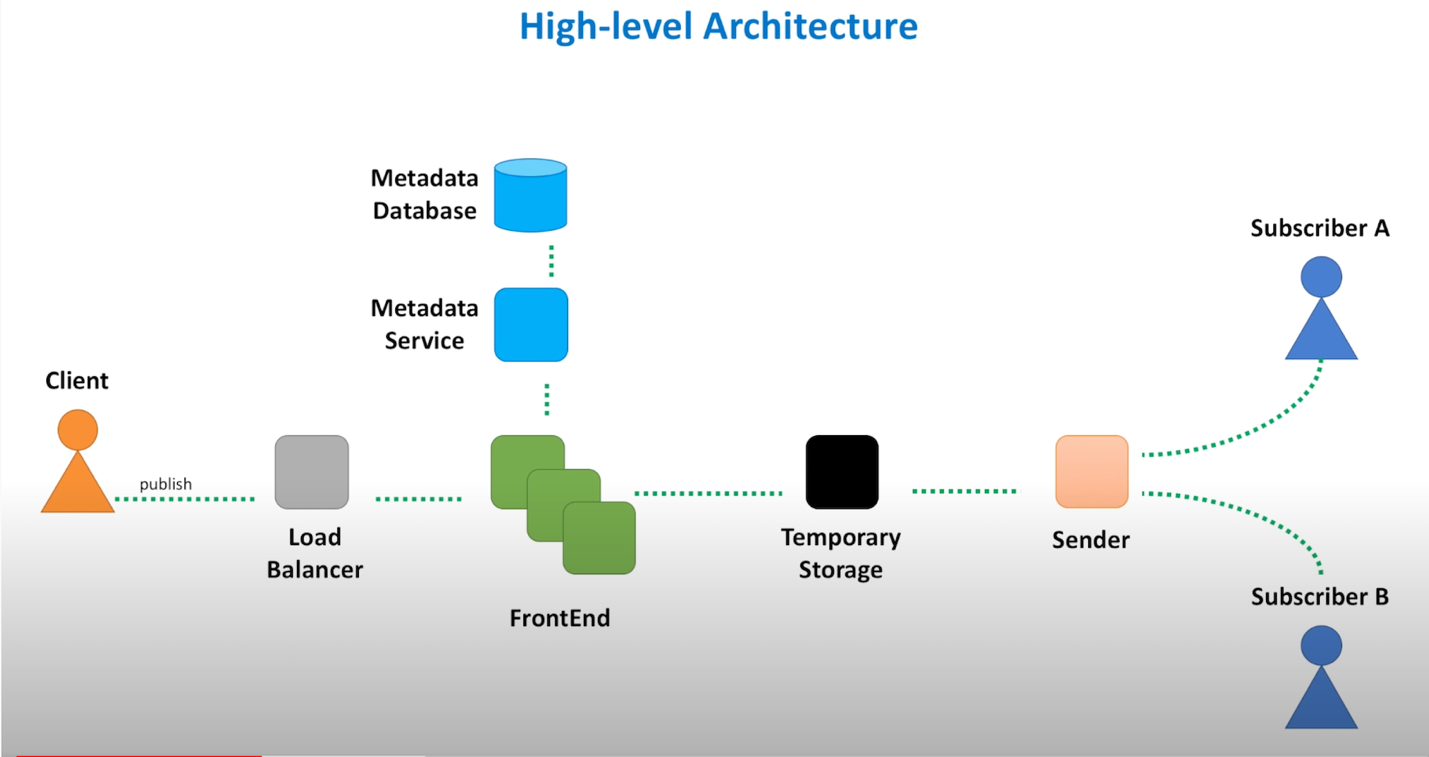
**Functional**

* createTopic (topicName)
* publish(topicName, message)
* subscribe(topicName, endpoint)

**Non-Functional**

* Scalable
* HA
* High performance

**Architecture**



**FE functions**

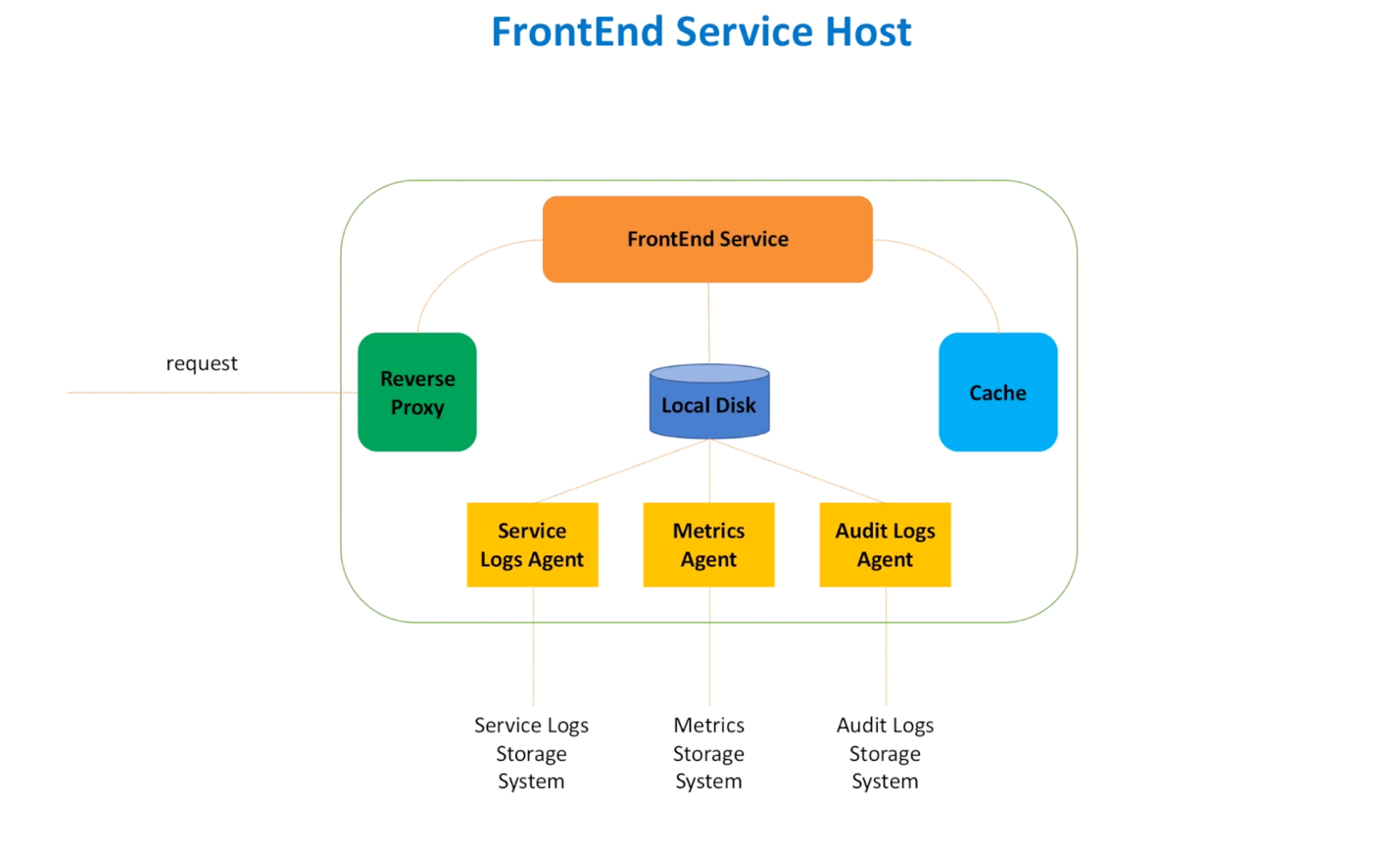
* A lightweight web service
* Sateless service deployed across different regions (DC)
* Request validation
* Authentication / Authorization
* TLS (SSL) termination
* Service - side encryption
* caching
* Rate limiting (throttling)
* Request dispatching
* Request dedup
* Usage data collection

重点

1. Reverse Server做
   * Decrypt SSL and parse in encrypted form
   * 压缩response， 返回给user，这里不重要，因为notification payload 不大
   * pass request to FE

2) FE write log/metric/audit data to local log

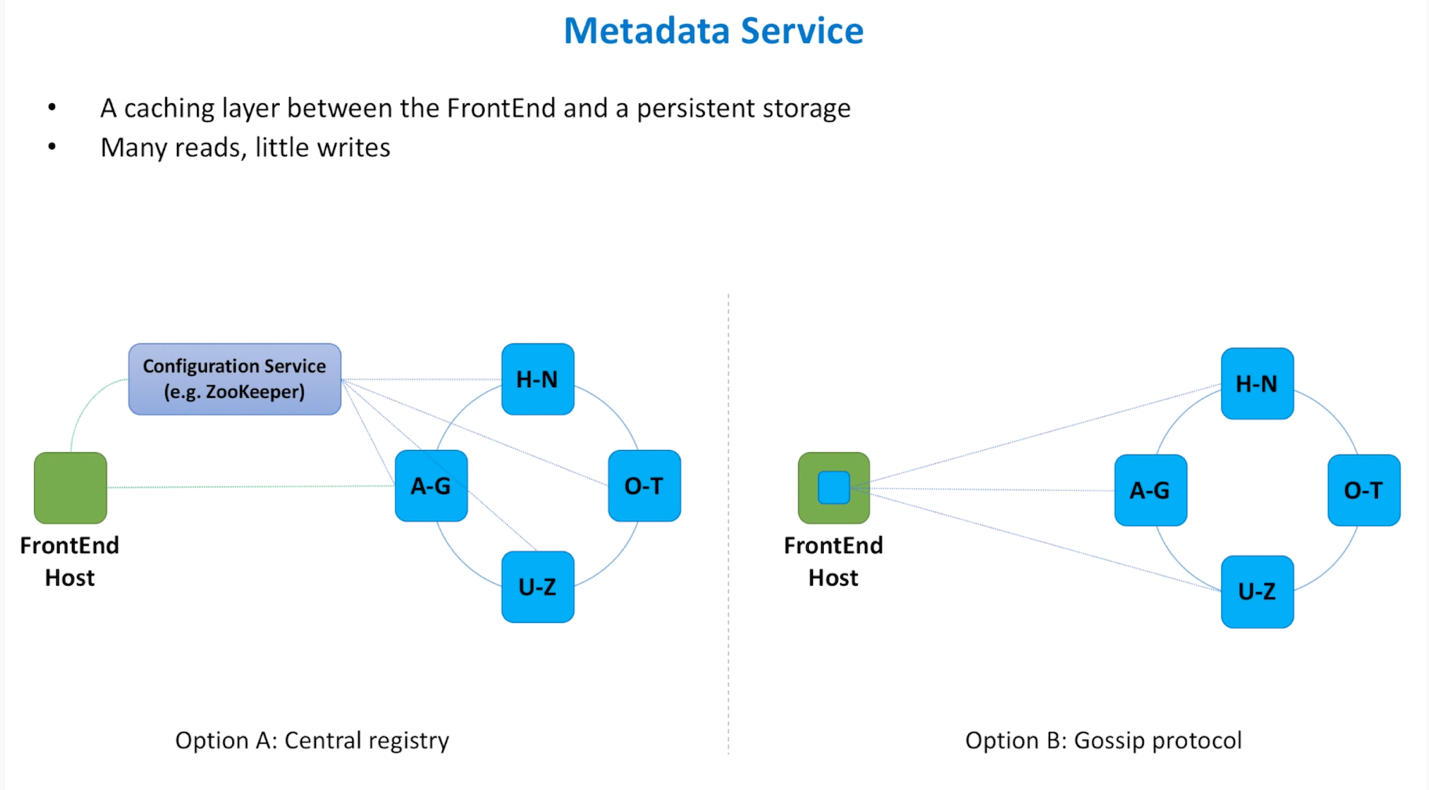
1. 3） **Agent** aggregates local log/metric/audit to 专门的service



**Config** / **Metadata Service**

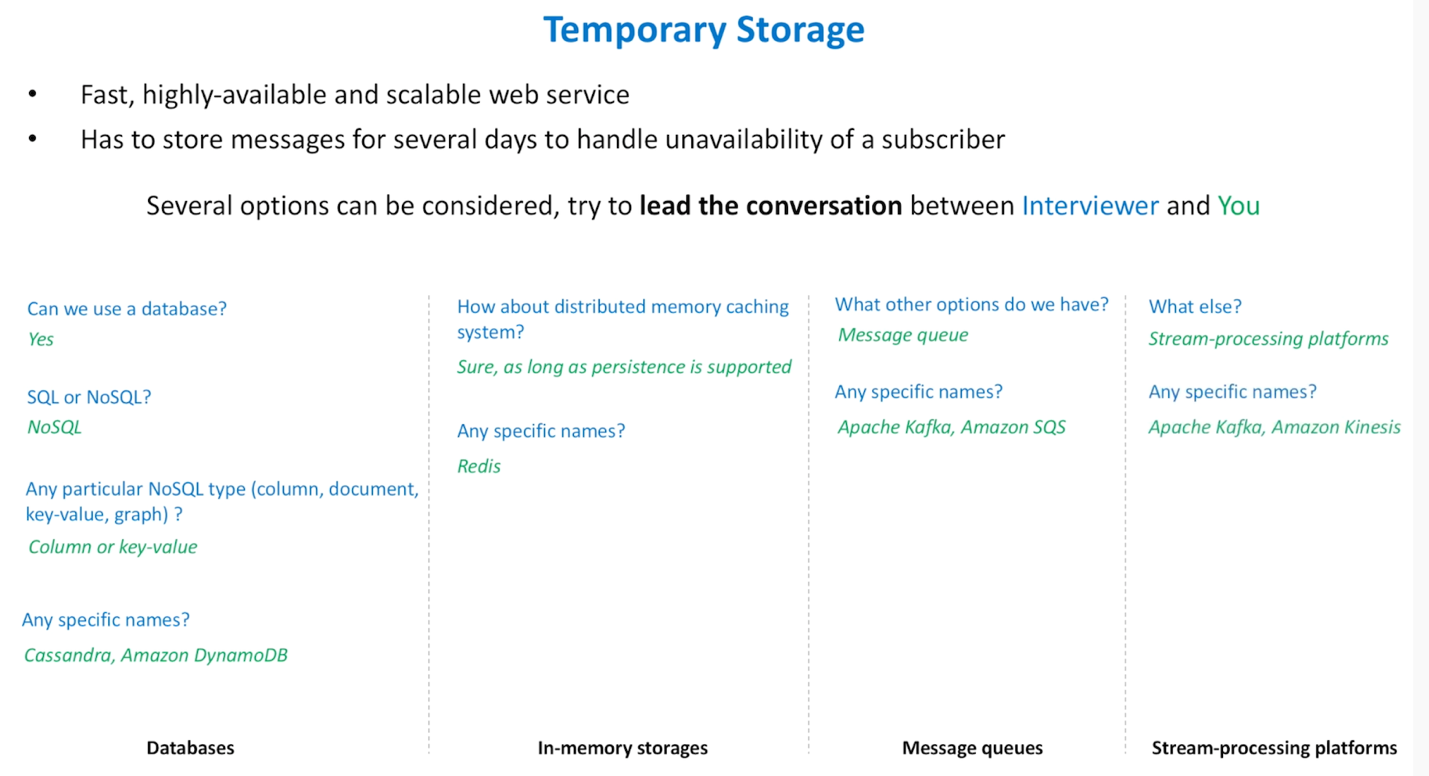
Store info of topics in db, it is a distributed cache, a cache layer between FE and DB

FE communication with Metadata 有两种方式-central （ZK）， open （gossip）



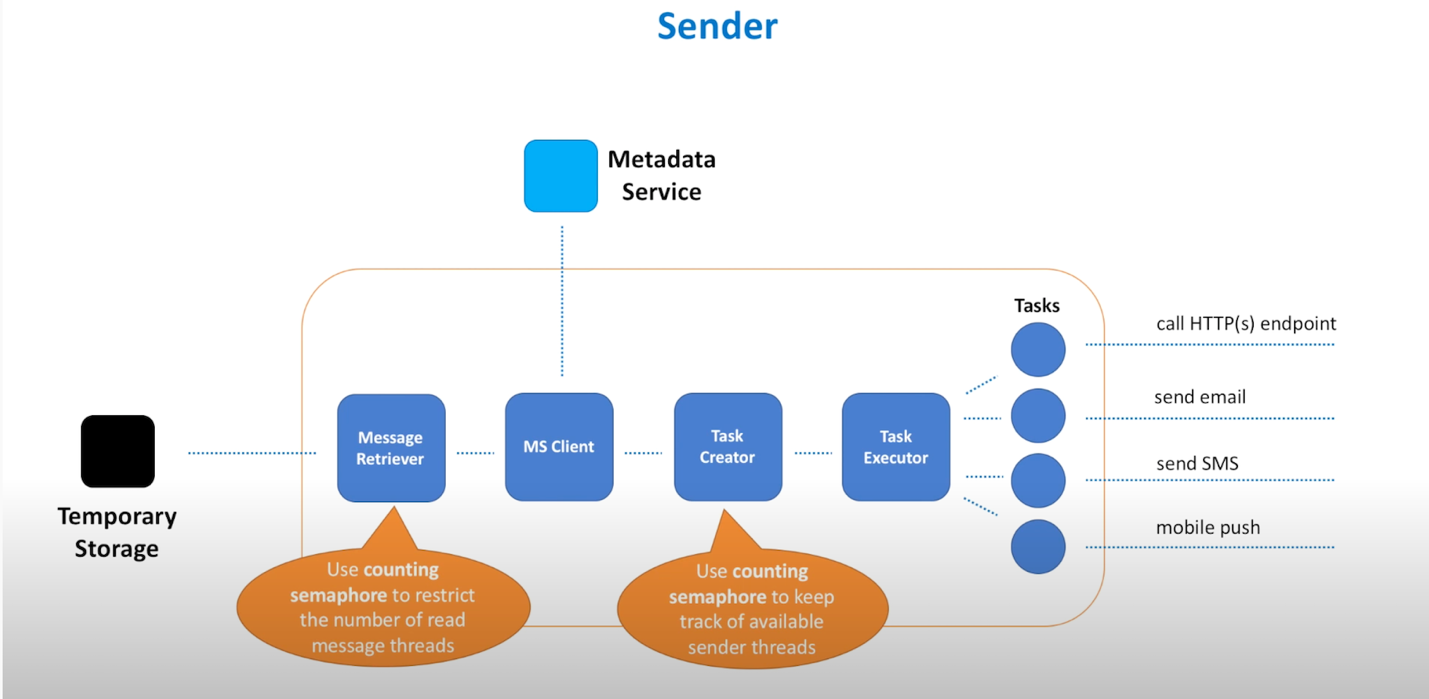
**Store Temp Messages - 那种最好？**

* Kafka & SQS both have 7-day default retention period
* Kafka & Kinesis both offer high bandwidth of streaming-processing capability
* Redis 本身有persistent 不guarantee 的问题，可能丢数据
* Casandra & DynamoDB 本身持久很好，但是速度没有Kafka/Kinesis 快
* 我的结论 - 首选 - Kafka，备选 - Cassandra （有丢数据的问题-实践证明）



**Sender 的处理**

* Sender从tmp storage读出数据，再准备send
* auto scaling sender threads - track idle message reader threads， 用semaphore 控制threads number - read thread
  + 如果有过多idle thread，不要create更多threads
  + create threads 仅当有需要的时候
* 把message sending 分成每个message 一个sender job，这样并发处理
  + 用semaphore控制sender thread的个数
  + 每个sender 一个message subscriber job
  + 每个message job 可以用micro service细分不同的work以加大效率，例如call http，send email，send SMS，mobile push etc



**Questions**

* spammer - subscriber confirm subscription request by sending a confirm request
* duplicate message - FE make sure remove dedup. However if publisher delete or retry due to network partition, it may cause dup.
* So subscriber also need to make sure no dup messages.
* Retry of delivery attempt - retry in exponential off, or better let user define undelivered policy
* Message order aka FIFO ? No, hard to implement this.
* Security - only subscriber can receive it.
* Monitoring -

**Final**

