System Design

Objectives/Goals

- Large-scale systems end to end. A strong performance: replicable to many systems at Facebook.
- Sample Questions:
    - Design a key-value store
    - Design Google search
    - Architect a world-wide video distribution system
    - Build Facebook chat
    - Google Search vs Twitter Search vs FB Search: Google's index building layer has many more components for document understanding. It would need components for extracting deep links, contact information, referrals (for page rank). On the other hand, Twitter's index building should be simpler due to small size tweets and some rich media information for the attached media. Twitter's search is head heavy. So a bulk of engineering efforts in designing their search should go to rapidly indexing new tweets and making them searchable.
- Expectations:
    - What we're looking for:
        - Can you arrive at an answer in the face of unusual constraints?
        - Can you visualize the entire problem and solution space?
        - Can you make trade-offs like consistency, availability, partitioning, performance?
        - Can you give ballpark numbers on QPS supported, # of machines needed using a modern computer?
        - How much have you thought about Facebook and some of the unique problems we face?

- A good design shows that you:
    - Clearly understand the problem
    - Propose a design for a system that breaks the problem down into components, that can be built independently, and you can drill into any piece of the design and talk about it in detail
    - Identify the bottlenecks as the system scales and understand the limitations in your design
    - Understand how to adapt the solution when requirements change
    - Draw diagrams that clearly describe the relationship between the different components in the system
    - Calculate (back-of-the-envelope) the physical resources necessary to make this system work

Key Characteristics of Distributed Systems

Cap Theorem:

Simply put, the CAP theorem demonstrates that any distributed system cannot guaranty C, A, and P simultaneously, rather, trade-offs must be made at a point-in-time to achieve the level of performance and availability required for a specific task.

- Consistency: Every read receives the most recent write or an error
- Availability: Every request receives a (non-error) response, without the guarantee that it contains the most recent write
- Partition tolerance: The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes When a network partition failure happens should we decide to cancel the operation and thus decrease the availability but ensure consistency. Proceed with the operation and thus provide availability but risk inconsistency

The CAP theorem implies that in the presence of a network partition, one has to choose between consistency and availability.

Order of Priority to keep in mind while designing a Distributed DB:

- Durability: Changes are permanent in the DB
- Availability
- Performance + Consistency: Make as much consistent as we can (given the constraint of Availability for Distributed Sys) and highly Performant.

Scalability

- The capability of a system to grow and manage increased demand.
- A system that can continuously evolve to support the growing amount of work is scalable.
- Horizontal scaling: by adding more servers into the pool of resources.
- Vertical scaling: by adding more resources (CPU, RAM, storage, etc) to an existing server. This approach comes with downtime and an upper limit.

Reliability

- Reliability is the probability that a system will fail in a given period.
- A distributed system is reliable if it keeps delivering its service even when one or multiple components fail.
- Reliability is achieved through redundancy of components and data (remove every single point of failure).

Availability

- Availability is the time a system remains operational to perform its required function in a specific period.
- Measured by the percentage of time that a system remains operational under normal conditions.
- A reliable system is available.
- An available system is not necessarily reliable.
    - A system with a security hole is available when there is no security attack.

Efficiency

- Latency: response time, the delay to obtain the first piece of data.
- Bandwidth: throughput, amount of data delivered in a given time.

Serviceability / Manageability

- Easiness to operate and maintain the system.
- Simplicity and spend with which a system can be repaired or maintained.

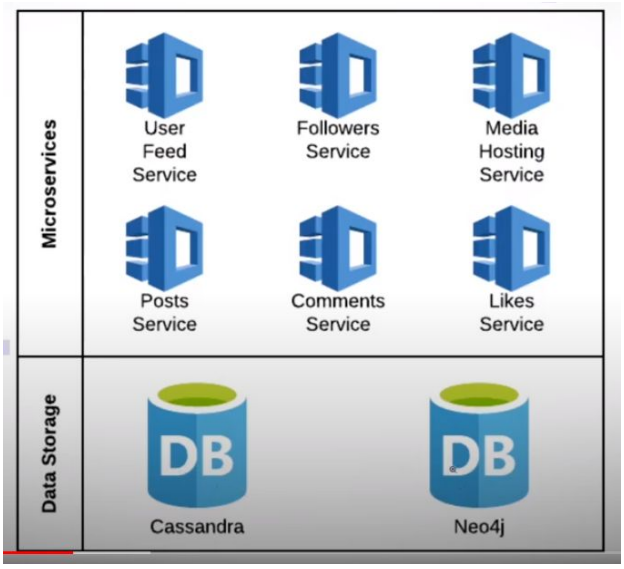
Concepts and Handy Terminologies

- **Traffic for 1B active users:**
    - Approximate QPS and num users using the Pareto principle (80-20)
    - 1 Billion active monthly users => 80% active daily users => 800 M active daily users
    - Peak traffic: 80% of this per hour => 640 M active users daily
    - 1/3rd population across the globe sleeps (24/8), so 2/3rd active => 640 *⅔ = 420 M users
    - Per second = 420M/(3600) ~116k ~150k active users per second
    - QPS = 150k/s for 1 B active monthly users
    - A typical server for JS, Android might serve 1000 Request per second
    - # Servers required ~150, accounting for duplication and fault tolerance (450 servers)

- **CAP Theorem:**
  As mentioned above

- **RDBMS vs NoSQL:**
    - https://github.com/chagri/CP/blob/master/system_design/System_Design_Databases/All_DBs_Cloud_And_Deployment_Systems.md

- **Microservices:** Kind of architecture for seamlessly working with various components/services seamlessly. Example of Instagram microservice arch:

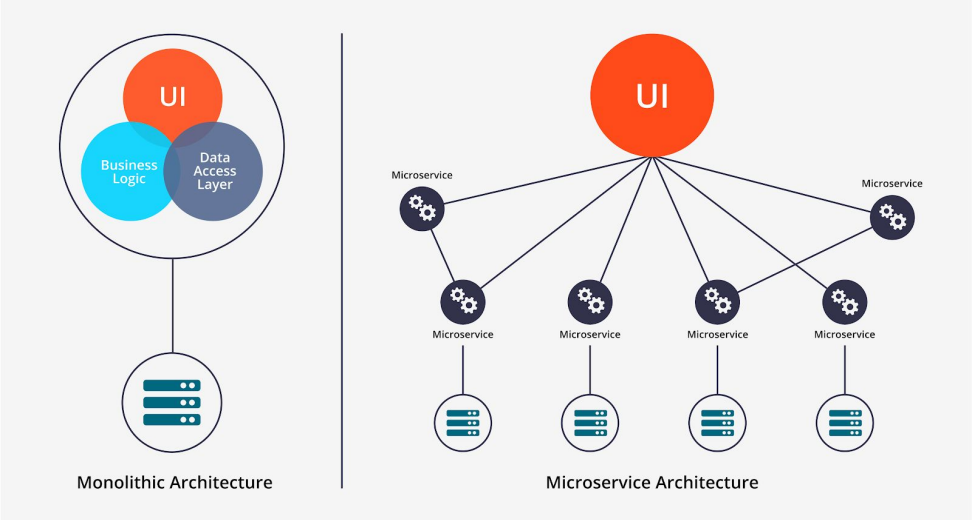


AWS Microservices: https://aws.amazon.com/microservices/

- **Microservice vs Monolithic vs Serverless:**

https://www.youtube.com/watch?v=qYhRvH9tJKw

A monolithic application is simply deployed on a set of identical servers behind a load balancer. In contrast, a microservice application typically consists of a large number of services. Each service will have multiple runtime instances.



In contrast to the microservices architecture, monolithic applications are much easier to debug and test. Since a monolithic app is a single indivisible unit, you can run end-to-end testing much faster. Simple to deploy. Another advantage associated with the simplicity of monolithic apps is easier deployment.

https://www.n-ix.com/microservices-vs-monolith-which-architecture-best-choice-your-business/

Serverless architecture is a way to build and run applications and services without having to manage infrastructure. Serverless computing allows you to run any function without worrying about the infrastructure. This means that servers, software, tools, backup, and scaling are parts of the platform. Serverless does not mean that servers are no longer involved, but developers no longer have to worry about managing them. Your application still runs on servers, but all the server management is done by a cloud provider such as AWS.

- **Load Balancer:**
  AWS Elastic Load Balancing automatically distributes incoming application traffic across multiple targets, such as Amazon EC2 instances, containers, IP addresses, and Lambda functions. It can handle the varying load of your application traffic in a single Availability Zone or across multiple Availability Zones. Elastic Load Balancing offers three types of load balancers that all feature the high availability, automatic scaling, and robust security necessary to make your applications fault-tolerant. Generally speaking, load balancers fall into three categories:
  - DNS Round Robin (rarely used): clients get a randomly-ordered list of IP addresses.

pros: easy to implement and free
cons: hard to control and not responsive, since DNS cache needs time to expire

- L3/L4 Load Balancer: traffic is routed by IP address and port. L3 is a network layer (IP). L4 is the session layer (TCP).
  pros: better granularity, simple, responsive
- L7 Load Balancer: traffic is routed by what is inside the HTTP protocol. L7 is the application layer (HTTP).

- **Memcached vs Redis/Distributed Cache:**
  Best for storing User meta-data/cache for faster access/availability.

  Redis is an in-memory data structure store, used as a database, cache, and message broker. ... While that's all that Memcached is its only the tip of the Redis iceberg. Memcached is a volatile in-memory key/value store. Redis can act like one (and do that job as well as Memcached), but it is a data structure server.

  Redis is sometimes described as “Memcached on steroids,” which is hardly surprising considering that parts of Redis were built in response to lessons learned from using Memcached. Redis has more features than Memcached and is, thus, more powerful and flexible.

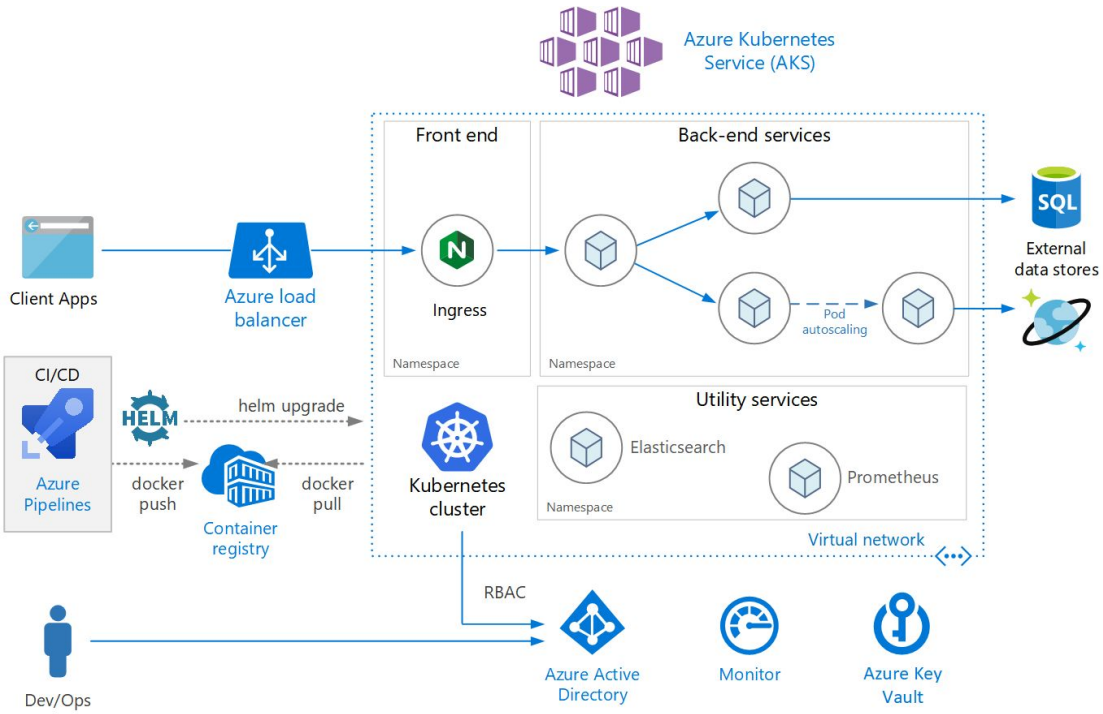
  Redis can handle up to 2^{32} keys and was tested in practice to handle at least 250 million keys per instance. Every hash, list, set and sorted set, **can** hold 2^{32} elements (~4B). **Redis** Strings are binary safe, this means that a **Redis** string can contain any kind of data, for instance, a JPEG image or a serialized Ruby object. A String value can be at **max** 512 Megabytes in **length**.

  https://www.infoworld.com/article/3063161/why-redis-beats-memcached-for-caching.html

- **Streaming DBs:** They help distribute data between several producers and many consumers (e.g. Mongo, MySQL, Redshift, Dynamo) easily. Here Apache Kafka serves as an "data" integration message bus.
  Potential Apache Kafka, AWS Kinesis

- **Kubernetes:** For CI/CD, Scaling and managing containers

  Microservice Architecture with Azure Kubernetes services:



- **HTTP 1/1.1 vs HTTP 2 vs WebSocket vs BOSH vs Long Poll HHP:**
  HTTP/2 is that it uses multiplexed streams. A single HTTP/2 TCP connection can support many bidirectional streams. These streams can be interleaved (no queuing), and multiple requests can be sent at the same time without a need to establish new TCP connections for each one. In addition, servers can now push notifications to clients via the established connection (HTTP/2 push).

  WebSocket (HTTP 2.0) is bidirectional i.e. a protocol providing full-duplex communication channels over a single TCP connection. Whereas, HTTP providing half-duplex communication, for e.g. HTTP will not be able to serve a chat message app i.e. User A sends a message to the server, and then the server needs to send to User B (the later part will not happen with HTTP). Means, the server can push information to the client (which does not allow direct HTTP). HTTP 2 can do it.

  Long Poll and BOST are other bidirectional options over HTTP.

- **REST vs RPC/GRPC:**
  - REST messages typically contain JSON. gRPC, on the other hand, accepts and returns Protobuf messages
  - GrPC uses HTTP2 while REST uses HTTP1, therefore it is faster and no need to create TCP connection every time, the same can be used for multiple requests, useful for FB, IG kind of apps with multi-service support. Other advantages:
    - The Growth of Page Size and Number of Objects per ask
    - Latency

    - Messages vs. Resources and Verbs: gRPC comes with clear interfaces and structured messages for requests and responses. This model translates directly from programming concepts like interfaces, functions, methods, and data structures. It also allows gRPC to automatically generate client libraries for you.
    - Streaming vs. Request-Response: REST request-response only, gRPC streaming as well.
    - gRPC is strongly typed i.e. more redundant but fewer bugs especially compared to JSON which completely depends on the developer.
    - More info: https://code.tutsplus.com/tutorials/rest-vs-grpc-battle-of-the-apis--cms-30711

- **Hashing Algorithms:**
    - **B62:** Based on 62 (26 Capital, 26 lower, 10 ints). 7 length string of base 62= 62^7 ~ 3.5 trillion combinations.
    - **MD5 Hash**

- **Consistent Hashing** and Load Balancing across the servers
    - https://www.youtube.com/watch?v=viaNG1zyx1g
    - https://www.youtube.com/watch?v=zaRkONvyGr8&t=556s

- **Message Queue:**
  A message queue is a form of asynchronous service-to-service communication used in serverless and microservices architectures. Messages are stored on the queue until they are processed and deleted. Each message is processed only once, by a single consumer. Message queues can be used to decouple heavyweight processing, to buffer or batch work, and to smooth spiky workloads. E.g.: Amazon Simple Queue Service (SQS) website.

- **Horizontal vs Vertical Scaling:**
    - Horizontal scaling: by adding more servers into the pool of resources.
    - Vertical scaling: by adding more resources (CPU, RAM, storage, etc) to an existing server. This approach comes with downtime and an upper limit.

- **Availability vs Reliability:**
    - Reliability:
        - Reliability is the probability that a system will fail in a given period.
        - A distributed system is reliable if it keeps delivering its service even when one or multiple components fail.
        - Reliability is achieved through redundancy of components and data (remove every single point of failure).
    - Availability:

        - Availability is the time a system remains operational to perform its required function in a specific period.
        - Measured by the percentage of time that a system remains operational under normal conditions.
        - A reliable system is available.
        - An available system is not necessarily reliable.
            - A system with a security hole is available when there is no security attack.
- **Efficiency:**
    - Latency: response time, the delay to obtain the first piece of data.
    - Bandwidth (QPS): throughput, amount of data delivered in a given time

- **Zookeeper**:
    - ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. All of these kinds of services are used in some form or another by distributed applications.

- **Paxos:** Consensus over distributed hosts, similar to Zookeeper.

- **On-device Scalability Inference, Federated Learning:**
    - Why: privacy, GDPR?
    - **Model save format for cross-platform inference:**
        - ONNX vs PMML: **ONNX** , the Open Neural Network Exchange Format is an open format that supports the storing and porting of predictive models across libraries and languages. ... **PMML** or Predictive model markup language is another interchange format for predictive models. ONNX preferred and supported for NN, Torch, TF, etc. PMML more traditional for SKlearn and ML kind of models
    - Tools:
        - TF lite
        - PyTorch Mobile: https://pytorch.org/mobile/home/
        - Core ML: For iOS only
        - API: Amazon Sagemaker

- **TCP vs UDP:** TCP is a connection-oriented reliable connection, while UDP is unreliable but fast, therefore, great for video streaming. TCP is great for sending

documents.

- **Caching over ISPs (**Internet Service Providers) via Open Connect for different regions. E.g. YouTube/Netflix caches content per region (e.g. India) and ISPs do not hit netflix.com everytime, rather netflix.in or something which fetches the data much faster. They cache popular content such as Bollywood movies/videos over these caches, which serves 90% of their traffic.

- **Synchronous vs Asynchronous (Sort of real-time vs near real-time for action/feed post-user-action):** E.g. On IG user uploads pictures (All actions related to this activity are Sync.), but pulling followers, showcasing them this new album and showcasing new content to the user is Async, which system/IG performs based on the previous activity (upload) of the user.

  In a synchronous system, operations (instructions, calculations, logic, etc.) are coordinated by one, or more, centralized clock signals. An asynchronous digital system, in contrast, has no global clock. Asynchronous systems do not depend on strict arrival times of signals or messages for reliable operation.

  The major difference between them lies in their transmission methods, i.e. Synchronous transmissions are synchronized by an external clock; whereas Asynchronous transmissions are synchronized by special signals along the transmission medium.

- **CDN and Edge:** Content delivery Network: dedicated server in the region to support data (mostly for videos like Netflix). Edge is similar to CDN with more advantage/local and dedicated line b/w edge and the consumer avoiding transfer for busy internet, therefore faster data transfer compared to CDN.

- **HTTPS =** HTTP + TLS (Transport Layer Security): More secure with TLS protocols

- **Bloom Filters, Count-min Sketch:** Space-Efficient Probabilistic Data Structures to identify if an item is part of a set or not. BF can have false positives but not false negatives, extremely space-efficient.

- **Pub-Sub and Queue:** Note that customer-facing requests through app/UI should not be directly exposed to Pub-Sub.

- **LRU Cache:** LRU stands for least recently used and the idea is to remove the least recently used data to free up space for the new data.

- Multi-threading, multi-processing, locks, synchronizations

- Kafka vs Kinesis: Git doc

- Cassandra vs Mongo vs Redis: Git doc

- Solr and Elastic Search built on top of Lucene: Highly available, scalable. Allow full-text search.

Approach

1. Questions to ask:
    a. Input:
        i. App type
        ii. List possible actions (such as Instagram: upload, like, share, comment, etc.)
        iii. Data: kind of data, GDPR/Privacy
        iv. Users: kind of users, demographics
    b. Optimizing for:
        i. Consistency, Availability, Performance, Partitioning?
        ii. ACID (Atomic, Consistent, Isolation, Durable) vs BASE (Basic Availability, Soft-State, Eventual Consistency)
    c. Traffic:
        i. # Users
        ii. # active users
        iii. QPS, Per day, per year
        iv. Lifecycle?

2. Data Model and Capacity Model:
    a. **(MUST) Schema**: Define all the tables and corresponding columns for users Different columns to store per row/sample and default columns such as the entry date/ts and expiry date.
    b. The best way to define tables and corresponding Data models is by features supported by the app.
    c. **Storage size** @ per row, per day, per year, 5 years
    d. **DB type:**

        i. RDBMS vs NoSQL (ACID, BASE)
        ii. Caching (Redis)
        iii. Streaming (Kinesis/Kafka)

3. Service:
    a. Continuous Integration and development: Kubernetes, Docker
    b. Microservices/Monolithic
    c. Synchronous, Asynchronous
    d. API: REST, GRPC
    e. Zookeeper/Paxo: Managing distributed

4. Scalability and Latency:
    a. Service side:
        i. Microservices/Monolithic
        ii. Load Balancer (service level, back-end or feature level)
        iii. Kubernetes + Docker for continuous integration and development
        iv. Zookeeper: Managing distributed
    b. Data Side:
        i. DB: NoSQL (Cassandra for fast read and writes through wide col scalability, Mongo for docs through multiple servers and no single point failure like Cassandra)
        ii. S3, Hadoop/Spark
    c. Caching:
        i. MemCache/Redis
    d. Streaming:
        i. Kinesis/Kafka

5. Model:
    a. Deployment:
        i. On Device: Look at concepts
        ii. API: gRPC/Protobuff/REST
    b. Active Learning

6. ML Design:
    a. Questions:
        i. Usecase:
        ii. Data: Annotation: Size
        iii. Metrics
        iv. Active learning
        v. Scalable: training vs inference
    b. Model:
        i. Regression, Classification, Supervised/Unsupervised
    c. Training:

i. Distributed Training: PyTorch.data-parallel, Distributed TF

Relevant Resources

- System Design Basics:
  - https://github.com/chagri/grokking-system-design/tree/master/basics
- Cracking the Coding Interview
- Glossary of terms:
  - https://www.youtube.com/watch?v=UzLMhqg3_Wc&list=PL73KFetZlkJSZ9vTDSJ1swZhe6CIYkqTL&index=3&t=0s
  -
- Grokking the system design interview:
  - Must Watch: https://github.com/lei-hsia/grokking-system-design

- Awesome Youtube Playlist:
  - https://www.youtube.com/playlist?list=PL73KFetZlkJSZ9vTDSJ1swZhe6CIYkqTL

- LC:
  - Design Youtube: https://leetcode.com/discuss/interview-question/system-design/496042/Design-video-sharing-platform-like-Youtube
  -
- CP Git:
  - All DBs summary: https://github.com/chagri/CP/blob/master/2020_practice/System_Design_Databases/All_DBs_Cloud_And_Deployment_Systems.md
  - https://github.com/chagri/CP/tree/master/system_design
- Kafka Theory and basics brush up:
  - https://learning.oreilly.com/videos/apache-kafka-series/9781789342604/9781789342604-video2_1
- Distributed system basics:
  - https://learning.oreilly.com/videos/distributed-systems-in/9781491924914
- Cassandra:
  - https://learning.oreilly.com/videos/mastering-cassandra-essentials/9781491994122

Examples

1. URL Shortening:
   a. CTCI
   b. https://www.youtube.com/watch?v=JQDHz72OA3c&list=PL73KFetZlkJSZ9vTDSJ1swZhe6CIYkqTL&index=26&t=180s

c. https://blog.codinghorror.com/url-shortening-hashes-in-practice/

> Suppose you're using something like MD5 (the GOD of HASH). MD5 takes any length string of input bytes and outputs 128 bits. The bits are consistently random, based on the input string. If you send the same string in twice, you'll get the exact same random 16 bytes coming out. But if you make even a tiny change to the input string -- even a single bit change -- you'll get a completely different output hash.
>
> So when do you need to worry about collisions? The working rule-of-thumb here comes from the birthday paradox. Basically **you can expect to see the first collision after hashing $2^{n/2}$ items, or 2^64 for MD5**.
>
> 2^64 is a big number. If there are 100 billion urls on the web, and we MD5'd them all, would we see a collision? Well no, since 100,000,000,000 is way less than 2^64:
>
> 2^{64} 18,446,744,073,709,551,616
>
> 2^{37} 100,000,000,000

2. Video Streaming:
    a. YouTube:
        i. https://leetcode.com/discuss/interview-question/system-design/496042/Design-video-sharing-platform-like-Youtube
    b. Netflix: https://www.youtube.com/watch?v=x9Hrn0oNmJM

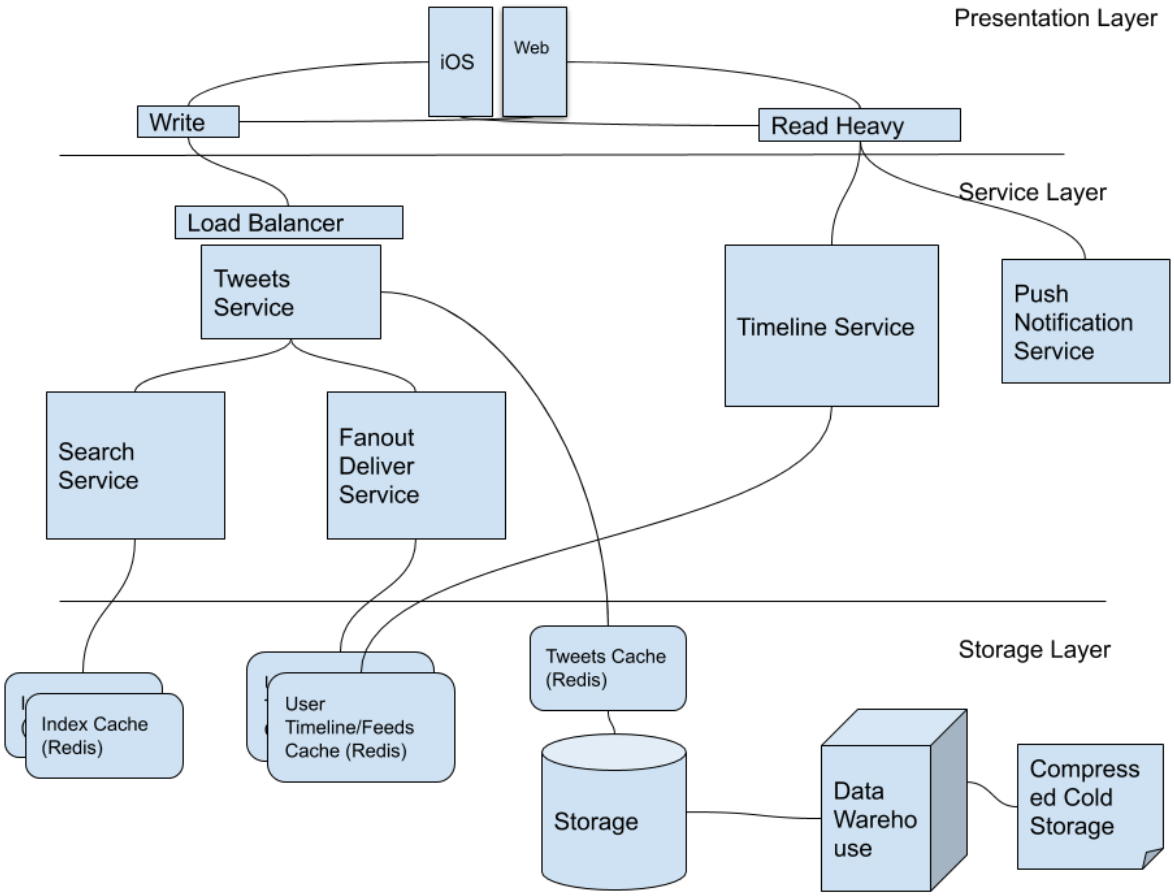
3. Design Distributed DB or Key-Value Store: https://www.youtube.com/watch?v=rnZmdmlR-2M

4. Image Video Sharing (Instagram/TikTok): https://www.youtube.com/watch?v=QmX2NPkJTKg

5. Messaging: WhatsApp, FB Messenger: https://www.youtube.com/watch?v=zKPNUMkwOJE&list=PL73KFetZlkJSZ9vTDSJ1swZhe6CIYkqTL&index=9&t=156s

6. Recommendation System: Amazon

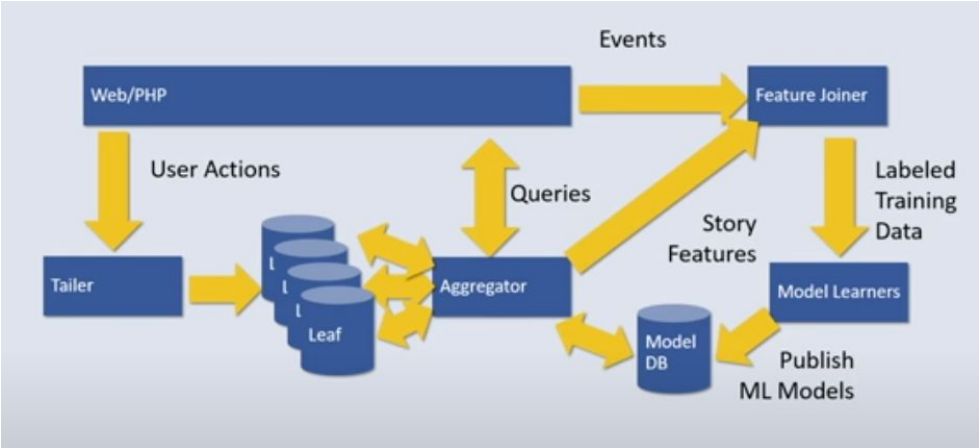
7. News Feed: FB feed, Twitter



8. FB example for News Feed:
   For each user-post pair obtain this (and calculate relevancy score by multiplying values with prob) and then showcase:

Example

Event	Probability	Value
Click	11%	1
Like	2.2%	5
Comment	0.41%	20
Share	0.054%	40
Friend	0.0062%	50
Hide	0.099%	-100
Total		0.2277



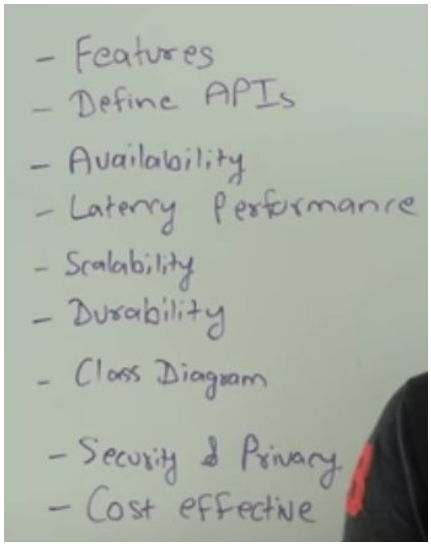
9. Search autocomplete: https://www.youtube.com/watch?v=us0qySiUsGU&list=PL73KFetZlkJSZ9vTDSJ1swZhe6CIYkqTL&index=15

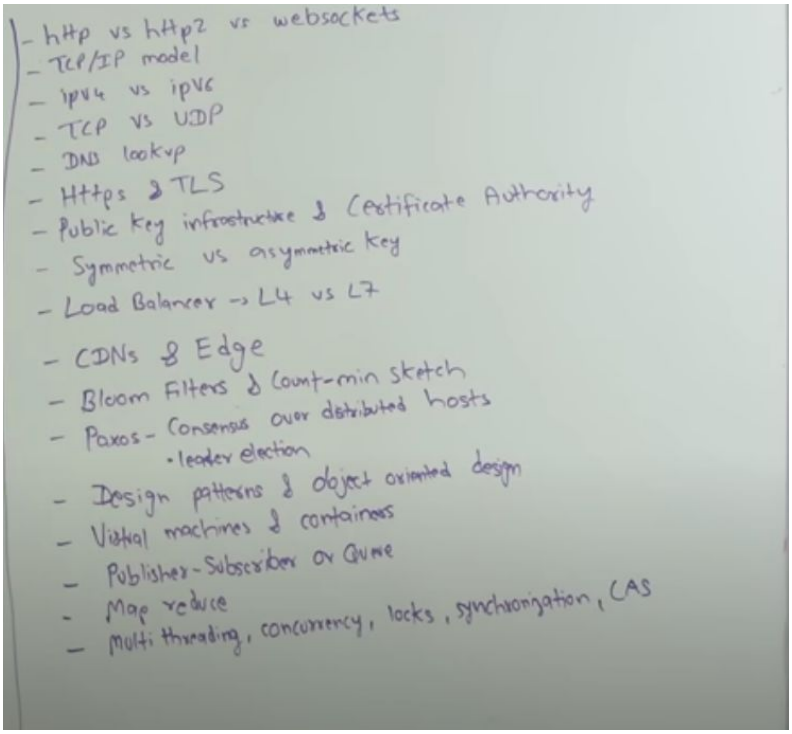
10. Logistics: Uber/Instacart

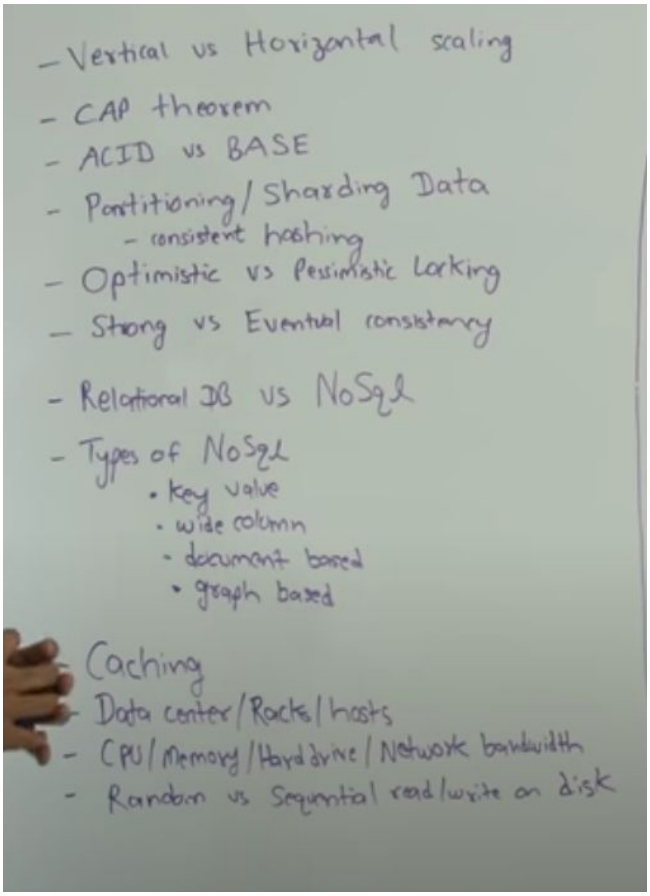
11. Logging Service for 1B users:

Glossary

https://www.youtube.com/watch?v=UzLMhqg3_Wc&list=PL73KFetZlkJSZ9vTDSJ1swZhe6CIYkqTL&index=3&t=0s


- Features
- Define APIs
- Availability
- Latency Performance
- Scalability
- Durability
- Class Diagram
- Security & Privacy
- Cost effective


- http vs http2 vs websockets
- TCP/IP model
- ipv4 vs ipv6
- TCP vs UDP
- DNS lookup
- Https & TLS
- Public key infrastructure & Certificate Authority
- Symmetric vs asymmetric key
- Load Balancer -> L4 vs L7
- CDNs & Edge
- Bloom Filters & Count-min sketch
- Paxos - Consensus over distributed hosts
  - leader election
- Design patterns & object oriented design
- Virtual machines & containers
- Publisher-Subscriber or Queue
- Map reduce
- multi threading, concurrency, locks, synchronization, CAS



Things to consider

- Features
- API
- Availability
- Latency
- Scalability
- Durability
- Class Diagram
- Security and Privacy
- Cost-effective

Concepts to know

- Vertical vs horizontal scaling
- CAP theorem
- ACID vs BASE
- Partitioning/Sharding
- Consistent Hashing
- Optimistic vs pessimistic locking
- Strong vs eventual consistency

- RelationalDB vs NoSQL
- Types of NoSQL
    - Key value
    - Wide column
    - Document-based
    - Graph-based
- Caching
- Data center/racks/hosts
- CPU/memory/Hard drives/Network bandwidth
- Random vs sequential read/writes to disk
- HTTP vs http2 vs WebSocket
- TCP/IP model
- ipv4 vs ipv6
- TCP vs UDP
- DNS lookup
- Http & TLS
- Public key infrastructure and certificate authority(CA)
- Symmetric vs asymmetric encryption
- Load Balancer
- CDNs & Edges
- Bloom filters and Count-Min sketch
- Paxos
- Leader election
- Design patterns and Object-oriented design
- Virtual machines and containers
- Pub-sub architecture
- MapReduce
- Multithreading, locks, synchronization, CAS(compare and set)

Tools

- Cassandra
- MongoDB/Couchbase
- Mysql
- Memcached
- Redis
- Zookeeper
- Kafka
- NGINX
- HAProxy
- Solr, Elastic search
- Amazon S3
- Docker, Kubernetes, Mesos
- Hadoop/Spark and HDFS

ML Design

Objective/Goals

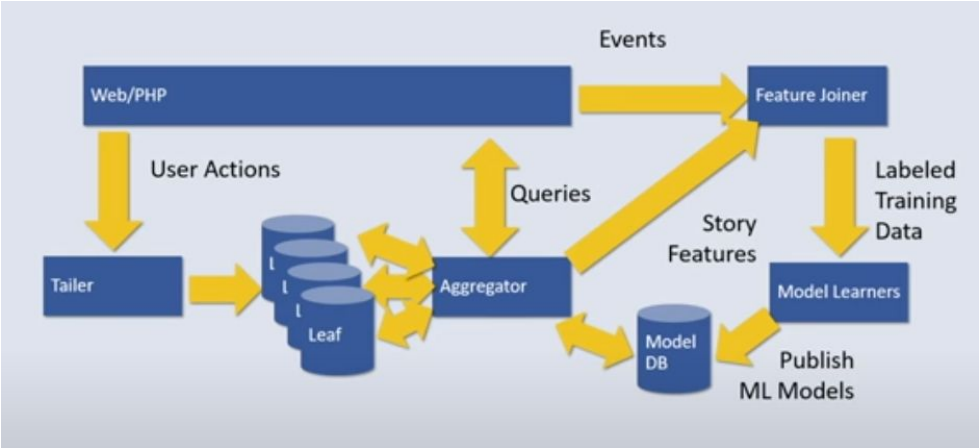
- A good design will touch on the following different components:
    - Problem formulation
    - Optimization function
    - Supervision signal
    - Feature engineering
        - - Data source
        - - Representation
        - - For example, the number of likes is a good idea but a better feature would involve normalization, smoothing, and bucketing.
    - Model architecture
    - Evaluation metrics
    - Deployment (A/B testing)
    - End to end:
        - What will you do after you train the model and the model does not perform well?
        - How do you go about debugging an ML model?
        - How do you evaluate and continuously deploy an ML model?
        - Weights and Biases tool or any other tool?
        - TensorBoard

Concepts

- Ranking:
  FB example for News Feed:
  For each user-post pair obtain this (and calculate relevancy score by multiplying values with prob) and then showcase:

Example

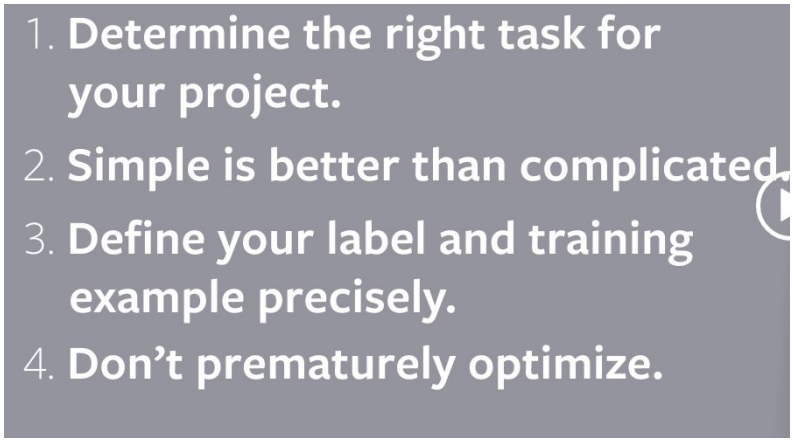
Event	Probability	Value
Click	11%	1
Like	2.2%	5
Comment	0.41%	20
Share	0.054%	40
Friend	0.0062%	50
Hide	0.099%	-100
Total		0.2277



Approach

https://research.fb.com/blog/2018/05/the-facebook-field-guide-to-machine-learning-video-series/

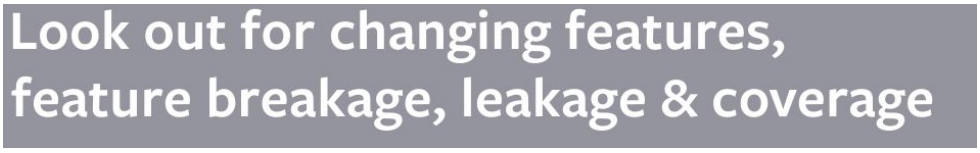
1. Problem formulation (Data, Target, Success Metrics, Type):



   a. Get data(input, output), application
   b. Get Target
   c. Success Metrics: NDCG, Precision, Recall, Regression, RMSE,
   d. Type: Ranking, Supervised (Classification, Regression), Semi, Un

2. Feature Engineering:
   a. Data:

   b. Types of features: Categorical, Continuous, Derived
   c. Normalization, smoothing, bucketing, Scaling
   d. Edge cases, sparsity, sampling, seasonality
   e. Representation
   f. Special featurization techniques: Embeddings, categorization, etc.



3. Training:
   a. Model architecture: Interpretable?
   b. Cross-Validation

c. Baseline Model:
    i. Get a simple baseline like random (normalized entropy), or general likelihood of click/not-click, average click rate.
    ii. Or training on 1st time clickers, i.e. with less # features with context/history, i.e. simpler inputs.

4. Optimization function:
    a. Based on the type and metrics: Ranking loss, RMSE, Cross-Entropy
    b. Define based on performance wrt baseline model (RMSE in test data wrt avg CTR, etc.)

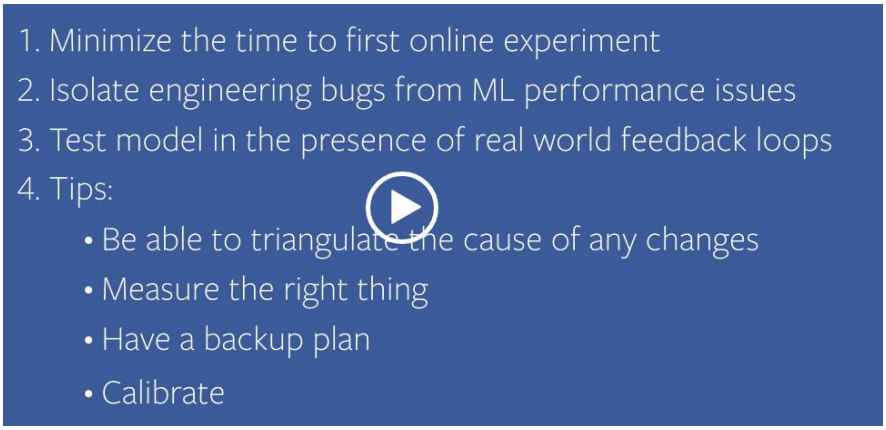
5. Evaluation and Deployment:
    a. Offline:
        i. Cross-Validation
        ii. Progressive Evaluation: Train on the older batch, evaluate the latest batch of data.
        iii. **Calibration** (Sanity check for overfitting): On train/test set calculate this for sanity i.e. average pre(generalization):

$$\text{Calibration} = \frac{\text{Sum of predictions}}{\text{Sum of labels}}$$

https://medium.com/analytics-vidhya/calibration-in-machine-learning-e7972ac93555

        iv. Convert Binary to multi-class and evaluate how and where the performance is coming from, also to debug.

    b. Online: A/B Test:



Way to test whether A/B test is configured properly is by comparing control with control and seeing exactly similar performance. If that's not the case, then the

experiment is not fairly setup.

c. Evaluate on devices, demographics, days/events, and different experiences
d. Statistical Tests
e. Active Learning, Real-time training (evaluate in latest data)
    i. Logging the data/activity during deployment.

6. Hybrid:
    a. Model + Rules (such as cache and latest seen items/interests)

Examples

1. Scalable Collaborative Filtering for FB Ads:
   https://engineering.fb.com/core-data/recommending-items-to-more-than-a-billion-people/
2. Pinterest Recommendation System using CNNs:
   Efficient convolutional network for recommender systems
   Patent: Look at images
3.