Architecture says why … why am I building this, who is the customer (not just who is paying for it; all of the stakeholders), what do they really want, what is the mission, what constitutes success, what are the constraints, and so on.

High level says what … what does the solution look like; components, capabilities, information flows, use cases, and how do they all interrelate; clearly define the abstract solution.

Low level says how … define specific solutions; how do I assemble a system of real components that implements all the whats and optimally satisfies all the whys.

Ricky Ho in Scalable System Design Patterns has created a great list of scalability patterns along with very well done explanatory graphics. A summary of the patterns are:

1. **Load Balancer** - a dispatcher determines which worker instance will handle a request based on different policies.
2. **Scatter and Gather** - a dispatcher multicasts requests to all workers in a pool. Each worker will compute a local result and send it back to the dispatcher, who will consolidate them into a single response and then send back to the client.
3. **Result Cache** - a dispatcher will first lookup if the request has been made before and try to find the previous result to return, in order to save the actual execution.
4. **Shared Space** - all workers monitors information from the shared space and contributes partial knowledge back to the blackboard. The information is continuously enriched until a solution is reached.
5. **Pipe and Filter** - all workers connected by pipes across which data flows.
6. **MapReduce** - targets batch jobs where disk I/O is the major bottleneck. It use a distributed file system so that disk I/O can be done in parallel.
7. **Bulk Synchronous Parallel** - a lock-step execution across all workers, coordinated by a master.
8. **Execution Orchestrator** - an intelligent scheduler / orchestrator schedules ready-to-run tasks (based on a dependency graph) across a clusters of dumb workers.

Right from the outset, before we begin designing any service there are a few questions which pop up in our minds like:

1. What are the primary use cases or the features of your product?
2. How much traffic are you estimating for your service to run up against?
3. What are the primary design goals?
4. Which database SQL or NoSQL will serve your requirement best?

HLD

1. Whatsapp design : https://www.youtube.com/watch?v=5m0L0k8ZtEs
2. Tiny URL : https://www.youtube.com/watch?v=fMZMm_0ZhK4
3. Twitter : https://www.youtube.com/watch?v=KmAyPUv9gOY
4. Cricbuzz : https://www.youtube.com/watch?v=exSwQtMxGd4
5. Chess : https://www.youtube.com/watch?v=yBsWza2039o
6. Netflix : https://www.youtube.com/watch?v=psQzyFfsUGU

7. Uber : https://www.youtube.com/watch?v=umWABit-wbk
8. Book my show : https://www.youtube.com/watch?v=lBAwJgoO3Ek

1. Creational Design Patterns
    a. Singleton Pattern
        i. Eager Initialization
        ii. Static block initialization
        iii. Lazy Initialization
        iv. Thread Safe Singleton
        v. Bill Pugh Singleton
    b. Factory Pattern (Computer Factory)
    c. Abstract Factory Pattern (Factory of factories)
    d. Builder Pattern (Computer graphics, call inner class static to build)
    e. Prototype Pattern (employee list, make class cloneable)
2. Structural Design Patterns
    a. Adapter Pattern (mobile adapter, interface to get different volt)
    b. Composite (Shapes, Triangle, Circle)
    c. Proxy Pattern (run command interface, check permission before running command)
    d. Flyweight Pattern (Shapes, Line, Oval, hashmap)
    e. Facade Pattern (Generate HTML/PDF report based on DB type)
3. Behavioral Design Patterns
    a. Template Method Pattern (House building, implement common function in abstract class)
    b. Mediator Pattern (Air traffic, implement send and receive interface)
    c. Chain of Command (try catch)
    d. Observer Pattern (Topic listening, implement interface for register unregister, publish, listen)
    e. Strategy Pattern (custom sort, shopping cart multiple payment method)
    f. Command Pattern (Unix, windows file open/write/closed commands)
    g. Visitor Pattern (Banana, Book, price calculation)

Steps to solve **LLD**:

1. problem statement
2. use cases analysis & design
3. identifying objects
4. identifying interactions among objects
5. identifying attributes of objects
6. refining with hierarchy
7. at the end of sixth step, we shall get good class diagram,
8. after we can proceed for implementation with functional modeling

Single responsibility principle : a class should have only a single responsibility (i.e. changes to only one part of the software's specification should be able to affect the specification of the class). (For Animal, we should have many subclasses for each of specie)

1. Every function / class / module / unit-of-code should have a single, well-defined responsibility
2. Another way to say it - any unit-of-code should have exactly 1 reason to change
3. If we find that some code is serving multiple purposes - break it down into smaller, individual pieces - each with it's own well defined responsibility

Example : Have different classes for different animals.

Open/closed principle : “software entities … should be open for extension, but closed for modification.” (Think from library POV, a Bird class should be free to be extended by the client user)

1. Your code should be open for extension, however, it should be closed for modification even people who don't have access to your code should be able to extend your code!

Example : If peacock is not defined by library class, client can extend Bird class and define it.

Liskov substitution principle : “objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.” See also design by contract. (Since all birds can’t fly, we can put it in a separate interface from bird ABC. Only implement for birds which fly.)

1. Any functionality in the parent class, must also work for all child classes
2. theoretical: any Parent class object must be replacable for any child class object
3. any extension to a class should not break the existing class

Example: Since Kiwi can’t fly. We should have an interface for Ifly. When we can call a function, which is supposed to return birds which can fly, we should have return type as IFly not Bird.

Interface segregation principle : “many client-specific interfaces are better than one general-purpose interface.” (Since Shaktiman don’t have wings, we can have interface for birds which have wings)

1. Keep your interfaces minimal
2. No code (the clients/users of your code) should not be forced to implement methods that they don't need

Example : Since Shaktiman don’t have wings. We can have 2 types of interfaces : ICanFly and ICanFlayLikeBird.

Dependency inversion principle : one should “depend upon abstractions, [not] concretions.”(Let there be a single cage class, the properties of the cage class are interfaces. We pass objects to the cage constructor. Now cage will behave independently of implementation of bowl and door interface)

1. Instead of creating your own dependencies, you let your client provide (inject) the dependencies into you

Example : We should not have 2 classes for Cage for Bird and BigCat. Instead we should have 1 class Cage with abstract properties like bowl and door defined as abstract. When we instantiate the object, we can pass the corresponding objects accordingly.

Advantages : **SOLID** Principles are certain standards to be followed by software developers.

- They help in identifying faults that lead to **bad** software designs.
- They help developers to handle **complex** designs.
- They make the code logical and **readable**.
- They help the developers to **change** software without altering the other. **de-coupled**

Low Level Design - how to write good code

1. Object Oriented Programming
2. SOLID Principles
3. Design Pattern
    a. Singleton
    b. Builder
        i. language specific - yes for java, but no for python
    c. Factory
    d. ...
4. Database Schema Design
    a. Indexes
    b. Normalize
    c. Optimize queries
5. ER-diagrams / Class diagram
6. REST API design

1. REASONS TO USE A SQL DATABASE
    a. You need to ensure ACID compliancy (Atomicity, Consistency, Isolation, Durability).
    b. Your data is structured and unchanging.
2. REASONS TO USE A NOSQL DATABASE

a. Storing large volumes of data that often have little to no structure.
b. Making the most of cloud computing and storage.
c. Rapid development

NoSQL

Advantages 4

- Non-Relational means table-less:
- Mostly Open Source and Low-Cost:
- Easier scalability
- No need to develop a detailed database model:

Disadvantages 3

- Community not as well defined:
- Lack of reporting tools:
- Lack of standardization:

Quick Sort vs Merge Sort

Auxiliary Space : Mergesort uses extra space, quicksort requires little space and exhibits good cache locality. Quick sort is an in-place sorting algorithm. In-place sorting means no additional storage space is needed to perform sorting. Merge sort requires a temporary array to merge the sorted arrays and hence it is not in-place giving Quick sort the advantage of space.

Worst Cases : The worst case of quicksort **O(n2)** can be avoided by using **randomized quicksort.** It can be easily avoided with high probability by **choosing the right pivot**. Obtaining an average case behavior by choosing the right pivot element makes it improvise the performance and become as efficient as Merge sort.

Locality of reference : Quicksort in particular exhibits good cache locality and this makes it faster than merge sort in many cases like in virtual memory environments.

Merge sort is better for large data structures: Mergesort is a stable sort, unlike quicksort and heapsort, and can be easily adapted to operate on linked lists and very large lists stored on slow-to-access media such as disk storage or network attached storage. Refer this for details

what actually process happens in background when we access website from browser
http://edusagar.com/articles/view/70/What-happens-when-you-type-a-URL-in-browser

Step 1. URL is typed in the browser.
Step 2. If requested object is in browser cache and is fresh, move on to Step 8.
Step 3. DNS lookup to find the ip address of the server
Check browser cache, Check OS cache, Router Cache, ISP cache
Step 4. Browser initiates a TCP connection with the server.
Step 5. Browser sends a (get/post) HTTP request to the server.
header content
User-Agent header specifies the browser properties,
Accept-Encoding headers specify the type of responses it will accept.
Connection header tells the server to keep open the **TCP** connection established here.
The request also contains **Cookies**,
Step 6. Server handles the incoming request
HTTP request made from browsers are handled by a special software running on server - commonly known as web servers e.g. Apache, IIS etc. Web server passes on the request to **the proper request handler** - a program written to handle web services e.g. PHP, ASP.NET, Ruby, Servlets
Step 7. Browser receives the HTTP response
Following is a very brief summary of what a status code denotes:
1xx indicates an informational message only
2xx indicates success of some kind
3xx redirects the client to another URL
4xx indicates an error on the client's part
5xx indicates an error on the server's part
Content-Type tells the type of the content the browser has to show,
Content-length tells the number of bytes of the response.
Using the Content-Encoding header's value, browsers can decode the blob data present at the end of the response.
Step 8. Browsers displays the html content
Rendering of html content is also done in phases.
If the html response contains an image in the form of img tags such as , browser will send a
HTTP GET request to the server to fetch the image following the complete set of steps which we have seen till now.
Step 9. Client interaction with server
Once a html page is loaded, there are several ways a user can interact with the server.
Step 10. AJAX queries
Another form of client interaction with server is through AJAX(**Asynchronous JavaScript And XML**) requests.

This is an asynchronous GET/POST request to which server can send a response back in a variety of ways - json,
xml, html etc. AJAX requests doesn't hinder the current view of the webpage and work in the background.
Because of this, one can dynamically modify the content of a webpage by calling an AJAX request and updating the web elements using Javascript.

7. Difference between multiprocessing and multithreading? How it works ? 4 points
A thread is a stream of instructions within a process. Each thread has its own instruction pointer, set of registers and stack memory. The virtual address space is process specific, or common to all threads within a process. So, data on the **heap** can be readily accessed by **all threads**, for good or ill.
Multi-threading is a more "light weight" form of concurrency: there is **less context per thread** than per process. As a result **thread lifetime, context switching and synchronisation costs are lower.** The shared address space (noted above) means data sharing requires no extra work.
Multi-processing has the opposite benefits**. Since processes are insulated from each other by the OS, an error in one process cannot bring down another process.** Contrast this with multi-threading, in which an error in one thread can bring down all the threads in the process. Further, **individual processes may run as different users and have different permissions.** When we open a new tab in browser a new process is created. So that 1 tab crash should not stop browser.

2. What is mutex and semaphore ?
Semaphore: Use a semaphore when you (thread) want to **sleep** t**ill some other thread tells you to wake up.** Semaphore 'down' happens in one thread (producer) and semaphore 'up' (for same semaphore) happens in another thread (consumer) e.g.: In producer-consumer problem, producer wants to sleep till at least one buffer slot is empty - only the consumer thread can tell when a buffer slot is empty.

Mutex: Use a mutex when you (thread) want to execute code that should **not be executed by any other thread at the same time**. Mutex 'down' happens in one thread and mutex 'up' *must* happen in the same thread later on. e.g.: If you are deleting a node from a global linked list, you do not want another thread to muck around with pointers while you are deleting the node. When you acquire a mutex and are busy deleting a node, if another thread tries to acquire the same mutex, it will be put to sleep till you release the mutex.

Spinlock: Use a spinlock when you really want to **use a mutex but your thread is not allowed to sleep**. e.g.: An interrupt handler within OS kernel must never sleep. If it does the system will freeze / crash. If you need to insert a node to globally shared linked list from the interrupt handler, acquire a spinlock - insert node - release spinlock.

A mutex is a mutual **exclusion *semaphore***, a special variant of a semaphore that **only allows one locker at a time** and whose ownership restrictions may be more stringent than a normal semaphore.

A semaphore, on the other hand, **has a count and can be locked by that many lockers concurrently**. **And it may not have a requirement that it be released by the same thread that claimed it**

Memory management keeps track of each and every memory location, regardless of either it is allocated to some process or it is free. It checks how much memory is to be **allocated to processes**. It **decides which process** will get memory at what time. It tracks whenever some memory gets **freed or unallocated** and correspondingly it updates the status.

1. **Symbolic addresses :** The addresses used in a source code. The variable names, constants, and instruction labels are the basic elements of the symbolic address space.
2. **Relative addresses :** At the time of compilation, a compiler converts symbolic addresses into relative addresses.
3. **Physical addresses :** The loader generates these addresses at the time when a program is loaded into main memory.

At the time of loading, with **static loading**, the absolute program (and data) is loaded into memory in order for execution to start.

If you are using **dynamic loading**, dynamic routines of the library are stored on a disk in relocatable form and are loaded into memory only when they are needed by the program.

Static vs Dynamic Linking

As explained above, when static linking is used, the linker combines all other modules needed by a program **into a single executable program to avoid any runtime dependency**.

When dynamic linking is used, it is **not required to link the actual module or library with the program,** rather a **reference** to the dynamic module is provided at the time of compilation and linking. **Dynamic Link Libraries (DLL)** in Windows and **Shared Objects(SO)** in Unix are good examples of dynamic libraries.

Swapping

Swapping is a mechanism in which a process can be swapped temporarily out of main memory (or move) to secondary storage (disk) and make that memory available to other processes. At some later time, the system swaps back the process from the secondary storage to main memory.

Though performance is usually affected by swapping process but it helps in running multiple and big processes in parallel and that's the reason **Swapping is also known as a technique for memory compaction**.

Fragmentation

As processes are loaded and removed from memory, the free memory space is broken into little pieces. It happens after sometimes that processes cannot be allocated to memory blocks considering their small size and memory blocks remains unused. This problem is known as Fragmentation.

first fit next fit best fit

External fragmentation

Total memory space is enough to satisfy a request or to reside a process in it, but it is not contiguous, so it cannot be used.

Internal fragmentation

Memory block assigned to process is bigger. Some portion of memory is left unused, as it cannot be used by another process.

Paging

Paging is a memory management technique in which process address space is broken into blocks of the same size called **pages** (size is power of 2, between 512 bytes and 8192 bytes). The size of the process is measured in the number of pages.

Similarly, main memory is divided into small fixed-sized blocks of (physical) memory called frames and the size of a frame is kept the same as that of a page to have optimum utilization of the main memory and to avoid external fragmentation.

Address Translation

Page address is called **logical address** and represented by **page number** and the **offset**.

Logical Address = **Page number** + page offset

Frame address is called **physical address** and represented by a **frame number** and the **offset**.

Physical Address = **Frame number** + page offset

A data structure called **page map table** is used to keep track of the relation between a page of a process to a frame in physical memory.

Here is a list of advantages and disadvantages of paging −

- **Paging reduces external fragmentatio**n, but still suffer from internal fragmentation.
- Paging is **simple** to implement and assumed as an efficient memory management technique.
- Due to equal size of the pages and frames, swapping becomes very easy.
- **Page table requires extra memory space**, so may not be good for a system having small RAM.

Segmentation is a memory management technique in which each job is divided into several segments of different sizes, one for each module that contains pieces that perform related functions.

Virtual memory
A computer can address more memory than the amount physically installed on the system. This extra memory is actually called virtual memory and it is a section of a hard disk that's set up to emulate the computer's RAM.

Advantages

Following are the advantages of **Demand Paging −**

- **Large virtual memory.**
- More e**fficient use of memory.**
- There is no limit on degree of multiprogramming.

Disadvantages

- Number of **tables** and the amount of **processor overhead** for handling page interrupts are greater than in the case of the simple paged management techniques.

First In First Out (FIFO) algorithm

- Oldest page in main memory is the one which will be selected for replacement.

Optimal Page algorithm (future)

- Replace the page **that will not be used for the longest period of time.** Use the time when a page is to be used.

Least Recently Used (LRU) algorithm

- Page which has not been used for the longest time in main memory is the one which will be selected for replacement.

ACID:

Atomicity : Each transaction is considered as one unit and either runs to completion or is not executed at all. It involves the following two operations.

—**Abort**: If a transaction aborts, changes made to the database are not visible.

—**Commit**: If a transaction commits, changes made are visible.

Atomicity is also known as the 'All or nothing rule'.

Consistency: This means that integrity constraints must be maintained so that the database is consistent before and after the transaction. It refers to the correctness of a database. Referring to the example above, The total amount before and after the transaction must be maintained.

Total before T occurs = 500 + 200 = 700.

Total after T occurs = 400 + 300 = 700.

Therefore, the database is consistent. Inconsistency occurs in case T1 completes but T2 fails. As a result, T is incomplete.

Isolation: This property ensures that multiple transactions can occur concurrently without leading to the inconsistency of the database state. Transactions occur independently without interference. Changes occurring in a particular transaction will not be visible to any other transaction until that particular change in that transaction is written to memory or has been committed. This property ensures that the execution of transactions concurrently will result in a state that is equivalent to a state achieved if these were executed serially in some order.

Durability: This property ensures that once the transaction has completed execution, the updates and modifications to the database are stored in and written to disk and they persist even if a system failure

occurs. These updates now become permanent and are stored in non-volatile memory. The effects of the transaction, thus, are never lost.

thrashing occurs when a computer's virtual memory resources are overused, leading to a constant state of paging and page faults, inhibiting most application-level processing. It causes the performance of the computer to degrade or collapse. The situation can continue indefinitely until the user closes some running applications or the active processes free up additional virtual memory resources.

To fix thrrashing:

1. **Adjust the swap file size**: If the system swap file is not configured correctly, disk thrashing can also happen to you.
2. **Increase the amount of RAM**: As insufficient memory can cause disk thrashing, one solution is to add more RAM to the laptop. With more memory, your computer can handle tasks easily and don't have to work excessively. Generally, it is the best long-term solution.
3. **Decrease the number of applications running on the computer**: If there are too many applications running in the background, your system resource will consume a lot. And the remaining system resource is slow that can result in thrashing. So while closing, some applications will release some resources so that you can avoid thrashing to some extent.
4. **Replace programs**: Replace those programs that are heavy memory occupied with equivalents that use less memory.

CAP Theorem : states that any **distributed** data store can provide **only two** of the following three guarantees:

1. Consistency : Every read receives the most recent write or an error.
2. Availability : Every request receives a (non-error) response, without the guarantee that it contains the most recent write.
3. 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, it must be decided whether to do one of the following:

1. cancel the operation and thus decrease the availability but ensure consistency
2. proceed with the operation and thus provide availability but risk inconsistency.

What are good coding practices?

- Well-structured codes must be used.
- Codes must adhere to quality standards.
- Complex codes should be avoided to reduce errors. Simple and readable codes must be used.
- Focus on solving a problem naturally instead of fitting design patterns into the code forcefully.

Association (Single line**)** is a relation between two separate classes which establishes through their Objects. Association can be one-to-one, one-to-many, many-to-one, many-to-many. In Object-Oriented programming, an Object communicates to another object to use functionality and services provided by that object. **Composition** and **Aggregation** are the two forms of association. (Bank and employee)

Association (empty diamond in parent, college, student, teacher)**:**

- It represents Has-A's relationship.
- It is a **unidirectional association** i.e. a one-way relationship. For example, a department can have students but vice versa is not possible and thus unidirectional in nature.
- In Aggregation, **both entries can survive individually** which means ending one entity will not affect the other entity.

Composition (filled diamond in parent, vehicle, engine, tyre)

- It represents **part-of** a relationship.
- In composition, both entities are dependent on each other.

- When there is a composition between two entities, the composed object **cannot exist** without the other entity.

ZooKeeper is an open source Apache project that provides a centralized service for providing configuration information, naming, synchronization and group services over **large clusters in distributed systems**. The goal is to make these systems easier to manage with improved, more reliable propagation of changes. (https://www.conduktor.io/kafka/zookeeper-with-kafka/)

ZooKeeper is utilized by several open-source projects to provide a highly reliable control plane for **distributed coordination** of clustered applications through a **hierarchical key-value store**. The suite of services provided by ZooKeeper include **distributed configuration services**, **synchronization services**, **leadership election services**, and a **naming registry**.

The Kafka team decided to use Zookeeper for this purpose.

Zookeeper is used for metadata management in the Kafka world. For example:

- Zookeeper keeps track of which brokers are part of the Kafka cluster
- Zookeeper is used by Kafka brokers to determine which broker is the leader of a given partition and topic and perform leader elections
- Zookeeper stores configurations for **topics** and permissions
- Zookeeper sends notifications to Kafka in case of changes (e.g. new topic, broker dies, broker comes up, delete topics, etc.…)

A Zookeeper cluster is called an ***ensemble***. It is recommended to operate the ensemble with an odd number of servers, e.g., 3, 5, 7, as a strict majority of ensemble members (a quorum) must be working in order for Zookeeper to respond to requests. Zookeeper has a leader to handle writes, the rest of the servers are followers to handle reads.

The **MD5 message-digest algorithm** is a widely used hash function producing a 128-bit hash value. MD5 can be used as a **checksum** to verify **data integrity** against unintentional corruption. Historically it was

widely used as a cryptographic hash function; however it has been found to suffer from extensive vulnerabilities. It remains suitable for other non-cryptographic purposes, for example for determining the partition for a particular key in a partitioned database, and may be preferred due to lower computational requirements than more recent Secure Hash Algorithms.

How Does OAuth 2.0 Work?

At the most basic level, before OAuth 2.0 can be used, the Client must acquire its own credentials, a _client id _ and *client secret*, from the Authorization Server in order to identify and authenticate itself when requesting an Access Token.

Using OAuth 2.0, access requests are initiated by the Client, e.g., a mobile app, website, smart TV app, desktop application, etc. The token request, exchange, and response follow this general flow:

1. The Client requests authorization (authorization request) from the Authorization server, supplying the **client id** and **secret** to as identification; it also provides the scopes and an endpoint URI (redirect URI) to send the Access Token or the Authorization Code to.
2. The Authorization server authenticates the Client and verifies that the requested scopes are permitted.
3. The Resource owner interacts with the Authorization server to grant access.
4. The Authorization server redirects back to the Client with either an Authorization Code or Access Token, depending on the grant type, as it will be explained in the next section. A Refresh Token may also be returned.
5. With the Access Token, the Client requests access to the resource from the Resource server.

Why is an API Gateway used? : An API Gateway serves the following functions:

1. With **authentication** it prevents overuse and abuse of your APIs
2. **Analytics** and monitoring tools can be configured on the gateway itself.
3. It provides a **single endpoint** to external users irrespective of the number of microservices running within your system.

4. Users don't need to change anything in case of **refactoring**, addition/removal of resources etc as long as the contract remains the same.
5. It also acts as a **traffic controller** by forming a single entry point for all requests.

It generally has the following capabilities:

- Developer Portal: This consists of API documentation, testing sandbox, onboarding manuals etc. that helps other developers to use their APIs
- API Gateway: This is used to provide a single abstracted layer to the external users
- API Lifecycle Management: This manages the design and implementation of all APIs, until it's deprecated.
- Analytics: This helps in deriving insights from the usage and performance of APIs, which can be used as valuable information when designing improvements and extensions.
- Monetisation: This helps in generating revenue from your APIs. Contracts can be defined on multiple parameters like scale, usage, number of users etc.

Disadvantages of API Gateway

1. **Latency**: The added network hop to the architecture accounts for an increase in latency throughout the system.
2. **SPoF**: The API Gateway being the single entry point for all requests acts as a Single Point of Failure(SPoF). This can be mitigated to some end by having multiple API Gateways and split the calls using Load Balancer and Elastic IP.
3. **Added Complexity**: The API Gateway can get complex when the end users can be of various kinds like IoS, Android, Web, etc. In this case we can add multiple configurations for different entry points. This architecture is also known as "Backend for Frontend" pattern.