



High Level Design - Interview Questions.

04.04.2023

—

Hina Arora

LinkedIn: <https://www.linkedin.com/in/hinaaroraa/>

Kindly Share your feedback on LinkedIn

Let me know if you want me to add more Design Questions.

Content

I. Introduction

II. Why HLD is important

III. System Design Basics

- A. Caching
- B. Proxy
- C. Scalability
- D. Horizontal Scaling
- E. Vertical Scaling
- F. Load balancer
- G. Databases
- H. Sharding
- I. Partitioning
- J. CAP theorem
- K. Consistent hashing
- L. Monolithic architecture
- M. Microservice architecture

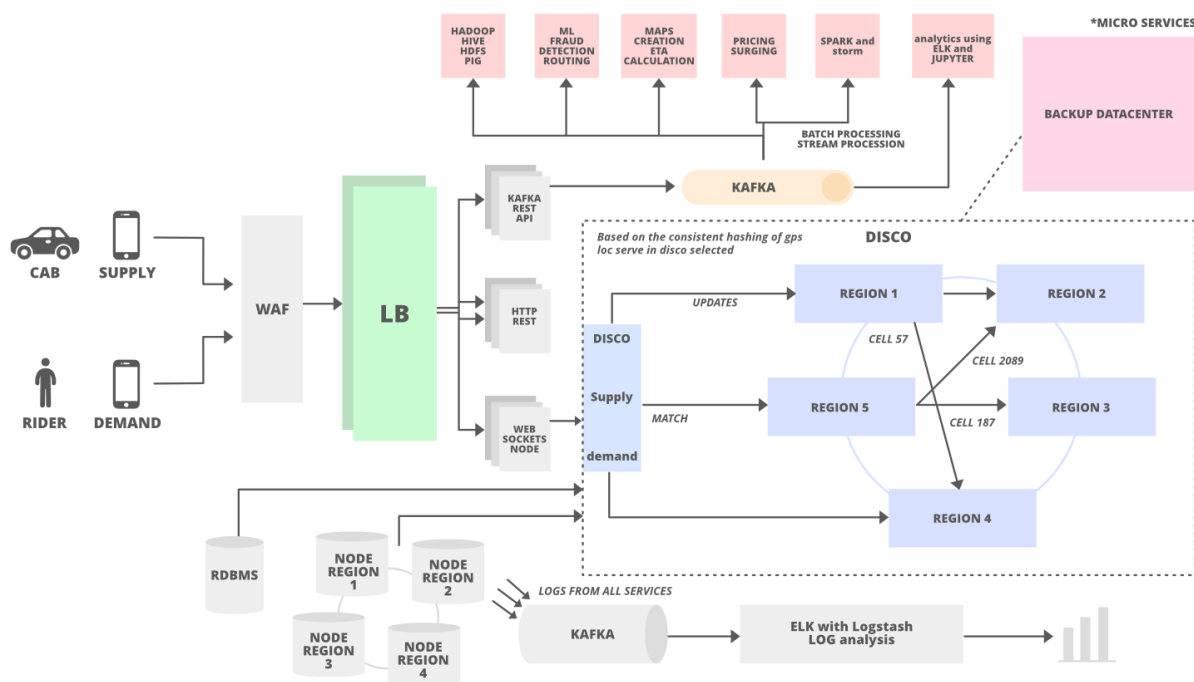
IV. Design Questions

- A. Design Notification System
- B. Design Netflix

Introduction:

High-level design is a plan or a blueprint for creating something like a building, a machine, or a computer system. It's like a map that shows how everything will be put together and how it will work. Just like when you draw a picture or build something with blocks, you first create a plan in your mind or on paper about what you want to make and how it will look like. Similarly, high-level design is the first step towards creating something big, where you plan everything in detail before you start building or making it. It helps to ensure that everything fits together correctly and that the final product will work the way it is supposed to.

It is the process of creating a general and abstract overview of a complex system or project. It involves identifying and defining the main components of the system, their relationships, and their interactions. High-level design is often used in software development, engineering, and other technical fields to plan and organize complex projects before detailed implementation begins. The goal of high-level design is to create a conceptual roadmap for the project that helps ensure that it meets its objectives and is completed on time and within budget. It is a crucial step in the overall design and development process.





Here are some basic principles of high-level design:

Understanding requirements: High-level design starts with understanding the requirements of the system. This includes functional requirements, which describe the system's behavior, and non-functional requirements, which describe the system's qualities, such as performance, security, and reliability.

Breaking down the system: A complex system can be broken down into smaller, more manageable components. Each component can be designed and developed separately, and then integrated into the system as a whole.

Defining the architecture: The high-level design should define the system's architecture, which includes the overall structure of the system and the relationships between its components. The architecture should be designed to meet the requirements of the system.

Creating a design document: The high-level design should be documented in a design document. The document should include the requirements, architecture, and other key components of the system, as well as any constraints or limitations that may affect the design.

Reviewing and testing: The high-level design should be reviewed and tested to ensure that it meets the requirements of the system. This includes reviewing the design document, conducting design reviews, and testing the system's components and overall functionality.

Why HLD is important

High-Level Design (HLD) is important because it provides a conceptual roadmap for developing a complex system, such as a software application, computer network, or engineering project. Here are some reasons why HLD is important:

Helps ensure that the system meets its objectives and requirements: HLD helps define the functional and non-functional requirements of the system, which ensures that the system meets its objectives and requirements.

Identifies potential issues early in the development process: HLD breaks the system down into smaller components and defines their interactions, which can reveal potential issues or conflicts early in the development process.

Facilitates communication and collaboration among the development team and stakeholders: HLD provides a shared understanding of the system's architecture and components, which facilitates communication and collaboration among the development team and stakeholders.

Reduces development time and costs: HLD can help reduce development time and costs by providing a clear roadmap for development and identifying potential issues early. By avoiding costly mistakes and rework, HLD can help save time and money in the long run.

Provides a foundation for future development and modifications: HLD can provide a foundation for future development and modifications by defining the system's architecture and components. This can make it easier to add new features, modify existing ones, or adapt the system to new requirements.

System Design Basics

System design is the process of defining the architecture, components, modules, interfaces, and data for a system to satisfy specified requirements. The basics of system design include understanding the requirements of the system, designing the architecture of the system, identifying the components and modules that make up the system, and defining the interfaces and data that are exchanged between the components and modules.

To design a system, it is important to first understand the requirements of the system. This involves defining the problem that the system is intended to solve, identifying the users and stakeholders of the system, and understanding the constraints that the system must operate within.

Once the requirements are understood, the architecture of the system can be designed. The architecture defines the overall structure of the system and the relationships between its components and modules. It is important to design an architecture that is scalable, maintainable, and flexible, and that can accommodate future changes and growth.

The components and modules of the system can then be identified and designed. This involves breaking down the system into smaller, more manageable pieces that can be developed and tested independently. Each component and module should have a well-defined purpose and interface with other components and modules.

The interfaces and data that are exchanged between the components and modules must also be defined. This involves specifying the protocols, data formats, and other details that are necessary for the components and modules to communicate and work together.

Overall, the basics of system design involve understanding the requirements of the system, designing the architecture of the system, identifying the components and modules that make up the system, and defining the interfaces and data that are exchanged between the components and modules. This process requires careful planning, attention to detail, and a deep understanding of the problem that the system is intended to solve.

Caching:

Caching is a mechanism of storing frequently accessed data in a location that is closer to the user or application, to reduce the response time and improve performance. Caching is an essential component of system design and can significantly improve the performance of a system. Following are the various caching strategies available

Time-based caching: In this strategy, data is stored in the cache for a certain amount of time before it is invalidated and refreshed.

Least Recently Used (LRU) caching: This strategy involves removing the least recently used data from the cache when the cache is full and new data needs to be stored.

Least Frequently Used (LFU) caching: This strategy involves removing the least frequently used data from the cache when the cache is full and new data needs to be stored.

Write-through caching: In this strategy, data is written both to the cache and to the backend storage at the same time. This ensures that the data in the cache is always up-to-date.

Write-behind caching: This strategy involves writing data to the cache first and then later writing it to the backend storage. This can result in better performance, but there is a risk of data loss if the system crashes before the data is written to the backend storage.

Read-through caching: In this strategy, when data is requested from the cache and is not present, the cache will automatically retrieve the data from the backend storage and store it in the cache for future requests.

Read-around caching: This strategy involves bypassing the cache for read operations and retrieving the data directly from the backend storage. This can be useful for large data sets that are not frequently accessed.

Write-around caching: This strategy involves bypassing the cache for write operations and writing data directly to the backend storage. This can be useful for data that is not frequently accessed and does not need to be stored in the cache.

There are several types of caching that can be used in computing:

- a. **Memory caching:** This involves storing frequently accessed data in memory to reduce the time needed to access it from the main memory or disk.
- b. **Web caching:** This is used to store web pages and web objects such as images, scripts, and style sheets, so that they can be served faster to users.

c. Browser caching: This is used by web browsers to store a copy of web pages, images, and other resources on a user's computer, so that they can be accessed more quickly the next time the user visits the same website.

d. Database caching: This is used to store frequently accessed data from a database in memory, so that it can be accessed faster than if it were read from disk.

e. Distributed caching: This involves storing frequently accessed data across multiple servers in a distributed system, so that it can be accessed faster and more reliably.

f. Content caching: This is used to store frequently accessed content such as videos, music, and images on a local server, so that it can be served more quickly to users.

Proxy:

A proxy is an intermediary server or software that sits between clients and servers, forwarding client requests to the server and then returning the server's responses to the clients. The primary function of a proxy server is to provide a layer of abstraction and control between clients and servers, acting as a gateway or filter for network traffic. Proxies can be used for a variety of purposes such as:

Improving performance: Proxies can be used to cache frequently accessed content, such as web pages or media files, to reduce the load on the origin server and improve response times for clients.

Enhancing security: Proxies can act as a buffer between clients and servers, filtering out malicious traffic and blocking access to known malicious sites or IP addresses.

Filtering requests: Proxies can be configured to block access to certain URLs, domains, or IP addresses, or to limit access to certain types of content, such as streaming media or social networking sites.

Anonymizing traffic: Proxies can be used to hide the true source of network traffic by acting as an intermediary between clients and servers, making it difficult to trace the origin of requests.

Controlling access: Proxies can be used to enforce access control policies, limiting access to certain resources or services based on user credentials or other criteria.

There are several types of proxies, including:

a. Forward Proxy: A forward proxy is a proxy that sits between the client and the internet. It's used to access websites and resources that are on the internet.

b. Reverse Proxy: A reverse proxy is a proxy that sits between the internet and a web server. It's used to distribute traffic among several servers and to provide load balancing, caching, and security.

- c. Open Proxy: An open proxy is a proxy server that anyone can use to connect to the internet. It's often used for anonymous browsing or to bypass content filters.
- d. Transparent Proxy: A transparent proxy is a proxy that doesn't modify the request or response headers. It's used to cache content and to improve network performance.
- e. Anonymous Proxy: An anonymous proxy is a proxy that doesn't disclose the client's IP address to the destination server. It's often used for privacy and to bypass content filters.
- f. High Anonymity Proxy: A high anonymity proxy is a proxy that doesn't disclose the client's IP address to the destination server and doesn't add any additional headers to the request. It's often used for high security applications.

Scalability:

Scalability refers to the ability of a system, network, or process to handle a growing amount of work, traffic, or data in a graceful and efficient manner. In other words, it is the capability of a system to increase its resources and handle more load and traffic without compromising its performance, reliability, and availability. Scalability is an important aspect of system design because it ensures that a system can continue to function properly even as the user base and workload increase over time.

It is often achieved through various techniques such as horizontal scaling, vertical scaling, load balancing, and caching.


Horizontal Scaling:

Horizontal scaling, also known as scaling out, is the process of adding more machines or nodes to a system to increase its capacity and performance. In horizontal scaling, multiple machines are added to the existing infrastructure to handle the increased workload, thus distributing the load among multiple machines.

Horizontal scaling is achieved by adding more servers or nodes to the existing server infrastructure, which helps in reducing the load on a single server, thereby improving the performance and availability of the system. The primary advantage of horizontal scaling is that it allows for more flexible scaling since new servers can be added as needed.

Vertical Scaling:

Vertical scaling, also known as scaling up or scaling vertically, refers to adding more resources to a single node in a system to improve its performance, such as increasing CPU, RAM, or disk space. This approach involves upgrading the hardware of an existing system to handle more load rather than adding more nodes to the system.



Vertical scaling can be useful for systems that require high processing power or have a high volume of data processing. It is often more cost-effective and less complex than horizontal scaling, as it involves fewer servers to manage. However, there is a limit to how much a single node can be scaled vertically, and eventually, it becomes necessary to add more nodes to achieve better scalability.

Load Balancer:

A load balancer is a networking device or software application that distributes incoming network traffic across multiple servers or resources. The main purpose of a load balancer is to improve the availability, scalability, and reliability of applications, websites, and services by spreading the workload evenly across multiple servers. Following are the benefits of load balancers:

- a. Improved performance and scalability: Load balancers distribute incoming traffic evenly across multiple servers, which can improve the performance and scalability of applications or services. By balancing the workload, a load balancer can prevent any one server from becoming overwhelmed with traffic and slowing down the entire system.
- b. Increased availability and reliability: Load balancers can monitor the health of servers and redirect traffic away from any servers that are not responding or are experiencing issues. This can help ensure that the application or service remains available and reliable to users.
- c. Flexibility and agility: Load balancers can be used to add or remove servers from a pool of available resources as needed. This can allow for easy scaling up or down to meet changing demand without affecting the overall performance or availability of the application or service.
- d. Security: Load balancers can help protect against certain types of attacks, such as distributed denial of service (DDoS) attacks, by filtering traffic and blocking malicious requests.

Databases:

A database is a structured collection of data that is stored and organized in a way that allows for efficient retrieval, manipulation, and updating of the information. Databases are used to store and manage a wide range of information, from small personal collections to large corporate and government data sets.

There are several types of databases, including:

- a. Relational databases: These are the most common type of database and are based on the relational model. In this model, data is organized into tables, with each table consisting of rows and columns. Relational databases use Structured Query Language (SQL) to query and manipulate the data.

- b. NoSQL databases: These databases are designed to handle large volumes of unstructured or semi-structured data. They are non-relational, meaning they do not use the traditional tabular structure of relational databases. Instead, they use flexible data models, such as document-based or graph-based structures, and may use different query languages or APIs for accessing the data.
- c. Object-oriented databases: These databases store objects, which can be thought of as individual units of data that contain both data and behavior. They are commonly used in object-oriented programming languages like Java and C#.
- d. Hierarchical databases: These databases are organized in a tree-like structure, with each parent node having one or more child nodes. They are primarily used in legacy systems and mainframe environments.
- e. Distributed databases: These databases are spread across multiple computers or locations, and can be accessed and managed as a single database. They are commonly used in cloud computing and other distributed computing environments

Sharding:

Sharding is a technique used in database architecture to improve scalability and performance in large, distributed systems. It involves dividing a large database into smaller, more manageable pieces called "shards" that are spread across multiple servers.

Each shard contains a subset of the data, and the data is partitioned based on a key or set of keys. For example, if a database contains customer information, the data might be partitioned based on the customer's geographic location or last name. Each shard is then stored on a separate server, which can be located in a different geographic location.

By dividing the data into shards, the database can handle a larger amount of data and a higher number of requests. Each server can handle a smaller amount of data, which makes it easier to manage and scale. Additionally, sharding can improve performance by allowing queries to be distributed across multiple servers, which can be processed in parallel.

However, sharding also introduces complexity and potential challenges. For example, if the partition key is not chosen carefully, it can lead to uneven distribution of data and uneven workload across servers. Also, some queries may require data from multiple shards, which can slow down the query processing time.

Sharding is commonly used in large-scale web applications and distributed systems, such as social media platforms and online marketplaces.

Partitioning:

Partitioning is a technique used in database management systems to divide a large database into smaller, more manageable sections called partitions. Each partition contains a subset of the data, and is stored on a separate storage device or server.

Partitioning is useful in situations where the size of a database exceeds the storage capacity of a single server or storage device, or when certain data needs to be accessed more frequently than others. Partitioning can also improve query performance by allowing queries to be processed in parallel across multiple partitions.

There are several different types of partitioning strategies:

- a. Range partitioning: Data is partitioned based on a range of values, such as dates or numeric values.
- b. List partitioning: Data is partitioned based on a specific list of values, such as geographic regions or customer types.
- c. Hash partitioning: Data is partitioned based on a hash function applied to a specific column or set of columns.
- d. Composite partitioning: Data is partitioned using a combination of multiple partitioning strategies.

Partitioning can be done at various levels, including at the table level, index level, and sub-partition level. When partitioning a database, it is important to carefully consider the partitioning strategy and the partitioning key, as it can significantly impact the performance and maintenance of the database.

Partitioning is commonly used in large-scale databases, such as data warehouses and distributed database systems, to improve performance and scalability.

CAP theorem:

The CAP theorem, also known as Brewer's theorem, is a concept in distributed systems that states that it is impossible for a distributed system to simultaneously guarantee all three of the following properties:

Consistency: All nodes in the system see the same data at the same time, even in the presence of updates.

Availability: Every request to the system receives a response, without guaranteeing that it contains the most recent version of the data.

Partition tolerance: The system continues to function even when network partitions occur, meaning that communication between nodes is lost or delayed.

The CAP theorem implies that in a distributed system, a trade-off must be made between consistency, availability, and partition tolerance. In the event of a network partition or failure, a distributed system must choose between being consistent (meaning that all nodes see the same data) or available (meaning that every request receives a response).

For example, in a distributed database system, a partition may occur between two nodes, causing them to lose communication. If the system prioritizes availability over consistency, each node can continue to operate independently, potentially resulting in different versions of the same data. If the system prioritizes consistency over availability, the system may pause all operations until the partition is resolved, ensuring that all nodes have the same data.

The CAP theorem is an important consideration when designing and building distributed systems, as it helps developers understand the trade-offs that must be made in order to achieve the desired level of consistency, availability, and partition tolerance.

Consistent hashing:

Consistent hashing is a technique used in distributed computing and computer networking to distribute data evenly across multiple servers or nodes in a network. It is a hash-based approach that provides a way to map a key or data item to a particular server in a consistent and efficient manner.


Consistent hashing solves this problem by using a hash ring, which is a circular space where each node in the system is represented by a point on the ring. Data items are also mapped to a point on the ring using the same hash function.

When a node is added or removed from the system, only a small portion of the data needs to be redistributed to new nodes, which makes the process much more efficient. The hash function used in consistent hashing ensures that each data item is consistently mapped to the same point on the ring, even if nodes are added or removed.

Consistent hashing is used in many distributed systems, such as content delivery networks, distributed databases, and load balancers. It allows these systems to distribute data and workload evenly across multiple nodes, while also providing fault tolerance and scalability.

Monolithic architecture:

Monolithic architecture is a traditional software architecture in which an application is built as a single, self-contained unit. In a monolithic architecture, the entire application, including the user interface, business logic, and data storage, is developed and deployed as a single package on a single server or computing platform.



In a monolithic architecture, the different components of the application are tightly coupled and share the same memory space and code base. This means that any changes to one component may affect the functionality of other components, and scaling the application may require scaling the entire monolith, which can be inefficient and costly.

Monolithic architecture was the dominant approach to software development for many years, but it has become less popular in recent years as more developers have moved toward microservices and other distributed architectures. However, monolithic architecture still has some advantages, including simplicity, ease of deployment, and easier maintenance for smaller applications.

Some examples of applications that still use monolithic architecture include many enterprise-level applications, such as accounting software, customer relationship management (CRM) systems, and human resources management systems (HRMS).

Microservice architecture:

Microservices is an architectural style used in software development where an application is built as a collection of small, independent, and loosely coupled services. Each service in a microservices architecture is designed to perform a specific function, and can be developed, deployed, and scaled independently from other services.

Unlike monolithic architecture, where the entire application is built and deployed as a single unit, microservices architecture breaks down the application into multiple smaller services that communicate with each other through well-defined APIs. Each microservice is responsible for a specific task or business function, such as user authentication, data storage, or user interface, and can be developed using different technologies and programming languages.

Design Questions

Now we get the basic understanding of all the components we can start with interview questions.

Design Notification System:

In today's mobile and web-dominated world, we receive a plethora of notifications on a daily basis. The question is, how can a product ensure that these messages reach users in an efficient and precise manner?

Functional Requirement:

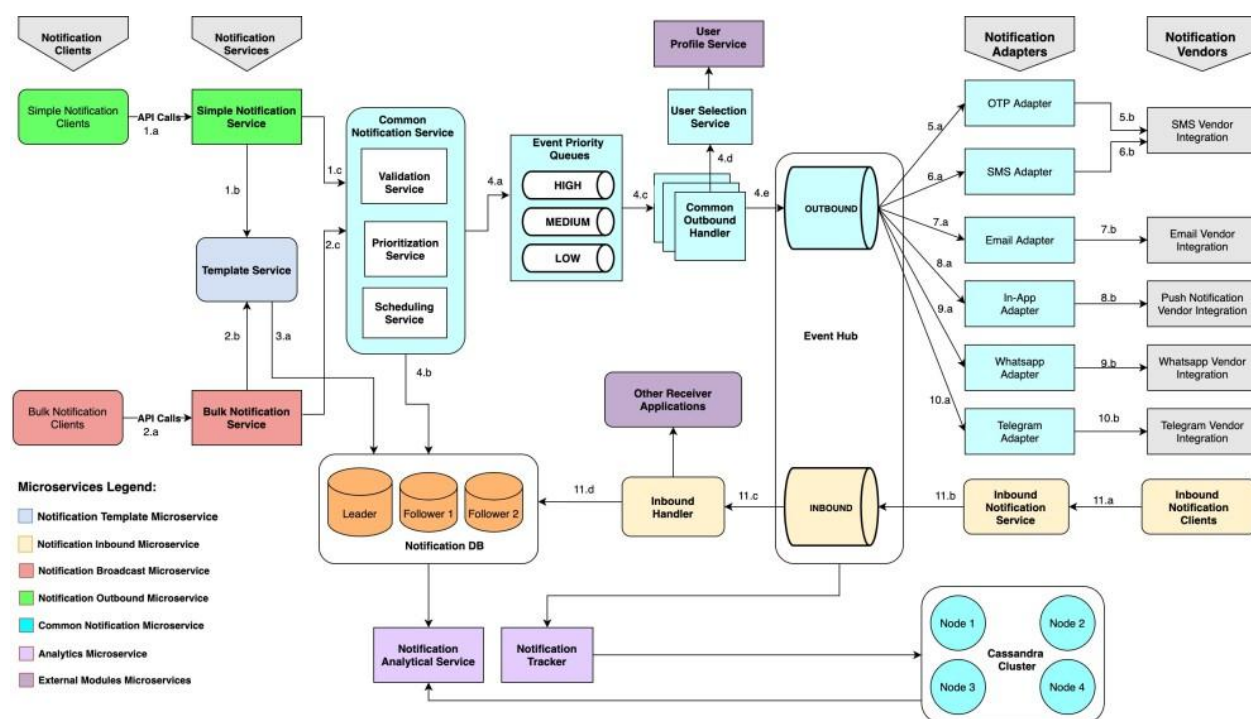
- Send notifications
- Prioritize notifications
- Send notifications based on customer's saved preferences
- Single/simple and bulk notification messages
- Analytics use cases for various notifications
- Reporting of notification messages

Non-functional requirements (NFR):

- High performance
- Highly available (HA)
- Low latency
- Extendable/Pluggable design to add more clients, adapters and vendors.
- Support Android/iOS mobile and desktop/laptop web browsers.
- API integration with all notification modules and external integrations with clients and service providers/vendors.
- Scalable for higher load on-prem (VMware Tanzu) and on public cloud services like AWS, GCP, or Azure etc.

Design enterprise level system architecture to support email, SMS, Chat and other public social app integrations using API:

- Email
- SMS/OTP
- Push notifications (Mobile and Web browser)
- Chat – Whatsapp/Telegram



[Notification System Technical Architecture]

The System Component

1. Notification clients:

These clients will request for single and bulk messages using API calls. These clients will send notification messages to simple and bulk notification services:


- Bulk Notification clients: These clients send bulk notification(s).
- Simple Notification clients: These clients send single notification(s).

2. Notification Services:

These services are entry services which will expose REST APIs to clients and interact with the clients. They are responsible to build notification messages by consuming Template Service. These messages will be also validated using Validation Service.

Simple Notification Service: This service will expose APIs to integrate client with backend services. It's a main service, which will handle simple notification request.

Bulk Notification Service: This service will expose APIs to integrate client with backend services. It's a main service, which will handle bulk notification request.



This service will also manage notification messages. It will persist sent messages to databases and maintain activity log. Same message can be resent using APIs of these services. It will provide APIs to add/update/delete and view old and new messages. It will also provide web dashboard which should have filter option to filter messages based on different criteria like date range, priority, module user, user groups etc.

3. Template Service:

This service manages all ready-to-use templates for OTP, SMS, Email, chat and other push notification messages. It also provides REST APIs to create, update, delete and manage templates. It will also provide an UI dashboard page to check and manage message templates from web console.

4. User Selection Service:

This service will provide services to choose target users and various application modules. There could be use cases to send bulk messages to specific group of users or different application modules. It could be also AD/IAM/eDirectory/user database/ user groups based on customer's preferences. Internally, it will consume API services of User Profile Service APIs and check customers notification preferences.

5. User Profile Service:

This service will provide various features including managing users profile and their preferences . It will also provide feature to unsubscribe for notifications and also notification receiving frequency etc. Notification Service will be dependent on this service.

6. Common Notification Service

Scheduling Service:

This service will provide APIs to schedule notifications like immediate or any given time. It could be any of these followings:

- Second
- Minute
- Hourly
- Daily
- Weekly
- Monthly
- Yearly
- Custom frequency etc.

There could be other services also, which can be auto-triggered messages based on the scheduled times.

Validation and Service:

This service solely responsible for validating notification messages against business rules and expected format. Bulk messages should be approved by authorized system admin only.

Validation Service:

It will also prioritize notification based on high, medium and low priorities. OTP notification messages have higher priority with a time-bound expiry time, they will always be sent in higher priority. Common Outbound Handler will consume messages and process and send based on the same priorities from reading in three different queues high, medium and low. Another use case of bulk messages can be send using low priority during off hours. Application notifications during transactions could be sent to medium priority like email etc. Business will decide priority based on criticality of the notifications.

7. Event Priority Queues (Event Hub):

It will provide event hub service which will consume messages from notification services in high, medium and low topics. It sends processed and validated messages to Notification Handler Service which internally uses Notification Preferences Service to check users personal preferences.

It will have these three topics, which will be used to consume/send messages based on business priority:

- High
- Medium
- Low

8. Common Outbound Handler:

This service will consume notification messages from Event Hub by polling event priority queues based on their priority. High precedence will be given to “High” queue and so on so forth. Finally It will send notification messages to message specific adapter thru Event Hub.

This service will also fetch target user/applications from User Selection Service.

9. Notification DB

It will persist all notification messages with their delivery time, status etc. It will have a cluster of databases with a leader which will be used to perform all write operations and read will be on read replica/followers. It should be No-SQL database.

10. Outbound Event Hub:

It finally transmits message to various supported adapters. These adapters will be based on different devices (desktop/mobile) and notification types(sms/OTP/Email/Chat/Push notifications).

11. Notification Adapters:

These are adapters which will transform incoming messages from event hub (Kafka) and send to external vendors according to their supported format. These are a few adapters, we can add more based on use case requirements:

- OTP Adapter Service
- SMS Adapter Service
- Email Adapter Service
- In-App Notification Adapter Service
- WhatsApp Chat Notification Adapter Service
- Telegram Notification Adapter Service

12. Notification Vendors:

These are the external SAAS (on cloud/on-prem) vendors, which provide actual notification transmission using their infrastructure and technologies. They maybe paid enterprise services like AWS SNS, MailChimp etc.

- SMS Vendor Integration Service
- Email Vendor Integration Service
- App Push Notification Vendor Integration Service
- WhatsApp Vendor Integration Service
- Telegram Vendor Integration Service

13. Notification Analytical Service

This service will do all analytics and identify notification usage, trends and do a reporting on top of that. It will pull all final notifications messages from analytical database (Cassandra) and Notification databases for analytics and reporting purpose.

These are a few use cases:

- Total number of notifications per day/per sec.
- Which is highly used notification system.
- What's average size and frequency of messages.
- Filter out messages based on their priorities and many more...

14. Notification Tracker

This service will continuously read Event hub queues and track all sent notifications. It captures metadata of the notifications like transmission time delivery status, communication channel, message type etc.

15. Cassandra Database Cluster

This database cluster will persist all notifications for analytics and reporting purpose. It's based on write more and read less concept.

This will provide good performance and low latency for high number of notifications, because it internally manages high number of write operations and sync up with other database nodes and keep duplicate data/messages for high availability and reliability. Messages will be always available in case of any node get crashed.

15. Inbound Notification Service

This service will expose API endpoint to external clients and applications to send inbound messages.

16. INBOUND event Hub

This topic will be used to queue and process all incoming notification messages from Inbound notification clients.

17. Inbound Handler

This will consume all incoming notification messages from INBOUND topic.

18. Inbound Notification Clients

These inbound notification messages will come from internal and external sources/applications.

Design Netflix

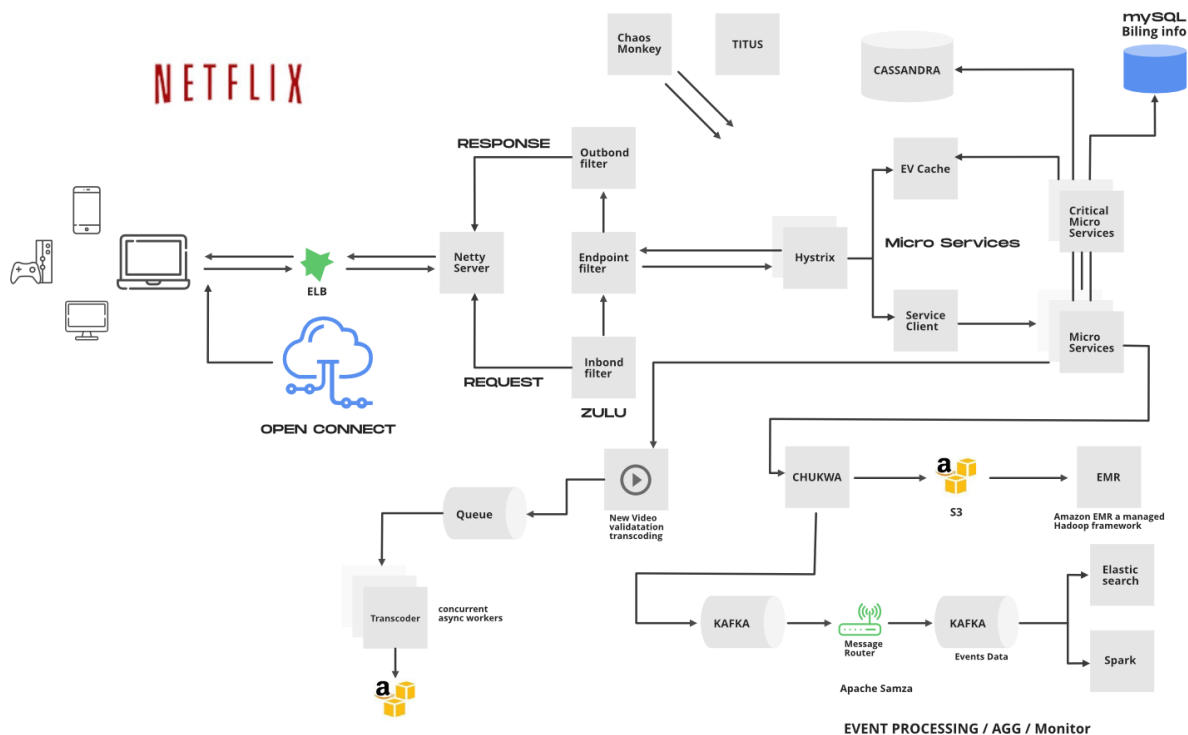
Functional Requirements

1. Feature to allow users to create accounts and subscribe to a plan
2. Allow users to handle multiple accounts
3. Allow users to watch videos
4. Let Netflix developers upload video from the backend and make it available on the platform.

Non-functional Requirements

1. There should be no buffering, i.e., provide users with real-time video streaming without any lag.
2. Reliable system
3. High Availability
4. Scalability

We all are familiar with Netflix services. It handles large categories of movies and television content and users pay the monthly rent to access these contents. Netflix has 180M+ subscribers in 200+ countries.



Netflix works on two clouds...**AWS** and **Open Connect**. The synergy of these two cloud systems serves as the foundation of Netflix, playing a significant role in delivering top-quality video content to its users.

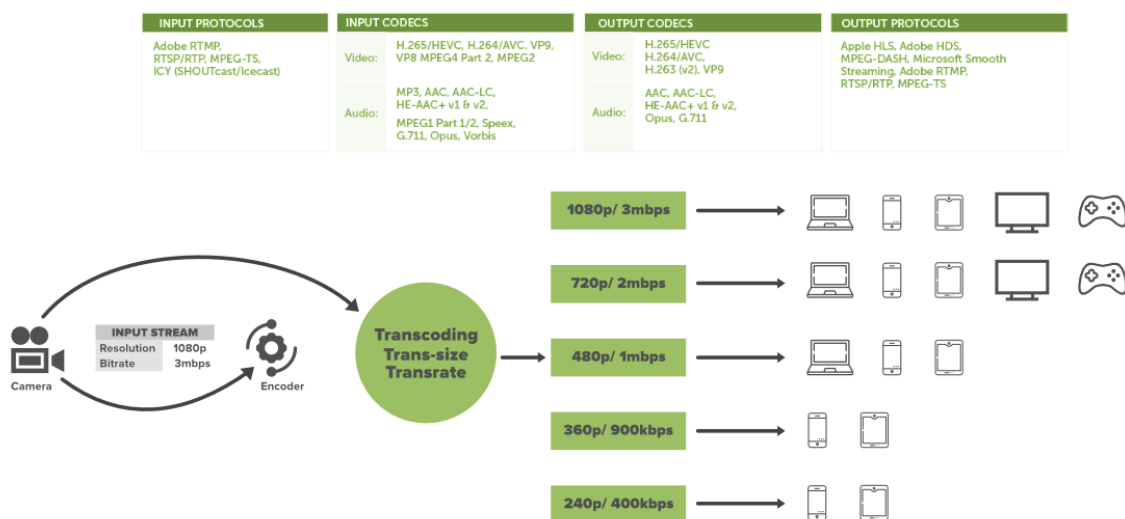
Netflix has 3 main components which we are going to discuss today

Client: Device (User Interface) which is used to browse and play Netflix videos. TV, XBOX, laptop or mobile phone, etc

OC (Open connect) or Netflix CDN: Open Connect stores Netflix video in different locations throughout the world. When you press play the video streams from Open Connect, into your device, and is displayed by the client. A content delivery network (CDN) is a system of distributed servers (network) that deliver pages and other Web content to a user, based on the geographic locations of the user, the origin of the webpage and the content delivery server.

Backend (Database): This part handles everything that doesn't involve video streaming (before you hit the play button) such as onboarding new content, processing videos, distributing them to servers located in different parts of the world, and managing the network traffic. Most of the processes are taken care of by Amazon Web Services.

How does Netflix Onboard a Movie/Video?



Netflix needs to transcode or encode a movie before it can be streamed to users, which involves converting the original video file into a format that is compatible with various devices and platforms.

This is necessary because the original video file is often very large, and different devices have different requirements for optimal viewing quality. For instance, if a user is watching Netflix on an iPhone, they need a video format that is specifically optimized for that device to provide the best viewing experience.

By transcoding the video into multiple formats, Netflix ensures that its users can access the content on any device, without sacrificing quality or experiencing playback issues.

Netflix also creates files optimized for different network speeds. If you're watching on a fast network, you'll see higher quality video than you would if you're watching over a slow network. And also depends on your Netflix plan. That said Netflix does create approx 1,200 files for every movie !!!!

That's a lot of files and processing to do transcoding. Now we have all the files we need to stream it. OC Open connect comes into picture, OC is Netflix own CDN no third-party CDN.

Advantages of OC

- a. Less expensive
- b. Better quality
- c. More Scalable

So once the videos are transcoded these files are pushed to all of the OC servers.

1. Elastic Load Balancer:

Elastic Load Balancer, is a service provided by Netflix to manage the routing of traffic to frontend services. In order to balance the load efficiently,

ELB uses a two-tier load-balancing scheme.

- The first tier is a DNS-based Round Robin Balancing, which is a basic load-balancing technique. When a request is received, it is balanced across one of the zones that the ELB is configured to use.
- The second tier is an array of load balancer instances, which use Round Robin Balancing to distribute the request across the instances behind it in the same zone. This allows for efficient distribution of traffic and helps ensure that the workload is distributed evenly across all servers, which improves performance and reduces the risk of downtime due to overloading.

2. ZUUL:

ZUUL is a gateway service that provides dynamic routing, monitoring, resiliency, and security. It provides easy routing based on query parameters, URL, and path. Let's understand the working of its different parts...

The Netty server takes the responsibility to handle the network protocol, web server, connection management, and proxying work. When the request hits the Netty server, it will proxy the request to the inbound filter.

The inbound filter is responsible for authentication, routing, or decorating the request. Then it forwards the request to the endpoint filter.

The endpoint filter is used to return a static response or to forward the request to the backend service (or origin as we call it). Once it receives the response from the backend service, it sends the request to the outbound filter.

An outbound filter is used for zipping the content, calculating the metrics, or adding/removing custom headers. After that, the response is sent back to the Netty server and then it is received by the client.

3. Hystrix

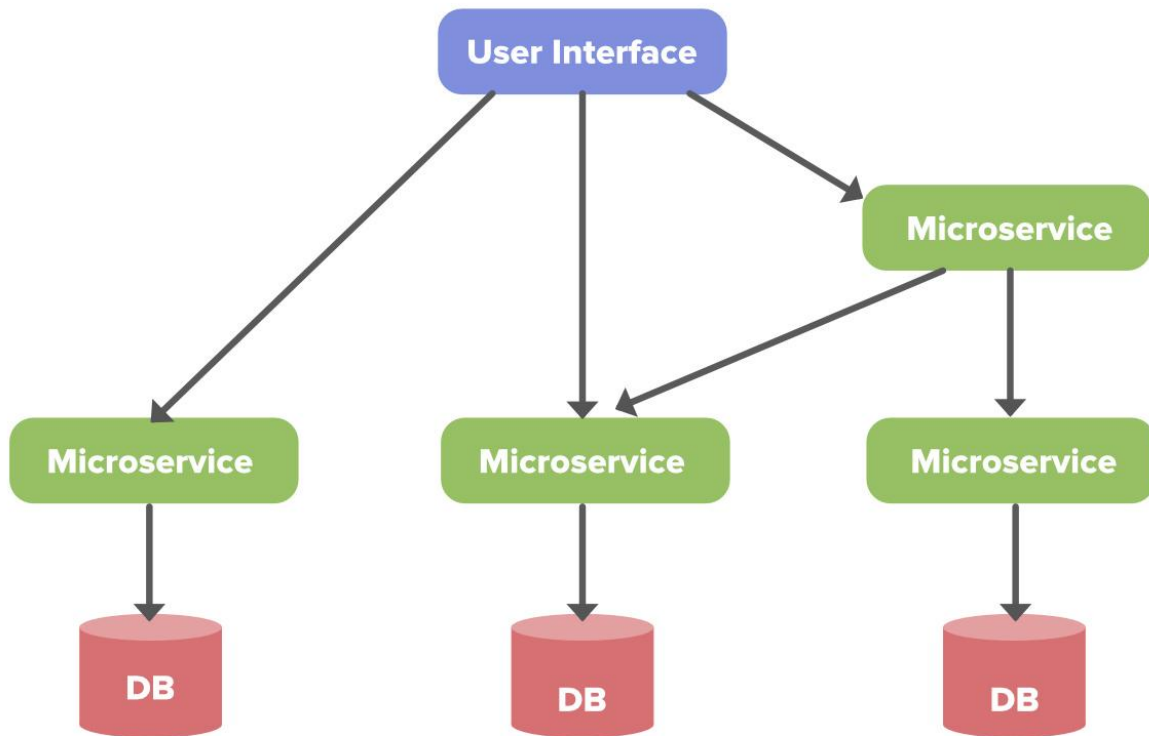
In a complex distributed system a server may rely on the response of another server. Dependencies among these servers can create latency and the entire system may stop working if one of the servers will inevitably fail at some point. To solve this problem we can isolate the host application from these external failures. Hystrix library is designed to do this job. It helps you to control the interactions between these distributed services by adding latency tolerance and fault tolerance logic. Hystrix does this by isolating points of access between the services, remote system, and 3rd party libraries. The library helps in.

- Stop cascading failures in a complex distributed system.
- control over latency and failure from dependencies accessed (typically over the network) via third-party client libraries.
- Fail fast and rapidly recover.
- Fallback and gracefully degrade when possible.
- Enable near real-time monitoring, alerting, and operational control.
- Concurrency-aware request caching. Automated batching through request collapsing

4. Microservice Architecture of Netflix


Netflix's architectural style is built as a collection of services. This is known as microservices architecture and this powers all of the APIs needed for applications and Web apps. When the request arrives at the endpoint it calls the other microservices for required data and these microservices can also request the data from different microservices. After that, a complete response for the API request is sent back to the endpoint.

In a microservice architecture, services should be independent of each other, for example, the video storage service would be decoupled from the service responsible for transcoding videos. Now, let's understand how to make it reliable...



5. EV Cache

In most applications, some amount of data is frequently used. For faster response, these data can be cached in so many endpoints and it can be fetched from the cache instead of the original server. This reduces the load from the original server but the problem is if the node goes down all the cache goes down and this can hit the performance of the application. To solve this problem Netflix has built its own custom caching layer called EV cache. EV cache is based on Memcached and it is actually a wrapper around Memcached.



Netflix has deployed a lot of clusters in a number of AWS EC2 instances and these clusters have so many nodes and they also have cache clients. The data is shared across the cluster within the same zone and multiple copies of the cache are stored in sharded nodes. Every time when write happens to the client all the nodes in all the clusters are updated but when the read happens to the cache, it is only sent to the nearest cluster (not all the cluster and nodes) and its nodes. In case, a node is not available then read from a different available node. This approach increases performance, availability, and reliability.

6. Database

Netflix uses different data stores comprising both SQL and NoSQL for different purposes.

MySQL

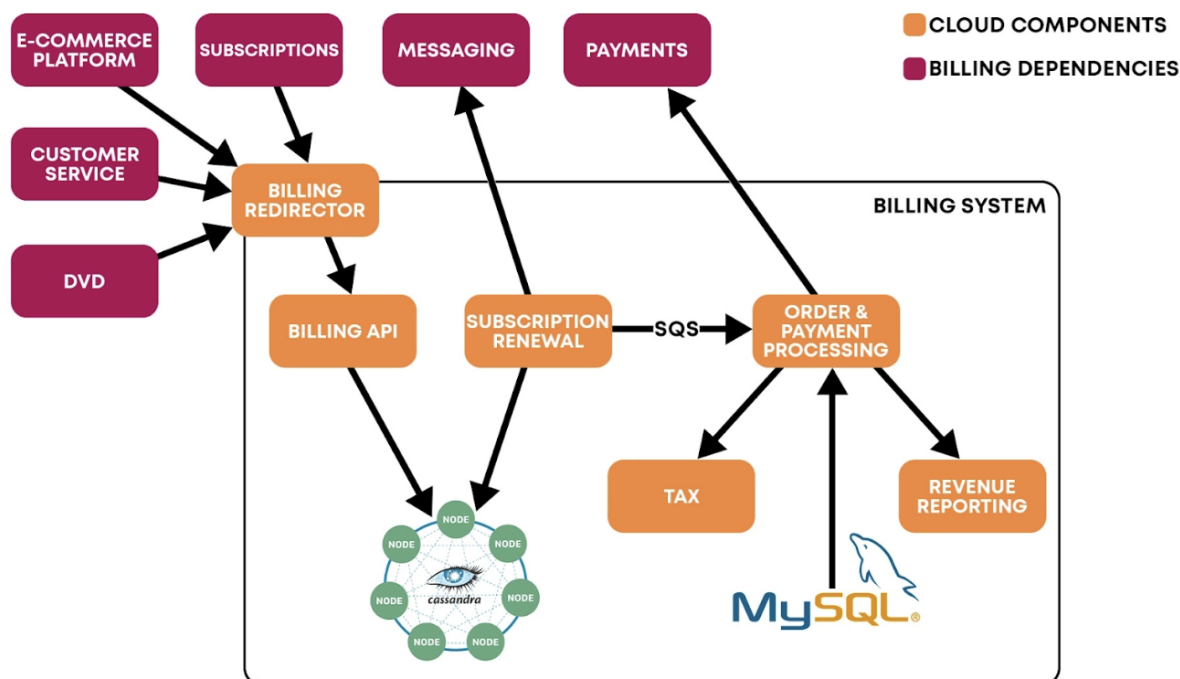
MySQL databases are used for managing movie titles, billing, and transaction purposes.

To be specific, AWS EC2 Deployed MySQL is used to store the data.

MySQL is built using the InnoDB engine over large AWS EC2 instances. Data from User Service where we need strong ACID properties, this RDBMS is an obvious choice.

Replication on this database is done synchronously, which states that there is a master-master relationship between nodes, and any write operation on the primary node will be considered as done only if that data is synchronised by both local and remote nodes to ensure high availability.

Read queries aren't handled by the primary(master) node; it's handled by replicas, only write queries are handled by master DB. In case of failover, the secondary node will take up as the master node and will handle the write query well.



Cassandra(NoSQL)

Cassandra is a distributed column-based NoSQL database that is free to use and open source that enables the storage of a large amount of data over servers. As we know, Netflix has a large user base globally, so it requires such DB to store user history. It enables handling of large amounts of reading requests efficiently and optimises the latency for large read requests.

As the user base grew, it became difficult to store so many rows of data, and it was also costly and slow. So Netflix designed a new Database to store the history of users based on the time frame and recent use.

1. LiveVH (Live Viewing History) - Only recent data, with frequent updates, smaller numbers of rows are stored in an uncompressed form that is used for many operations like analysis, recommendations to the user after performing ETL(extracting, transforming, and loading). This fulfils the motive of using fast and smaller DB and also performing the functionality.

2. CompressedVH (Compressed Viewing History) - Old data of browsing history and viewed by users is stored after compressing with occasional updates. Storage size is also decreased, and only one column per row was stored.

Searching and Data Processing

Search

Search is implemented using Elasticsearch DB that enables users to search for movies, series by title, or any meta-data associated with the video. Elastic search provides the feature of full-text data search and ranking the data based on recommendations, reviews, rankings during search only.

Another application of Elastic search is tracking down users' events in cases of failures(e.g. if a user is unable to play some video). Then the customer care team uses elastic search to resolve issues.

Data Processing

Data processing involves all the events required after a user clicks on the video; it takes nanoseconds to process the video and stream it to the user.

There are around 600 billion events daily, resulting in 1.5 PB data, and during peak hours(evening and night), there are around 8 million events per second.

Events are UI Activities, Video viewing activities, logging errors, troubleshooting, processing events and performance events in the backend.

Reference:

1. <https://medium.com/@narengowda>
2. <https://practice.geeksforgeeks.org/>
3. <https://www.codingninjas.com/>
4. <https://www.linkedin.com/in/rajivkumarsrivastava/>
5. <https://www.youtube.com/@TechDummiesNarendraL>