peopleTECH

A Quest Global Company

INTERNSHIP REPORT

WEEK 3 – System Design

SUBMITTED TO

PEOPLE TECH GROUP INC

GOVIND SHARMA

RECRUITMENT LEAD

SUBMITTED BY

SANJEEV REDDY SIRIPINANE

What is System Design:

- Systems design is actually the determination of the characteristics of a system that is supposed to solve a given problem in a given organization. It entails the ability to turn clients' expectations into a clear roadmap to follow once in the implementation process. Its purpose is to design a consistent and effective structure which will answer the specified function, as well as receiving possible parameters of complexity, sustainability, and speed.
- An example of system design is achieving the implementation of the application of a central platform for sharing car rides popularised by Uber. It includes the development of features like user registration, ride sharing and matching at real time, and the methods of payment. There must be proper architecture with data base for users and rides; proper integration of GPS for tracking the location of the rider ; proper notification system for the updates during the ride. Some include the pairing of riders with drivers and determination of the fare to be charged as well as handling of payments while the other include the frontend interfaces that allow user book a ride easily. To make the system more reliable load balancers inclusion, caching on regularly utilized data, and microservices work on separation of module development is used.

What are Design Patterns?

- Design patterns refer to solutions that can be used whenever there is a common software design problem. It allows those expert object-oriented software engineers to write better, more manageable code – as well as facilitating scale. Many developers need to design patterns to solve certain problems that remain constant and use a standard form of words. Remember, design patterns are not the actual code, but rather pretty stubs or skeletons at the best.

Types of System Design Patterns:

Creational Patterns: Concentrate on the object generation process or associated issues. They contribute to making a system independent of how its items are generated, constructed, and displayed.

- Examples:
  - Singleton: A way of guaranteeing that a class will have only one instance and also that it can be accessed globally.
  - Factory Method: Establishes a way of creating objects but allows subclasses to modify what sort of objects will be created.
  - Abstract Factory: Human:Serve as a starting point for family of goal related or dependent objects without know their actual classes.
  - Builder: Divides creating an object and representation of that same object.

    - Prototype: Causes production of a second object from the first object known as the prototype.

Structural Patterns: Addresses issues with how classes and objects are constructed/put together to build larger structures that are effective as well as elastic. Structural class patterns use inheritance for interfaces or implementation to be used together.

- Examples:
    - Adapter: Translates a set of functions of a class to another, which is expected by the clients.
    - Decorator: Make new actions a part of an object in real-time.
    - Facade: Acts as a tool with a user friendly front end for a complicated back end system.
    - Composite: A scale 1 treatment is the methodology where power-law scaling is applied to individual objects and compositions of objects in a way that is said to be uniform.
    - Bridge: Separates an abstraction from an implementation in order that each may change independently of the other.
    - Proxy: Creates its own object that acts as a stand-in for the actual object in order to manage it.

Behavioral Patterns: Behavioral Patterns address the concept of algorithms and definite roles of performing tasks between objects. Behavioral patterns do not only capture patterns of objects or classes but also patterns of interactions between them. These patterns dictate interactive dynamic control flow that is hard to trace at runtime.

- Examples:
    - Observer: Describes a conditional association between objects meaning when one object undergoes a state change all dependent are informed.
    - Strategy: Allows algorithms to be passed as values in expressions so that they may be replaced by other algorithms at runtime.
    - Command: Wraps a request in an object for passing parameters and queuing of the given request.
    - Iterator: grants sequence access to some components of a collection without showing the collection’s implementation.
    - State: Enables an object to respond differently when a state is changed in that object.
    - Mediator: Reduces the object interactions’ complexity since all interactions are managed from a central point.
    - Chain of Responsibility: Its purpose is to forward a request along a sequence of handlers until one of them accepts it.

Live Streaming System Design

A Live Streaming System Design involves low-cost and low-latency architecture to stream live audio and video from one broadcaster to many viewers. It incorporates several parts for good information exchange, redundancy, and user interactions.

- Ingestion: This is the process whereby the broadcaster feeds raw video and audio streams to the system through protocols such as RTMP or WebRTC. These protocols make it possible to have low latency, and real-time streaming, something necessary for the live stream.
- Transcoding: Transcoding involves the process of turning raw video into lesser and more amounts of video data for normal streaming, for instance, 480p, 720p, 1080p with the use of adaptive bitrate. This means the video quality will depend on the network of the viewer and makes the streaming universal across devices.
- Chunking: The video is split into little clips (lets say 2-5 seconds) in order to better facilitate using protocols such as HLS or DASH. These make it possible to have an adaptive bitrate streaming and reduce the instance where a viewer must buffer.
- While Content Delivery Network (CDN): A CDN delivers video chunks around the world, as it stores content close to the viewer. It maintains high availability and distribution of load to support viewership of millions at any one time.
- Player: The player retrieves video chunks and consumes them with no break. It has adaptive streaming to buffer which means the system can switch to higher bandwidth once the viewer has faster internet connection so as to avoid interruption of play back.
- Real-Time Communication: The capability of WebRTC or WebSocket's that allow live participation, reaction and interaction between the broadcaster and the viewers with little permitted latency.
- Metadata Management: Engagement and streams data like, titles of the stream, viewer count, comment section is live updated to present real time data about the audience and the stream.
- Monitoring and Analytics: The performance sampling systems monitor performance indicators such as latency, bandwidth consumption, and audience interaction. These aid in enhancing the system's performance as well as guarantee the streaming service.

Fault Tolerance:

Fault tolerance in system design means that a system has been developed to work even when the component such as the hardware or software resources has failed. This guarantees operational high availability, reliability, and undisturbed user experience through interruptions. Approaches such as duplicity, error checking, and self-healing are used in order to ensure the availability of the system.

Key strategies include:

- Full Replication: Full replication of all the system components to ensure smooth working when the other set is out of order.
- Partial Replication: The use of redundant frameworks regarding essential applications to acquire extra worth from resources.
- Shadowing/Passive Replication: The maintenance of backup systems inactive and ready only for use in case of an emergency.
- Active Replication: To enable high fault tolerance, all replicas perform inputs at the same time.

Fault tolerance should therefore not be confused with high availability because the latter is primarily concerned with load balancing for system up time. Fault tolerance can make differences in performance, scalability, and costs but is imminent when high availability is needed.

Extensibility: This means how easy it is to add more functionality, features and sizes as well as changes to the features in a system without a large overhaul of the system. It is achieved through:

- Modular Architecture: Individual components can be added or removed, updated and changed as may be necessitated by the business environment.
- Loose Coupling: Signs of its current structure are that the elements in one part have very little effect on those of another.
- Scalability: The system is extendible with little effort. Backward Compatibility: New features are compatible with past versions.
- Separation of Concerns: One system functionality is isolated from another so that it is easier to implement modifications.

Benefits: Anticipating Change, Flexibility, and Economy.

Testing: Testing is the process of determining the quality, reliability and suitability of a system or part of a system through the exercise of a set of tests. Key types include:

- Unit Testing: The third way of system testing is testing individual components.
- Integration Testing: The reusability, testing of integration interactions of the components.
- System Testing: Testing of the system with focus on all the aspects of the system.
- Acceptance Testing: Making sure the system is suitable for the user.
- Performance Testing: Cost benefit analysis of adopting speed, stability, and scalability.

Benefits: The benefits it could bring include areas such as Bug detection, Quality assurance, Improved reliability, User satisfaction.

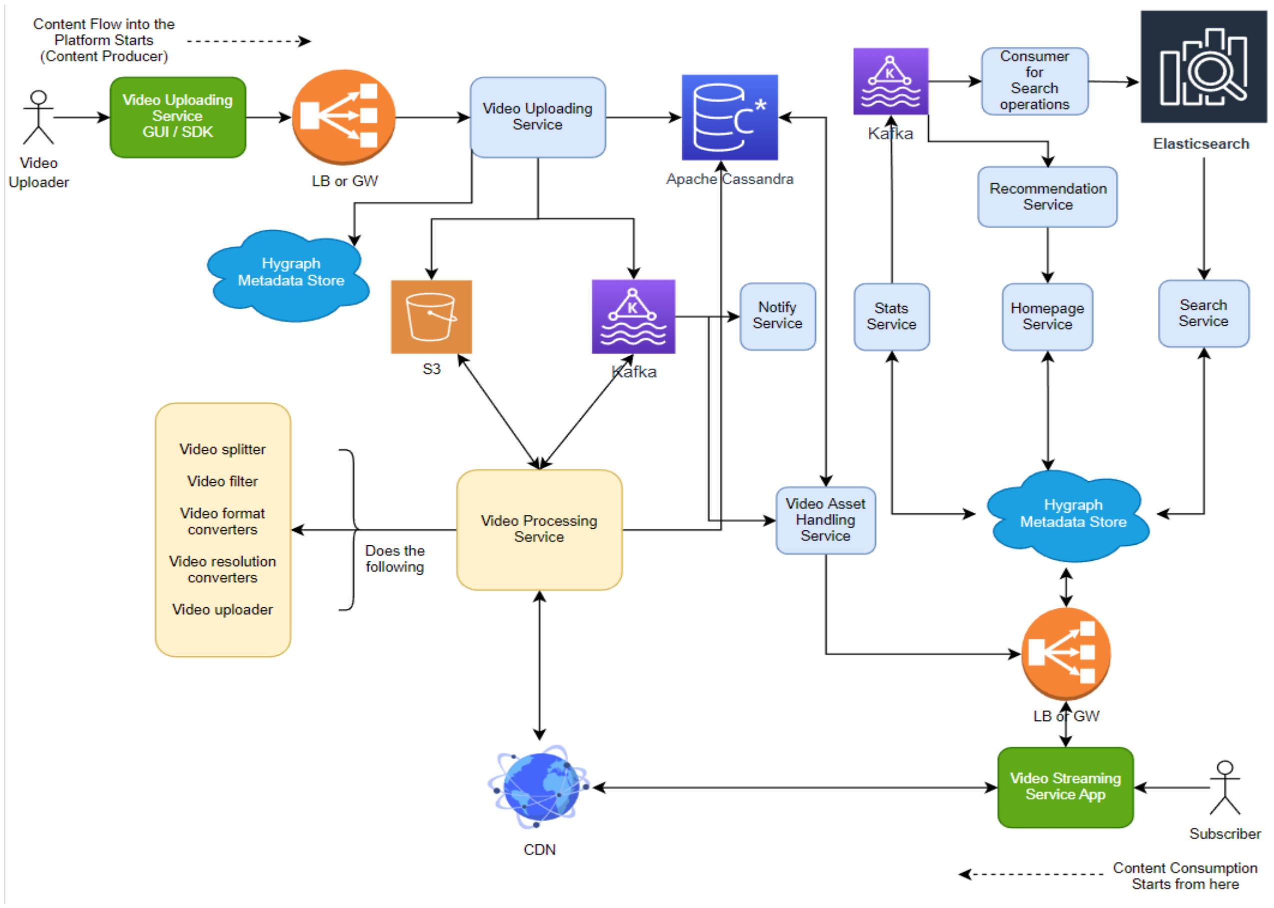
Summarizing the requirements: System design is the process of establishing architecture, components, modules, interfaces, and data for a system in order to meet certain criteria. It entails both high-level and detailed planning to guarantee that a system is scalable, efficient, and satisfies all business and technical requirements.

- Business Goals: Stating clearly the aims and goals of the given system.
- Core Features: Describing the necessary that serves as a base for the system.
- Non-functional Requirements: When it comes to thinking about performance, scalability, security, etc.
- Constraints: Establishing any constraint like technological, financial or physical constraint.
- User Needs: Users expect an easy-to-use system, a system designed to be friendly to them and a system that would have been designed to fit their expectations.
- Architecture: What structure should the system and its components have.
- Performance: After making sure it satisfies load handling capability, response time and throughout.
- Compliance: Compliance with requirements for the operation of the system established under the requirements of antimonopoly legislation and other legislation.
- Success Metrics: Determining how the success of the system will be benchmarked.

Video Streaming System: A video streaming system enables users to see video information in real time via the internet. Rather than downloading the complete movie before watching it, streaming transmits the video in little data chunks, allowing viewers to begin watching right away and continue without waiting for the entire file to download.

- Video capture is the process of recording a video.
- Encoding is the process of compressing video to make it more efficient for streaming.
- The Streaming Server sends video to viewers.
- CDN: Distributing the video globally.
- Player: Showing the video to users.
- Monitoring: Ensures smooth playback and tracking performance.
- Fault Tolerance: Managing mistakes to ensure a consistent experience.

Diagramming the approaches:



API Design:
API (Application Programming Interface) design is the process of developing a set of rules and requirements that enable software programs to communicate and share data with one another. It focuses on the architecture, structure, and protocols that govern how clients (such as web apps, mobile apps, and other systems) interact with a service.

A well-developed API ensures:

- Ease of Use: Easy for developers to integrate.
- Scalability: Enables increased demand and new services.
- Consistency: Follows a uniform design across endpoints.
- Security: Authentication protects sensitive data and restricts access.

API Design Examples

- Upload Video

  Endpoint: POST /videos/upload

Headers: Authorization: Bearer <token>

Body:

{ "title": "Live Session", "description": "API Design Tutorial", "category": "Education" }

Response:

{ "videoId": "12345", "status": "uploaded", "url": https://cdn.streaming.com/videos/12345

- Start Live Stream
  Endpoint: POST /streams/start

  Body:

  { "videoId": "12345", "resolution": "1080p" }

  Response:

  { "streamId": "abc123", "status": "LIVE" }

  Fetch Stream Status

  Endpoint: GET /streams/{streamId}

  Response:

  { "streamId": "abc123", "status": "LIVE", "viewerCount": 1500 }

- Stop Live Stream
  Endpoint: POST /streams/stop

  Body:

  { "streamId": "abc123" }

  Response:

  { "streamId": "abc123", "status": "stopped" }

Database Design: Database design involves arranging data into organized forms for effective storage, retrieval, and management. It entails determining data needs, establishing entities and relationships, and creating tables, schemas, and constraints. A good database architecture maintains data integrity, reduces redundancy, and improves performance. The method generally comprises conceptual design (ER diagrams), logical design (schema definition), and physical design (storage strategy implementation). A well-designed database facilitates scalability, ensures data integrity, and simplifies maintenance.

Example of Database Design

Scenario: A video streaming platform.

Entities:

1. **Users**: Stores user information.
2. **Videos**: Details about uploaded videos.
3. **Streams**: Tracks live streaming sessions.

Schema Design

1. Users Table

- user_id (Primary Key)
- name
- email
- created_at

2. Videos Table

- video_id (Primary Key)
- title
- description
- user_id (Foreign Key referencing Users)
- uploaded_at

3. Streams Table

- stream_id (Primary Key)
- video_id (Foreign Key referencing Videos)
- status
- start_time
- end_time

ER Diagram

- **Users** → (1-to-many) → **Videos**
- **Videos** → (1-to-1) → **Streams**

Network Protocols:
Network protocols are defined standards that control how devices interact over a network. In system architecture, they are critical for facilitating data flow between components such as servers, clients, databases, and external systems. Proper protocol selection affects performance, dependability, security, and scalability. Protocols work at several layers of the OSI (Open Systems Interconnection) or TCP/IP architecture, providing efficient data flow and system compatibility.

Key Protocols and Use Cases:

HTTP/HTTPS: Communication between web clients and servers; HTTPS secures data.

Example: Fetching API data.

TCP/IP: Ensures reliable (TCP) and routed (IP) data transmission.

Example: Sending video data.

WebSocket: Real-time, two-way communication.

Example: Live chat or notifications.

UDP: Fast, connectionless communication with no reliability checks.

Example: Live video streaming.

FTP/SFTP: File transfer; SFTP adds encryption.

Example: Uploading files to a CDN.

DNS: Resolves domain names into IP addresses.

Example: Translating example.com to its server IP.

SSL/TLS: Encrypts data in transit for secure communication.

Example: Securing API connections.

MQTT: Lightweight messaging for IoT devices.

Example: IoT telemetry updates.

SMTP/IMAP: Sending (SMTP) and receiving (IMAP) emails.

Example: Email-based notifications.

CDN Protocols: Optimized delivery of content.

Example: Streaming high-quality video.

Choosing a Datastore for System Design:
Selecting the appropriate datastore is critical for developing a strong, efficient, and scalable system. The selection is based on unique system requirements such as data type, access patterns, scalability demands, and consistency requirements. Datastores can be roughly classified as relational or non-relational (NoSQL), with each suited to a certain use case.

Key Considerations

1. **Data Structure**:
    - **Relational Databases** (e.g., MySQL): Structured data and relationships.
    - **NoSQL Databases** (e.g., MongoDB): Semi-structured or unstructured data.
2. **Read/Write Patterns**:
    - **Key-Value Stores** (e.g., Redis): High-read systems.
    - **Log-structured Databases** (e.g., Cassandra): High-write systems.
3. **Scalability**:
    - **Horizontal Scaling**: NoSQL (MongoDB, Cassandra).
    - **Vertical Scaling**: Relational (MySQL, PostgreSQL).
4. **Consistency vs. Availability**:
    - **Consistency**: Relational (PostgreSQL).
    - **Availability**: NoSQL (DynamoDB).
5. **Latency**:
    - Use **In-Memory Databases** (e.g., Redis) for low-latency needs.
6. **Complex Queries**:
    - Relational databases for complex joins.
    - Key-Value or Document stores for simple lookups.

Uploading Raw Video Footage: Uploading raw video footage involves designing a system to handle large files efficiently, ensuring reliability, scalability, and security. This is a critical feature in platforms like YouTube or Vimeo, where users upload video content for storage, processing, and distribution.

Key Considerations

1. **File Size and Chunking**:

    - Large video files are uploaded in chunks to handle network interruptions and optimize resource usage.
    - Protocols like Multipart Uploads (e.g., AWS S3) allow resumption from failed chunks.
2. **Storage Solution**:
    - Object Storage (e.g., AWS S3, Google Cloud Storage): Scalable, cost-effective, and supports metadata tagging.
    - Cold Storage for archival needs.
3. **Authentication and Authorization**:
    - Secure access using tokens (e.g., JWT, OAuth).
    - Users must have permissions to upload.
4. **Upload Mechanism**:
    - Direct Uploads: Clients upload directly to a storage service to offload the backend.
    - Backend Proxy: Backend validates and proxies uploads for security.
5. **File Validation**:
    - Validate file format, size, and metadata before or during upload to ensure compliance.
6. **Monitoring and Logging**:
    - Track progress with real-time updates via WebSocket or polling.
    - Log upload errors for troubleshooting.
7. **Scalability**:
    - Handle concurrent uploads using load balancing and scalable cloud storage.
8. **Post-Processing**:
    - Queue the video for transcoding to optimize for playback on various devices.

MapReduce for Video Transformation:

MapReduce is a distributed programming technique designed to effectively process huge datase ts.

It is perfect for video transformation activities, which need raw video data to be converted into alternative formats, resolutions, or encoded versions to accommodate diverse devices and netw ork configurations.
During the Map Phase, the input video is separated into smaller pieces, which are processed ind ividually on dispersed nodes.
Parallel tasks include extracting frames, applying filters, and encoding into a specified format.
The Shuffle and Sort stage organizes intermediate results by grouping similar data together, ens uring that tasks are appropriately aligned for aggregation or future processing.

During the Reduce Phase, the processed chunks are aggregated or finished, such as sewing video segments back together into a single file or creating varied video resolutions. This guarantees that the finished product satisfies the specified requirements.

Using MapReduce provides considerable benefits for video transformation. It grows effortlessly by spreading processing over numerous nodes and guaranteeing fault tolerance by retrying processes that fail. Parallel processing speeds up the transformation, and the model's versatility allows for customisation of tasks like as watermarking or resolution modifications.

For example, in a video transcoding operation, a raw video file is saved in a distributed file system such as HDFS. The Map step converts video segments to certain resolutions or formats, such 720p or 1080p. The Reduce phase combines these parts, resulting in a final output suitable for playing on numerous devices. This strategy makes MapReduce an excellent option for video processing pipelines.

WebRTC vs. MPEG DASH vs. HLS:

Feature	**WebRTC**	**MPEG-DASH**	**HLS (HTTP Live Streaming)**
Purpose	Real-time, low-latency communication	Adaptive bitrate streaming over HTTP	Adaptive bitrate streaming over HTTP
Latency	Very low (typically <1 second)	Low, but higher than WebRTC	Higher than both WebRTC and MPEG-DASH (10-30 seconds)
Use Case	Video conferencing, real-time chats	Live streaming, on-demand video (adaptive quality)	Live streaming, on-demand video
Streaming Type	Peer-to-peer (P2P)	Server-client, HTTP-based	Server-client, HTTP-based

Feature	WebRTC	MPEG-DASH	HLS (HTTP Live Streaming)
Network Protocol	UDP (User Datagram Protocol)	HTTP-based, uses TCP	HTTP-based, uses TCP
Adaptability	No adaptive bitrate	Supports adaptive bitrate	Supports adaptive bitrate
Supported Devices	Web browsers (via JavaScript API)	Web browsers, mobile apps, smart TVs, etc.	Web browsers, mobile apps, smart TVs, etc.
Scalability	Limited (mainly for small groups)	Highly scalable, ideal for large audiences	Highly scalable, ideal for large audiences
Browser Support	Native support in most modern browsers	Requires player or browser support for MPEG-DASH	Native support in Safari; third-party players for other browsers
Security	Encrypted (end-to-end)	Supports DRM, encrypted streams	Supports encryption and DRM
Pros	Very low latency, no plugin required	Adaptive streaming, supports multiple bitrates	Standardized, broad device compatibility, adaptive streaming
Cons	Limited scalability, P2P issues with NAT/firewall	Requires player support, latency higher than WebRTC	High latency, not ideal for real-time apps

Content delivery networks (CDNs):
A material Delivery Network (CDN) is a collection of dispersed computers that collaborate to provide material (such as movies, photos, and web pages) to users based on their geographic location, network performance, and other criteria. CDNs are intended to increase the speed, reliability, and scalability of content delivery, allowing consumers to experience quicker load times and less downtime.

High-Level Summary: A high-level summary of content delivery networks (CDNs)
A material Delivery Network (CDN) is a network of dispersed computers designed to effectively provide material to customers while minimizing latency and assuring high availability. CDNs alleviate the stress on origin servers by catching material at many edge servers near users' locations, improving content delivery speed and reliability. This makes CDNs critical for swiftly distributing media (such as video, pictures, and web pages), especially during peak traffic times.

CDNs utilize powerful routing algorithms to route user requests to the nearest or least-loaded server, therefore improving speed and guaranteeing consistent experience across devices and locations.

Key advantages:

- Faster material Delivery: Load times were reduced by providing material from servers that were geographically closer together.
- Improved Reliability and Redundancy: Distributed servers improve fault tolerance and reduce downtime.
- Scalability: It effectively manages traffic surges, making it ideal for large-scale streaming or high-demand events.

Introduction to Low Level Design (LLD):
Low-Level Design (LLD) is the comprehensive technical design of a system's individual components or modules. It is an important phase in the software development lifecycle that specifies how the system's components will interact with one another, as well as the system's underlying logic. LLD transforms the high-level design (HLD) into code structure and specifies how to implement features or functions.
LLD divides the system architecture from HLD into smaller, more detailed components. This includes creating classes, methods, functions, data structures, algorithms, interfaces, and how various system components interact.

Key Elements of LLD:

1. Classes & Objects: Specifies classes, their attributes, methods, and relationships.
2. Methods/Functions: Details functions with parameters, return types, and logic.
3. Data Structures: Defines data structures (e.g., arrays, hash maps) based on module needs.
4. Algorithms: Provides implementation details for algorithms.
5. Error Handling: Defines error management and exceptions.
6. API Design: Outlines detailed API endpoints and formats.

Video Player Design:
Designing a video player entail developing a system capable of handling video streaming, playback, user controls, and several formats. The design must be considered for performance, flexibility, and user experience. Here's a quick description of the basic components for creating a video player.

Core Components:

1. Video Decoding: Uses libraries like FFmpeg for supporting multiple video formats.
2. UI: Includes play/pause, seek bar, volume control, full-screen toggle, and subtitles.
3. Video Streaming: Implements buffering, adaptive streaming (HLS, MPEG-DASH), and streaming protocols.
4. Playback Controls: Provides playback speed, rewind, forward, and audio synchronization.
5. Error Handling: Manages buffering issues and format errors.
6. Performance Optimization: Uses hardware acceleration and optimizes CPU/memory usage.

Engineering Requirements:

When creating a video player, various engineering criteria must be met to ensure seamless funct ioning, performance, and scalability.
These criteria include features of video rendering, user experience, streaming, and backend infr astructure.

Key Requirements:

1. Video Decoding: Support for formats (e.g., MP4, WebM) and efficient decoding using FFmpeg or hardware acceleration.
2. Streaming: Adaptive streaming (HLS, MPEG-DASH), buffering, and low-latency for live content.
3. UI & Controls: Responsive design with playback controls, subtitles, and customization.
4. Performance: Low CPU/memory usage and hardware-accelerated rendering.
5. Security: DRM and encryption for secure video delivery.
6. Scalability: CDNs for scalable delivery and backend systems for high-volume traffic.

Use Case UML Diagram:
A Use Case. A UML Diagram graphically displays a system's functional needs from the perspective of the end user. It depicts interactions between users (actors) and the system, including the different processes or use cases that the system conducts.

Key Components:

1. Actors:
    - Represent users or external systems that interact with the system.
    - E.g., User, Admin, Payment Gateway.
2. Use Cases:
    - Represents the functions or actions that the system performs.
    - E.g., Login, Play Video, Make Payment.
3. System Boundary:
    - The box that defines the scope of the system. Everything inside the box is part of the system.
4. Associations:
    - Lines connecting actors to the use cases, showing interaction.
5. Extend/Include Relationships:
    - Include: A use case that always happens as part of another.
    - Extend: A use case that is optional and extends another use case.

Class UML Diagram:

A Class UML Diagram is a structural diagram of the Unified Modeling Language (UML) that depicts the system's classes, characteristics, methods, and connections. It is useful for describing a system's static structure in object-oriented design.

Key Components:

1. Classes: Define entities (e.g., User, Video).
    - Attributes: Data fields (e.g., username, duration).
    - Methods: Functions (e.g., login(), play()).
2. Relationships:
    - Association: Solid line (e.g., User to Video).
    - Aggregation/Composition: Whole-part relationship.
    - Inheritance: Solid line with a triangle (e.g., Premium User inherits User).
    - Dependency: Dashed line (e.g., Payment depends on Subscription).

3. Visibility:
    - Public (+), Private (-), Protected (#).

Sequence UML Diagram:
A Sequence: UML Diagram is an interaction diagram that depicts how objects communicate in a time-ordered sequence. It simulates the flow of messages between objects and the sequence in which these interactions occur.

Key Components:

1. Actors: Represent external entities that interact with the system (e.g., User).
2. Objects: Represent instances of classes involved in interaction (e.g., VideoPlayer, PaymentSystem).
3. Lifelines: Vertical dashed lines representing the existence of an object over time.
4. Messages: Horizontal arrows indicating method calls or data sent between objects.
5. Activation: Thin rectangles on lifelines showing when an object is active or processing.
6. Return Messages: Dashed arrows showing responses or data returned from an operation.

Coding the Server:

Coding the server involves setting up backend logic to handle client requests, process data, and interact with databases. Using a framework like Express.js (Node.js), Spring Boot (Java), or Django (Python), the server defines routes (e.g., GET, POST) that map to functions handling business logic, such as user authentication, data retrieval, and form submissions. The server also manages connections to databases, ensures data security through proper authentication, and returns responses (usually in JSON format) to the client. Proper error handling, logging, and scalability considerations are essential for efficient server operation.

Load Balancer:

A load balancer is a network hardware or software that distributes incoming traffic among numerous servers, ensuring that no one server is overburdened with requests. This helps to improve application performance, availability, scalability, and fault tolerance. Balancing the load promotes optimal resource use and reduces reaction time while ensuring system dependability.

Where Can It Be Added?

- Web Servers: Distributes HTTP/HTTPS traffic.

- Application Servers: Manages requests in microservices.
- Database Servers: Balances queries (less common).
- Cloud Infrastructure: Distributes traffic in cloud setups.
- Global Systems: Routes traffic across regions or datacenters.

Types of Load Balancers

- Hardware: Dedicated physical devices.
- Software: Flexible solutions (e.g., NGINX, HAProxy).
- Cloud: Managed services (AWS ELB, Azure Load Balancer).
- Global: Balances traffic across regions (AWS Route 53).

Algorithms

- Round Robin: Rotates requests across servers.
- Least Connections: Sends to the least loaded server.
- Weighted Round Robin: Accounts for server capabilities.
- IP Hash: Routes based on client IP.
- Least Response Time: Prefers the fastest server.
- Resource-based: Considers CPU/memory load.

Caching

Caching is the technique of storing frequently requested data in a fast storage layer (cache) to increase application performance and minimize latency. Systems gain quicker response times and reduce backend resource burden by providing data from the cache rather than retrieving it repeatedly from a slower data source.

Where Can It Be Added?

- Database Layer: Store query results to reduce database load.
- Application Layer: Cache computation results or frequently used data.
- Content Delivery: Cache static content (e.g., images, scripts) in CDNs.
- Browser: Store recently fetched resources to speed up page loads.
- API Layer: Cache responses for frequently requested endpoints.

Types of Cache

- Client-side Cache: Stored on the user’s device (e.g., browser cache).
- Server-side Cache: Stored on the server (e.g., in-memory or disk-based).
- CDN Cache: Distributed globally for static assets like images and videos.
- Database Cache: Query results stored in systems like Redis or Memcached.

Cache Invalidation

Ensure the cashed data stays accurate and consistent:

- Time-to-Live (TTL): Data expires after a set time.
- Write-through: Updates cache immediately when the source is updated.
- Write-back: Updates the source when the cache is modified.
- Manual Invalidation: Explicitly removes stale cache entries.

Cache Eviction Policies

- Least Recently Used (LRU): Removes the least recently accessed items.
- Least Frequently Used (LFU): Removes the least accessed items overall.
- First In, First Out (FIFO): Removes the oldest cached items.
- Random: Randomly selects data to evict.
- Time-based: Evicts items based on their age.

Sharding or Data Partitioning:
Sharding, also known as Data Partitioning, is the process of breaking down a huge dataset into smaller, more manageable chunks known as shards, which are then dispersed among numerous databases or servers. This improves scalability, speed, and fault tolerance by dispersing data and workload.

Sharding Methods

- Horizontal Sharding:
    1. Split rows of a table into separate shards. Each shard contains a subset of rows.
    2. Example: Users with IDs 1–1000 in Shard 1, and 1001–2000 in Shard 2.

- Vertical Sharding:
    1. Splits tables by columns, grouping related columns into separate shards.
    2. Example: User profile details in one shard and user activity logs in another.
- Hash-based Sharding:
    1. Uses a hash function on a key (e.g., user ID) to determine the shard for each record.
    2. Ensures an even distribution of data.
- Range-based Sharding:
    1. Divides data based on continuous ranges of a key (e.g., date, ID).
    2. Example: Orders from Jan–Jun in Shard 1, and Jul–Dec in Shard 2.
- Directory-based Sharding:
    1. Maintains a lookup table mapping records to shards.

Sharding Criteria

1. Primary Key: Commonly used to determine shard allocation (e.g., user ID).
2. Geographic Location: Shards data based on region (e.g., US data vs. EU data).
3. Data Sensitivity: Separates sensitive data from non-sensitive data.
4. Time-based Partitioning: Shards data based on time ranges (e.g., yearly or monthly data).

Sharding Challenges

1. Complexity:
    - Managing multiple shards increases operational and development complexity.
2. Rebalancing:
    - Adding or removing shards requires redistributing data, which can cause downtime.
3. Cross-shard Queries:
    - Queries spanning multiple shards are slower and harder to implement.

4. Data Consistency:
    - Ensuring consistency across shards is challenging in distributed setups.
5. Fault Tolerance:
    - A single shard failure can cause partial data inaccessibility.

Index:
An index is a data structure used in databases to speed up data retrieval processes by offering a fast lookup mechanism. It functions similarly to a book's table of contents, providing quick access to certain rows in a database table without having to scan the entire table.

Indexing:
Indexing is the practice of adding and maintaining indexes to a database table to improve query speed. It improves read operations (e.g., SELECT queries) by lowering search time while potentially increasing write operation overhead (e.g., INSERT, UPDATE).

How to Create Indexes:

Indexes are typically created using SQL commands. For example:

- **Single-column Index**:

```
CREATE INDEX idx_column_name ON table_name(column_name);
```

- **Composite Index** (on multiple columns):

```
CREATE INDEX idx_composite ON table_name(column1, column2);
```

- **Unique Index**:

```
CREATE UNIQUE INDEX idx_unique ON table_name(column_name);
```

- **Full-text Index**:

```
CREATE FULLTEXT INDEX idx_fulltext ON table_name(column_name);
```

Types of Indexing

- Primary Index:
    - Automatically created on the primary key column of a table.
    - Ensures uniqueness and fast access to rows.
- Unique Index:
    - Prevents duplicate values in the indexed column(s).
    - Often used for fields like email or username.

- Clustered Index:
    - Physically sorts the table data based on the index.
    - Only one clustered index is allowed per table.
    - Example: Primary key index.
- Non-Clustered Index:
    - Keeps the index separate from the table data.
    - Allows multiple non-clustered indexes on a table.
- Composite Index:
    - Index on multiple columns, improving performance for queries using those columns.
- Full-text Index:
    - Specialized for text-based data, allowing efficient searching of large text fields.
- Spatial Index:
    - Used for geographic data (e.g., latitude, longitude) to optimize spatial queries.

What is Proxy

A proxy acts as an intermediary between a client and a server, facilitating or modifying requests and responses.

What is a Proxy Server

A proxy server is a network device or software that routes client requests to the destination server and relays responses back, often adding security, caching, or anonymity.

Benefits of Using a Proxy Server

1. Improved Security: Masks client IPs, protecting identity and preventing attacks.
2. Anonymity: Hides user details, enabling private browsing.
3. Caching: Stores responses to reduce server load and improve speed.
4. Access Control: Filters traffic, blocking unauthorized access or content.
5. Load Balancing: Distributes requests across multiple servers.

Tasks of a Proxy Server

1. Request Forwarding: Routes client requests to appropriate servers.
2. Content Filtering: Blocks or allows specific content based on policies.
3. Traffic Encryption: Secures data transfer with encryption.
4. Performance Optimization: Caches data for faster access.
5. Logging and Monitoring: Tracks traffic for analytics and troubleshooting.

Messaging Queue:

A Messaging Queue is a communication mechanism that enables asynchronous message transmission across computers, allowing components to be decoupled in distributed applications. It works by having a producer submit messages to a queue, where they are held until a consumer collects and processes them. This provides consistent message transmission even when one system is momentarily unavailable. Benefits include increased scalability, fault tolerance, and adaptability to changing workloads. RabbitMQ, Apache Kafka, Amazon SQS, Azure Service Bus, and ActiveMQ are among the most popular message queues, each catering to a distinct use case such as event streaming or task queueing.

SQL (Relational Databases)

- Use Case: Ideal for structured data and applications requiring complex queries and transactions (e.g., banking, ERP systems).
- Schema: Follows a fixed schema with predefined tables and relationships.
- Scalability: Vertically scalable (by adding resources to a single server).
- Examples: MySQL, PostgreSQL, Oracle Database.

NoSQL (Non-Relational Databases)

- Use Case: Suited for unstructured, semi-structured, or rapidly evolving data (e.g., IoT, real-time analytics, content management).
- Schema: Flexible schema, allowing dynamic and varied data models.
- Scalability: Horizontally scalable (by adding servers).
- Examples: MongoDB, Cassandra, Redis, Couchbase.

Key Factors to Consider

- Data Structure: Choose SQL for structured data; NoSQL for flexible data formats.

- Scalability Needs: Use NoSQL for horizontal scaling; SQL for vertical scaling.
- Consistency vs. Availability: SQL prioritizes consistency; NoSQL often focuses on availability and partition tolerance (CAP theorem).
- Transaction Requirements: SQL is better for applications requiring ACID transactions.

Monolithic Vs Microservice:

Aspect	**Monolithic**	**Microservices**
Architecture	Single, unified codebase	Independent, loosely coupled services
Development	Simple to develop and deploy	Complex requires coordination
Scalability	Scales as a whole	Scales individual services independently
Flexibility	Limited (all components use the same tech)	High (different technologies per service)
Fault Isolation	A failure affects the entire application	Faults are isolated to specific services
Maintenance	Harder to maintain as it grows	Easier to maintain as services are small and focused
Testing	Easier due to a single codebase	More complex due to multiple services
Deployment	One deployment for the entire app	Multiple independent deployments
Best For	Small to medium-sized applications	Large, complex, or rapidly evolving applications

REST API:

A **REST API** (Representational State Transfer) is an architectural style that allows communication between different systems over the internet using standard HTTP methods like GET, POST, PUT, DELETE, and PATCH. It operates on the principle of statelessness, where each request contains all necessary information, and does not rely on the server to store client session states. REST APIs use a client-server model, with each system interacting through a consistent interface to perform operations on resources (e.g., data, objects). They are cacheable, scalable, and flexible, making them suitable for web and mobile applications that require easy access to backend data or services.

Common HTTP Methods Used in REST APIs

- GET: Retrieve data from the server
- POST: Create new data on the server
- PUT: Update existing data
- DELETE: Remove data from the server
- PATCH: Partially updated data

REST API:

A REST API (Representational State Transfer) is an architectural style that allows communication between different systems over the internet using standard HTTP methods like GET, POST, PUT, DELETE, and PATCH. It operates on the principle of statelessness, where each request contains all necessary information, and does not rely on the server to store client session states. REST APIs use a client-server model, with each system interacting through a consistent interface to perform operations on resources (e.g., data, objects). They are cacheable, scalable, and flexible, making them suitable for web and mobile applications that require easy access to backend data or services.

Hashing in System Design:

Hashing is a method for creating a fixed-size value (hash) from an input string or data using a hash function. This hash value is a unique identifier for the input data. Hashing allows for quick data retrieval and storage while preserving data integrity and security. It is very useful in system architecture for operations such as database indexing, password storing, caching, and load balancing.

A hash function uses a mathematical technique to turn input into a hashed value, assuring predictable and consistent results. Furthermore, in security-sensitive cases, hashing helps maintain data integrity during transmission, guaranteeing that the message is tamper-proof for the intended receiver.

Consistent Hashing in System Design:
Consistent hashing is a distributed hashing technique that distributes data uniformly among a system's dynamic nodes (servers or caches). It reduces the impact of adding or deleting nodes, guaranteeing that only a small percentage of the data must be redistributed, as opposed to typical hashing techniques.

The approach maps data and nodes to a circular hash space (0 to 2^{32}).
Data is allocated to the first node whose hash value equals or exceeds its own.
When a node is added, it assumes responsibility for a piece of the data formerly held by its succ essor.Similarly, when a node is eliminated, its data is transferred to its replacement, ensuring bal ance and minimizing interruption.

Consistent hashing has various advantages, including scalability, as it allows for the dynamic addition and removal of nodes with minimum cost. It also provides load balancing by equally distributing data among nodes and improves fault tolerance by redistributing data when a node fails. Consistent hashing is used in distributed databases to partition data, caching systems such as Redis or Memcached, and content delivery networks (CDNs) to allocate resources efficiently. For example, in a distributed cache, consistent hashing ensures that adding a new cache server impacts just a subset of the keys, reducing cache invalidation and increasing system efficiency.

Kafka in System Design:

Kafka is a highly scalable, distributed streaming platform built for real-time data processing and message brokering. It enables systems to publish and subscribe to record streams and is widely used in the development of data pipelines, streaming applications, and decoupling systems for greater scalability and flexibility.

Kafka is quick because it splits data records into chunks, allowing for effective compression and lower I/O latency. This end-to-end batching, which runs from the producer to the consumer via Kafka topics, provides little overhead and good performance. Kafka is also robust and scalable, preserving all published records for as long as disk space allows or until a retention limit is established, making it perfect for real-time data streaming into data lakes, analytics systems, or applications.

LRU

In system architecture, an LRU (Least Recently Used) cache is a caching technique that deletes the item with the fewest recent accesses when the cache is full. It optimizes memory use by prioritizing frequently requested data.

Typically, an LRU cache is built using a doubly linked list and a hash map. The doubly linked list keeps the order of pages, with the most recently used (MRU) page at the top and the least recently used (LRU) page at the end. The hash map allows O(1) access to cached pages by storing the page number as the key and the reference to the doubly linked list as the value.

In a sequence of cache operations, adding a new page pushes it to the top of the list, but reading an existing page reorders it to the top. If the cache is full, the page at the end (the least recently used) gets evicted. This architecture enables rapid lookups, updates, and evictions.

Apache Hadoop

Apache Hadoop is an open-source platform for distributed storage and processing of large datasets over a network of computers. It is intended to handle large-scale data and computing operations effectively, especially in the face of hardware failures, which it automatically controls.

Hadoop comprises of two major components:

HDFS (Hadoop Distributed File System) manages distributed storage by dividing files into big blocks and distributing them among cluster nodes.
The MapReduce programming model handles processing by sending code to the nodes where the data is stored, allowing for parallel computing and exploiting data proximity to decrease data travel.
Hadoop's design provides scalability, fault tolerance, and efficiency, making it perfect for Big Data analytics and processing applications.

HDFS (Hadoop Distributed File System

HDFS (Hadoop Distributed File System) is a distributed storage system designed to store and manage large files across multiple machines. It ensures high reliability and fault tolerance by replicating data across nodes, typically using a replication factor of 3: two replicas on the same rack and one on a different rack.

HDFS enables data nodes to communicate for rebalancing and maintaining replication levels in case of failures. Optimized for low-cost hardware, HDFS provides scalable and fault-tolerant storage, making it suitable for Big Data applications.

HBase

HBase is an open-source, distributed, column-oriented database modeled after Google's BigTable, designed to work seamlessly with Hadoop and HDFS. It provides a fault-tolerant, high-throughput solution for storing and retrieving large quantities of sparse, structured data.

Key features include compression, in-memory operations, and Bloom filters on a per-column basis. HBase supports multiple APIs (Java, REST, Avro, and Thrift) and integrates with Hadoop's MapReduce for processing. It is particularly suited for high-speed read/write operations on large datasets, offering fast lookups for massive tables.

Major Components:

1. ZooKeeper: Coordinates distributed systems.
2. HMaster Server: Manages and assigns regions.
3. Region Server: Handles read/write requests for specific regions.

HBase excels in handling diverse data types and is widely used in scenarios requiring high scalability and performance.

ZooKeeper

ZooKeeper is a distributed coordination service that ensures synchronization, configuration management, and naming across large-scale, distributed systems. It helps maintain consistency in distributed applications by providing centralized coordination among different nodes in the system.

Key Features:

1. Configuration Management: Centralized management of configuration data across multiple nodes.
2. Naming Service: Assigns unique names to the cluster of servers, making it easier to manage large systems.
3. Distributed Synchronization: Provides mechanisms like locks, barriers, and queues to synchronize processes across nodes.
4. Group Services: Facilitates leader election and group membership management.

Replication:

ZooKeeper runs as a cluster, consisting of one leader and multiple followers. All followers maintain a copy of the state, so if the leader fails, one of the followers can be elected as the new leader.

Consistency Guarantees:

- Sequential Consistency: Ensures operations are executed in the order they were received.
- Atomicity: Guarantees that updates are performed completely or not at all.

ZooKeeper uses Znodes, which are data containers that hold information, and can have children or both data and children. It is crucial for maintaining system reliability and coordination in large, distributed systems.

Apache Solr

Apache Solr is an open-source enterprise search tool based on Apache Lucene. It offers sophisticated capabilities for full-text search, faceted search, geographical searches, and text analysis, allowing for scalable and fast indexing and querying of huge datasets. Solr enables distributed indexing and replication, which guarantees high availability and fault tolerance. It is often used in applications that require advanced search capabilities, such as web search engines, analytics platforms, and enterprise search solutions, and provides quick and reliable querying even for large datasets.

Apache Cassandra

Apache Cassandra is an open-source, highly scalable, distributed NoSQL database that can handle massive volumes of data over several commodity computers with no single point of failure. It is designed for write-intensive applications and is noted for its capacity to handle large amounts of organized and unstructured data with low latency. Cassandra's peer-to-peer design enables high availability, fault tolerance, and horizontal scalability. It employs a ring-based architecture in which each node in the cluster is equal, and data is automatically replicated across numerous nodes to ensure durability and uptime. Cassandra is frequently used in applications requiring quick data writes and high availability, such as real-time analytics, recommendation engines, and Internet of Things systems.

URL shortener

A URL shortener is a service that reduces lengthy URLs (web addresses) to shorter, more manageable ones. These abbreviated URLs are easier to communicate and may be utilized in circumstances with limited space, such as social media postings or SMS messages. When a user clicks on a shortened URL, they are taken to the original, lengthier URL. URL shorteners generate a unique identifier, usually alphanumeric, and attach it to a base domain, resulting in a short link. Bitly, TinyURL, and Google's previous Goo.gl (since defunct) are some of the most popular URL shorteners. URL shorteners also include capabilities such as click tracking, analytics, and link expiry.

Designing a Dropbox-like system

Upload/Download Content: Use distributed storage (e.g., S3, HDFS) with encryption for secure uploads and downloads. Implement chunking for large files to improve speed.

- Device Syncing: Implement real-time syncing using event-driven architecture (WebSockets/push notifications) with conflict resolution for simultaneous edits.
- Content Sharing: Implement role-based access control (RBAC) and allow link sharing with customizable permissions (view/edit).
- File Editing: Support file locking and collaborative editing for simultaneous modifications.
- Non-Premium Storage Limits: Enforce storage quotas for free users and send notifications when limits are approached.
- REST APIs: Provide secure APIs for uploading/downloading files, with rate-limiting and authentication via OAuth or API keys.
- Analytics: Track user activity, performance metrics, and usage patterns for system optimization.
- File Versioning: Store multiple versions of files, with options for users to revert or delete old versions.
- Scalability and Reliability: Use horizontal scaling, data replication, and caching to ensure high availability and performance.

HTTP VS HTTPS

- HTTP is an insecure protocol that transfers data in plaintext, making it vulnerable to eavesdropping and tampering. It operates on port 80 and does not provide encryption or authentication.
- HTTPS is the secure version of HTTP, using SSL/TLS encryption to ensure data confidentiality and integrity. It operates on port 443 and provides authentication through SSL certificates, protecting data from interception and tampering.

Key Differences:

- Security: HTTPS is secure with encryption, while HTTP is not.
- Port: HTTP uses port 80, while HTTPS uses port 443.

- Encryption: HTTPS uses SSL/TLS encryption, HTTP does not.
- Authentication: HTTPS provides authentication; HTTP does not.

Cap Theorem

The CAP Theorem describes the trade-offs that distributed systems must make among three fundamental properties:

Consistency (C): Each read operation returns either the most recent writing or an error. This assures that all nodes in the system have the same data at any moment.

Availability (A): Every request receives a response, whether successful or unsuccessful, however it may not represent the most recent write. Availability assures that the system answers to all queries, but it does not guarantee that the returned data is current.

Partition Tolerance (P): The system will continue to function even if network partitions (i.e., communication failures) occur between nodes. This means that the system should continue to function even if portions of the network are unable to interact with one another.

When a partition occurs:

If you select Consistency, the system may return an error or timeout because it is unable to update all nodes owing to partitioning.
If you select Availability, the system will provide an answer based on the most current accessible data, even if it is not the most recent version (and may be inconsistent).
In the absence of network segmentation, a system can ensure consistency and availability.