**System**

**Design**

Aishwarya Pol

**🔧 System Design Fundamentals**

* System design involves integrating multiple components to ensure scalability and efficiency.
* **Key Components**:
  + **Load Balancer**: Distributes incoming traffic across servers to prevent overload.
  + **API Gateway**: Central access point for routing API requests to correct services.
  + **Caching Systems**: Store frequently accessed data to reduce backend load and speed up responses.
* **Scaling Techniques**:
  + **Horizontal Scaling**: Adding more servers to handle increased load.
  + **Vertical Scaling**: Increasing resources (CPU, RAM) of existing servers.
* **DNS (Domain Name System)**:
  + Converts domain names into IP addresses, allowing users to connect to servers.

**📈 Vertical Scaling Challenges**

* Enhances a single server’s capability by upgrading hardware.
* **Limitations**:
  + Causes downtime during upgrades.
  + Not suitable for systems needing 24/7 uptime.
  + Overloaded servers may crash due to limited resources.
* Requires predicting traffic accurately to avoid waste or performance issues.

**⚙️ Horizontal Scaling & Load Balancing**

* Adds multiple servers to share the load, avoiding downtime.
* **Load Balancer**:
  + Evenly distributes traffic to prevent server overload.
  + Uses algorithms like **Round Robin**, **Least Connections**, etc.
  + Increases reliability, performance, and fault tolerance.

**🔁 Microservices & Routing**

* Routes requests to specific services based on type or endpoint.
* **Load Balancer**: Essential for distributing requests in microservices architecture.
* **API Gateway**:
  + Centralized point for managing and directing API calls.
  + Ensures secure and accurate request routing.
* **Batch Processing**:
  + Efficient for large-scale operations like sending emails or handling bulk uploads.

**🔄 Asynchronous Communication**

* Allows services to proceed without waiting, improving performance.
* **Synchronous**: Waits for tasks (e.g., email) to complete—can cause delays.
* **Asynchronous**: Proceeds without waiting—better for real-time systems like payments.
* **Queue System**:
  + First-In-First-Out (FIFO) processing for orderly and efficient task handling.
* **Long Polling**:
  + Optimizes event detection and reduces the need for constant API calls.

**📬 Event-Driven Architecture**

* Systems respond to events in real time rather than polling continuously.
* **SNS (Simple Notification Service)**:
  + Broadcasts events to multiple subscribers (e.g., SMS, email).
* **SQS (Simple Queue Service)**:
  + Guarantees a message is processed by only one consumer.
* **Challenges**:
  + SNS lacks acknowledgment; custom retry mechanisms are needed to avoid dropped events.
* **Fan-Out Pattern**:
  + Sends the same notification to multiple services at once.

**🕸️ Fan-Out Architecture & Optimization**

* Improves scalability by invoking multiple services simultaneously.
* **Rate Limiting**:
  + Prevents excessive requests and protects services from overload.
* **Database Optimization**:
  + **Read Replicas**: Handle read operations, reducing the load on the main database.
  + **Caching (e.g., Redis)**: Reduces database hits and improves speed.

**🌍 Content Delivery with CloudFront**

* **CloudFront (CDN)**:
  + Globally distributed servers deliver content quickly and reliably.
  + Reduces latency by serving content from the nearest edge location.
* **Anycast Routing**:
  + Routes requests to the closest server using a single IP address.
* **Caching**:
  + Speeds up repeat access and reduces backend server load.

**🧠 Conclusion: Best Practices in System Design**

* Use **horizontal scaling** and **load balancing** for high availability.
* Leverage **CDNs** and **caching** for faster performance.
* Prefer **asynchronous** and **event-driven architectures** for real-time systems.
* Utilize **rate limiting**, **queuing**, and **replica strategies** to handle scale and reliability

**Advanced System Design: Key Concepts and Insights**

**1. Importance of System Design**

* System design evolves over time depending on the use case, traffic pattern, and scaling needs of each company.
* Scalability and fault tolerance are essential goals but must be balanced with cost-efficiency.
* A one-size-fits-all approach doesn't work—platforms like YouTube, Netflix, and Hotstar need tailored solutions due to different user behaviors and load conditions.

**2. Traffic Patterns and Scaling Strategies**

* **Burst Traffic**: Sudden spikes in user activity (e.g., during live events or content releases) can crash systems if not handled properly.
* **Gradual Growth vs. Sudden Spikes**:
  + Gradual increases allow progressive scaling.
  + Sudden spikes (e.g., viral content) need predictive strategies to avoid downtime.
* Companies use **historical data and predictive analytics** to anticipate and prepare for high loads.

**3. Horizontal vs. Vertical Scaling**

* **Horizontal Scaling**: Adding more servers to distribute load.
* **Vertical Scaling**: Increasing resources (CPU, RAM) on existing servers.
* Scaling decisions are made based on traffic trends and system thresholds to prevent failures.

**4. Predictive Scaling**

* Platforms like Netflix and Hotstar pre-scale their systems based on anticipated events (e.g., movie premieres or cricket matches).
* YouTube adjusts server loads dynamically, especially during festive seasons or major events.
* Understanding user behavior and content popularity is crucial for scaling decisions.

**5. Serverless Architectures**

* **AWS Lambda** offers event-driven, auto-scaling capabilities without server management.
* **Cold Starts**: Initial latency occurs when functions are triggered after being idle, affecting user experience.
* Continuous traffic helps "warm" functions to avoid cold start delays.
* Serverless is cost-effective (e.g., free tier), but may lead to **vendor lock-in** at scale.

**6. Statelessness in Serverless Design**

* Serverless systems do not retain state between requests, which makes them highly scalable but complex in terms of session handling and data retention.
* High volumes can stress databases and backend services if not managed properly.

**7. Virtual Machines vs. Containers**

* **Virtual Machines (VMs)**: Solve dependency issues but are heavy on system resources and slower to scale.
* **Containers** (e.g., Docker): Lightweight, share host OS, and are easier to scale and manage.
* Containers ensure consistency across environments and support microservices architecture.

**8. Container Orchestration with Kubernetes**

* **Kubernetes**: An open-source orchestration tool for automating deployment, scaling, and management of containerized applications.
* Handles tasks like:
  + Rolling updates (replacing old containers with new ones seamlessly)
  + Load balancing
  + Resource monitoring and fault recovery
* Inspired by Google’s internal system, Borg.

**9. Load Testing and System Readiness**

* Companies like Hotstar conduct **load and stress testing** before big events to ensure systems are prepared for high demand.
* Streaming platforms must coordinate between content delivery networks (CDNs), server clusters, and scaling mechanisms to deliver smooth playback (especially for 4K videos).

**10. Deployment Evolution**

* The evolution from bare-metal servers to VMs, containers, and now Kubernetes has drastically improved scalability and deployment speed.
* Modern deployment strategies focus on fault tolerance, zero downtime updates, and high availability during user peaks.