Meritshot
EDUCATION

TOP 25

SYSTEM DESIGN

INTERVIEW QUESTIONS

FROM

Google

DESIGNED FOR EXPERIENCED PROFESSIONALS

Repost

1. Design a URL shortening service (like Bitly).
Answer:

The URL shortening service involves mapping long URLs to shorter ones. We would start by using a hash function or base62 encoding to generate a unique identifier. To handle collisions, we could store the mappings in a distributed hash table with a fallback mechanism (like appending numbers or using a second hash). The service must be scalable, so we’d use a NoSQL database like Cassandra to store mappings, ensuring high availability. Additionally, we'd use caching (e.g., Redis) to reduce the load for frequently accessed URLs.

2. Design a cache system (e.g., for a website).
Answer:
To design a cache, we would implement a Least Recently Used (LRU) eviction strategy, storing frequently accessed data in memory (e.g., using Redis or Memcached). Data that isn't accessed often would be evicted to make space for newer data. A cache miss would trigger a fetch from the backend database. To ensure cache consistency, we would use TTL (Time to Live) and implement cache invalidation strategies like write-through or write-behind caching depending on the system's needs.

3. Design a file storage system like Google Drive.
Answer:
For file storage, we need to handle large-scale data efficiently. We would store metadata (file names, sizes, paths) in a distributed database like DynamoDB. The files themselves would be stored in a distributed object storage system like Amazon S3. We would implement versioning to allow users to access previous versions of files. For performance, we would use a content delivery network (CDN) for fast access to frequently requested files and encrypt data both at rest and in transit for security.

4. Design a notification system.

Answer:

The notification system must support sending notifications via different channels (e.g., push, email, SMS). We would use a message queue (e.g., Kafka or RabbitMQ) to decouple the notification generation from the actual sending. Notifications would be stored in a scalable database (e.g., Cassandra) to handle bursts of traffic. To scale, we could partition notifications by user and type and ensure message deduplication to avoid redundant notifications.

5. Design a distributed messaging system (like WhatsApp).

Answer:

For a messaging system, we'd focus on real-time message delivery and persistence. The system should support message queuing and notifications. For this, we'd use Kafka or Google Pub/Sub for handling real-time streams, and a distributed NoSQL database like Cassandra for message storage. We'd implement message acknowledgment and delivery status tracking. To scale, we could shard messages by user or conversation, and use encryption for secure message transmission.

6. Design a content recommendation system (like Netflix).
Answer:
A recommendation system can be based on collaborative filtering, content-based filtering, or hybrid models. We would store user preferences, interactions, and item metadata in a distributed database like DynamoDB. For the recommendation algorithm, we could use machine learning models (e.g., matrix factorization or deep learning). To ensure low-latency recommendations, we would cache popular recommendations and personalize suggestions using real-time data processing frameworks like Apache Flink or Spark Streaming.

7. Design a search engine.
Answer:
A search engine needs an indexing system to retrieve data efficiently. We would start by crawling web pages and using a distributed inverted index (Elasticsearch or Solr) to map keywords to documents. For ranking, we could implement algorithms like PageRank or BM25. To scale, we would partition the index across multiple servers, with redundancy to handle failover. Caching would be essential for frequently queried terms, and we would use a distributed system (e.g., Hadoop) for web crawling.

8. Design an e-commerce platform (like Amazon).
Answer:
An e-commerce platform needs to handle inventory management, product listings, shopping carts, payments, and order fulfillment. For scalability, we’d separate these concerns into microservices, with each service (e.g., product catalog, payment gateway) independently scalable. We'd store product data in a distributed database like PostgreSQL, with caching (e.g., Redis) for fast product lookups. The payment system would integrate with third-party providers, and we’d implement eventual consistency for order processing using queues like Kafka.

9. Design a ride-sharing application (like Uber).
Answer:
For a ride-sharing app, we need to track real-time driver locations and match riders with drivers efficiently. We'd use a distributed system like Kafka to manage location updates and ride requests. The matching algorithm would consider factors like proximity, traffic, and rider preferences. We would store ride data in a distributed database like Cassandra and use a caching layer (e.g., Redis) to store the real-time locations of drivers. Scalability would be achieved by partitioning the data by region or driver ID.

10. Design a social media platform (like Facebook).

Answer:

For a social media platform, we need to support user profiles, newsfeeds, messaging, and media sharing. We would use a NoSQL database (e.g., MongoDB or Cassandra) for scalability. For the newsfeed, we would use a combination of precomputed rankings and a personalized feed based on user interactions (using machine learning). To handle real-time messaging, we'd use WebSockets or a message broker like Kafka. User data would be cached to improve read performance, and media would be stored in a distributed object storage system like S3.

11. Design a payment gateway (like PayPal).

Answer:

A payment gateway must handle secure, real-time transactions, fraud detection, and integration with external banking systems. We would implement end-to-end encryption for security and use microservices to handle different payment methods (credit card, PayPal, etc.). For scalability, we’d store transaction data in a distributed, fault-tolerant database like PostgreSQL with replication. To handle high throughput, we would use message queues to ensure transaction processing is decoupled and can scale independently.

12. Design a system for real-time analytics.

Answer:

For real-time analytics, we would use stream processing frameworks like Apache Kafka and Apache Flink. Data would be ingested in real-time and processed in near-real-time to provide insights. We would store the results in a scalable, low-latency database (e.g., ClickHouse or BigQuery). To handle bursts of traffic, we'd implement auto-scaling mechanisms and use distributed processing to ensure high availability and fault tolerance.

13. Design a video streaming platform (like YouTube).

Answer:

For video streaming, we would use a content delivery network (CDN) to cache videos closer to the user's location for fast delivery. Videos would be stored in an object storage system like S3, and we would use HLS (HTTP Live Streaming) for adaptive bitrate streaming. Metadata and user data would be stored in a distributed database (e.g., PostgreSQL). To scale, we would partition the videos by category and user-generated content would be processed asynchronously via message queues.

14. Design a multi-tenant application.
Answer:
A multi-tenant application serves multiple customers (tenants) while isolating their data. For this, we would choose between a shared database with tenant-specific schema or a database-per-tenant approach, depending on scale. Each tenant's data would be partitioned to avoid cross-tenant data leakage. We would use API gateways to handle tenant-specific routing and implement rate limiting to prevent abuse. We'd ensure tenant data isolation using access control and encryption.

15. Design an analytics dashboard for a large dataset.
Answer:
To design an analytics dashboard, we would use a data warehouse (e.g., Redshift, BigQuery) to store and process large datasets. The dashboard would be connected to a backend API that queries the data warehouse and aggregates data based on user queries. Caching would be implemented to speed up frequent queries. To scale, we would partition the data by time (e.g., daily or monthly partitions) and precompute aggregations. The front end would use a data visualization library (e.g., D3.js, Chart.js).

16. Design an email delivery system.

Answer:

An email delivery system must handle high throughput while ensuring reliability and deliverability. We would use a message queue (e.g., RabbitMQ) to decouple email sending from the backend system, allowing us to scale independently. The system would retry sending failed emails with exponential backoff. To ensure delivery, we would implement SMTP servers and third-party APIs (e.g., SendGrid) for email dispatching, and use tracking systems to monitor delivery success.

17. Design a photo-sharing application (like Instagram).

Answer:

For a photo-sharing app, images would be stored in an object storage system (e.g., AWS S3), and metadata (captions, tags, etc.) would be stored in a relational database. We would implement a scalable user authentication system with OAuth, and use a CDN to cache images closer to the user for fast delivery. To handle high traffic, we'd shard data based on user IDs and employ horizontal scaling. For real-time interaction, we would use WebSockets for features like comments and likes.

18. Design a real-time collaborative document editing system (like Google Docs).

Answer:

A real-time collaborative document editing system requires syncing changes across multiple users in real-time. We would implement operational transformation (OT) or conflict-free replicated data types (CRDT) algorithms to ensure consistency and resolve conflicts. Changes would be broadcasted through a WebSocket connection. To store documents, we'd use a distributed database like MongoDB. To scale, we would implement sharding and replication to handle concurrent users.

19. Design a job scheduling system (like cron).

Answer:

A job scheduling system would need to manage job executions, retries, and failures. We would design a service that periodically checks for jobs to run, using a priority queue to schedule them. To ensure fault tolerance, jobs would be stored in a database with status tracking, and retries would be implemented with exponential backoff. For scalability, we would shard jobs by type or priority, and use a distributed task queue (e.g., Celery) to manage execution.

20. Design a scalable logging system.

Answer:

A scalable logging system needs to handle high throughput and provide easy search capabilities. We would collect logs using a logging framework like Fluentd and store them in a distributed system like Elasticsearch, which allows for real-time querying. Logs would be partitioned by service and timestamp, and we would use log aggregation and indexing for quick search. To handle scale, we'd use sharding and implement retention policies to delete old logs.

21. Design a chat application (like Slack).

Answer:

For a chat app, we need real-time message delivery and a scalable backend. We would use WebSockets for real-time messaging and a distributed database like Cassandra for message storage. To handle large-scale teams, we'd implement partitioning by user or chat room and replicate data across data centers for fault tolerance. To ensure scalability, we would use microservices for different functionalities (user management, message history, notifications).

22. Design a news feed system (like Twitter).

Answer:

A news feed system needs to deliver real-time updates to users. We would use a message queue (e.g., Kafka) for real-time updates and a NoSQL database like DynamoDB for storing user posts. To generate the feed, we would use a mix of push and pull models, with precomputed rankings and personalized feeds. To handle scaling, we would partition feeds by user and use caching for frequently accessed data.

23. Design a distributed rate limiter.

Answer:

A distributed rate limiter ensures that requests from clients don’t exceed a predefined limit. We could use Redis with a sliding window algorithm to track request counts in real-time. Each request would update a counter in Redis, and if the counter exceeds the limit, it would be rejected. To ensure consistency across nodes, we could use Redis’s Pub/Sub feature to propagate state changes.

24. Design a voting system (e.g., for an election).

Answer:

A voting system needs to ensure fairness, scalability, and security. We would use a distributed database to store votes, ensuring that each voter can only vote once by using unique voter IDs and token validation. For scalability, we could shard data by election type or region. To prevent fraud, we would implement encryption and audit logs. The result calculation would be done asynchronously to ensure high availability.

25. Design a system to detect fraud in real-time transactions.

Answer:

Fraud detection requires real-time transaction monitoring. We would use a stream processing platform like Apache Kafka or Flink to analyze transactions in real-time. For fraud detection, we would implement machine learning models trained on historical data to identify patterns. To scale, we would partition the transaction data based on region or account ID and use caching for frequently analyzed patterns.

Meritshot

EDUCATION

Your one-step destination for your Career Upskilling

Lets make a community of 20k+ learnerrs

meritshoteducation