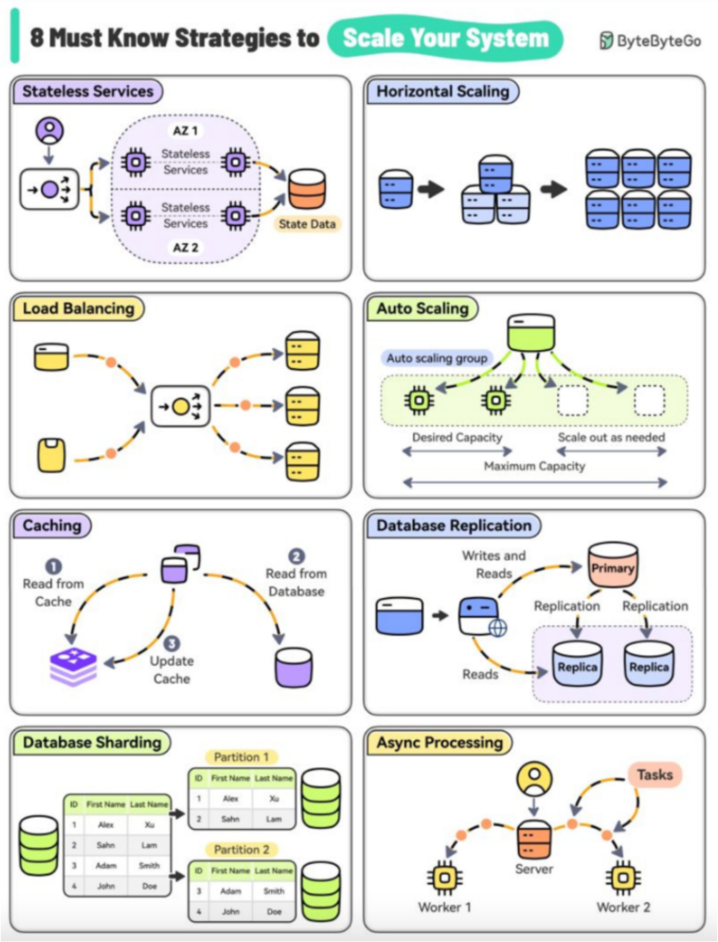
adityakrverma ★ 738 Last Edit: December 5, 2024 10:33 AM 314 VIEWS

5

```
# 01. Generic System Design NFR Concepts
# 02. NFR fulfillment specific to BT (Imp)
# 03. System Design Questions specific to BT (Imp)
# 04. TRADE-OFFs in Distributed System Design
# 05. PROS & CONS of Microservices
# 06: CHALLANGES in Distributed System Design
# 07. BEST PRACTICES for designing microservices
# 08. Scaling the Stateful Services
# 09. Scaling communication b/w microservices
# 10. SD Questions for Senior Roles
# 11. Implement Rate limiter, Consistent Hashing
# 12. Addendum ( Building Blocks Concepts )
```



I.) Generic System Design NFR Concepts:
++++++++++++++++++++++++++++++++++

Non-Functional Requirements (NFRs) is crucial to demonstrate your understanding of how a system performs under real-world conditions. Here are some key NFRs that you should consider discussing during a system design interview: 10/20/24

1. Scalability

- Definition: The ability of the system to handle increasing loads or to be expanded to accommodate growth.
- Types:
  - Vertical Scaling: Adding more power (CPU, RAM) to an existing server.
  - Horizontal Scaling: Adding more servers to distribute load.
- Discussion Points:
  - Question: How would you scale your system to handle millions of users?
    `Answer: I would use horizontal scaling to add more instances of services across multiple servers to distribute the load. For example, deploying the services on Kubernetes or AWS EC2 Auto Scaling groups ensures that as the traffic grows, more instances are automatically added. I would also implement load balancing using tools like NGINX, AWS ELB, or Google Cloud Load Balancer to distribute incoming`

requests efficiently across the available instances.

- Question: How does your system handle sudden traffic spikes (e.g., during a flash sale)?
  Answer: To handle sudden traffic spikes, I would implement auto-scaling based on CPU or request load metrics. Additionally, I would use a queueing mechanism (like RabbitMQ or AWS SQS) to buffer requests if the system gets overwhelmed. For read-heavy workloads, caching layers like Redis or Memcached can offload pressure from the database.

Discuss load balancers, sharding, and replication.

---

2. Performance

- Definition: The responsiveness and speed of the system under load.
- Key Metrics:
  - Latency: The time it takes for a request to receive a response.
  - Throughput: The number of requests that the system can process in a given time frame.
- Discussion Points:
  - Question: How would you minimize latency (e.g., through caching, using Content Delivery Networks)?

    Answer: I would minimize latency by implementing caching at different layers: * Database caching using in-memory databases like Redis to cache frequent queries. * CDN (Content Delivery Network) for static assets (e.g., images, JavaScript, CSS) to reduce latency by serving content from the nearest edge server. * Application-level caching to cache API responses for frequently requested data. Additionally, I would optimize database queries using indexes and minimize network latency by colocating services in the same region
  - Question: How does the system handle high throughput (e.g., batch processing, message queues)?
    Answer: To handle high throughput, I would implement message queues like Kafka or RabbitMQ to distribute tasks across multiple consumers asynchronously. For example, user-facing requests can be processed in real-time, while background jobs (such as data processing or notifications) are offloaded to the queue. For batch operations, I would use a distributed framework like Apache Spark for efficient processing of large datasets.

---

3. Reliability

- Definition: The system's ability to perform consistently and correctly over time without failure.
- Key Concepts:
  - Fault Tolerance: The system's ability to continue functioning even if some components fail.
  - High Availability (HA): Ensuring the system is available a high percentage of the time (e.g., 99.99% uptime).
- Discussion Points:
  - Question: How does your system handle failures in individual components (e.g., using failover mechanisms, backups)?
    Answer: I would design the system using redundant components so that if one instance of a service fails, another can take over without disrupting the user experience. This includes database replication (e.g., using master-slave or multi-primary configurations) and load balancers for service redundancy. In case of catastrophic failures, I would rely on automated backups of critical data, with replication across multiple data centers.
  - Question: How would you ensure reliability through redundancy (e.g., data replication, multi-region deployment)?

    Answer: I would ensure reliability by deploying services across multiple regions using geo-replication. This would allow failover to another region if the primary region goes down. Additionally, I would use database replication strategies like multi-master replication for critical databases, so data is replicated in real-time across regions.

---

4. Availability

- Definition: The percentage of time the system is operational and accessible.
- Key Concepts:
  - SLA (Service Level Agreement): Defines expected availability (e.g., "four nines" = 99.99% uptime).
- Discussion Points:
  - Question: How would you design for high availability (e.g., using load balancing, geo-redundancy)?
    Answer: For high availability, I would use geo-redundant architectures where services are replicated across multiple data centers or regions. A load balancer (e.g., AWS Elastic Load Balancer or NGINX) would route traffic to healthy instances. I would implement auto-scaling to ensure the system can handle fluctuating loads and leverage failover strategies to route traffic to backup instances if the primary fails.
  - Question: What happens if a particular data center or region goes down?
    Answer: If a data center or region goes down, my system would trigger a failover mechanism to route traffic to services running in other regions or data centers. This would be facilitated by global load balancers like AWS Route 53, which can detect outages and reroute traffic. Data would remain consistent through multi-region replication, and the downtime would be minimal.

---

5. Fault Tolerance

- Definition: The system’s ability to continue functioning even when some components fail.
- Discussion Points:
  - Question: How would you ensure that critical services can still operate if other services fail (e.g., using retries, circuit breakers like Resilience4j or Hystrix)?
    Answer: I would implement circuit breakers using libraries like Resilience4j or Hystrix to prevent cascading failures when dependent services fail. For example, if a downstream service becomes unresponsive, the circuit breaker would trip and prevent further requests from overloading the failing service. I would also implement retries with exponential backoff for transient failures.
  - Question: How do you plan for data loss prevention and service degradation?
    Answer: For data loss prevention, I would use regular backups and replication across multiple data centers. For service degradation, I would implement graceful degradation by reducing functionality (e.g., disabling non-essential features) when the system is under heavy load, while ensuring core services remain available.

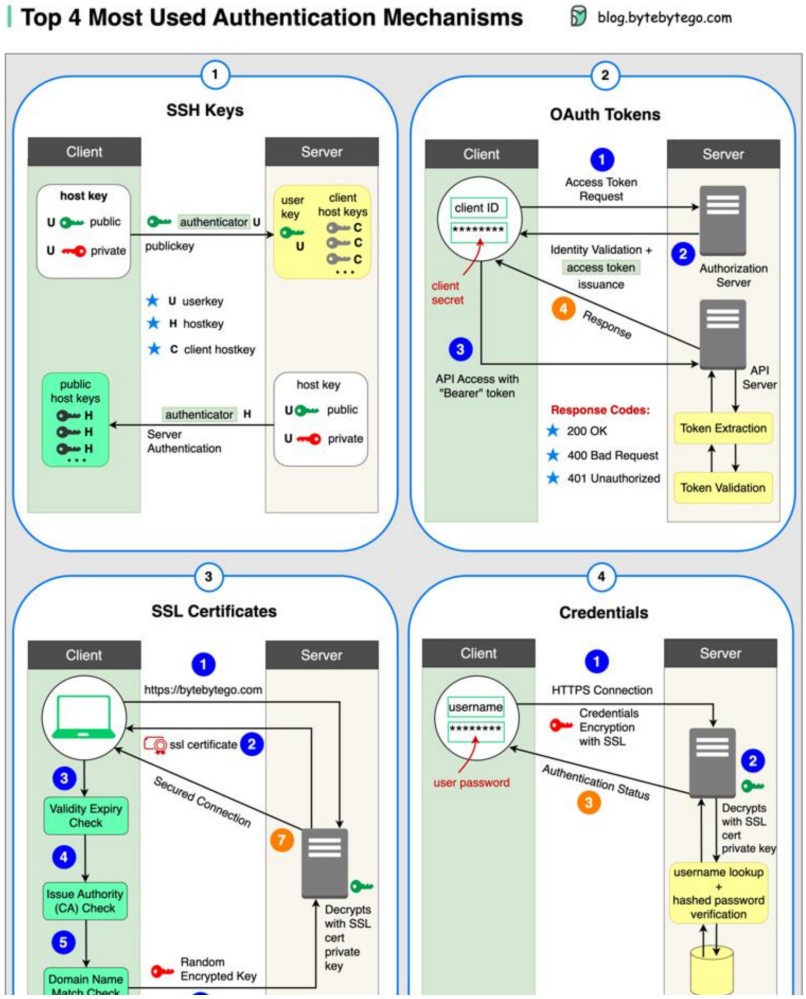
6. Consistency

- Definition: The accuracy of data across the system, ensuring all users see the same data at the same time.
- Key Concepts:
  - Strong Consistency: Ensuring that data is always consistent after a write operation.
  - Eventual Consistency: The system may not immediately reflect all changes but will eventually become consistent.
- Discussion Points:
  - Question: What consistency model would you use (e.g., eventual consistency in a NoSQL database like Cassandra)?
    `Answer: In cases where strong consistency is not required (e.g., social media feed updates), I would use eventual consistency with databases like Cassandra. For scenarios requiring strong consistency (e.g., financial transactions), I would rely on databases with ACID properties such as PostgreSQL or MySQL in a replicated setup.`
  - Question: How would you handle consistency in distributed systems? (e.g., distributed databases like MongoDB or DynamoDB)?
    `Answer: To handle consistency in distributed systems, I would use techniques like two-phase commit for transactions that span multiple services or databases. For high-availability systems, I would implement conflict resolution strategies in eventual consistency models (e.g., last write wins or vector clocks).`

---

7. Security

- Definition: Measures to protect the system and its data from unauthorized access, attacks, or breaches.
- Key Concepts:
  - Authentication: Verifying the identity of users or services.
  - Authorization: Controlling access levels and permissions.
  - Encryption: Protecting data at rest and in transit.
- Discussion Points:
  - Question: How do you secure sensitive data (e.g., using encryption, TLS/SSL, token-based authentication like OAuth)?
    `Answer: I would secure sensitive data using: 1 Encryption at Rest: All sensitive data (such as user credentials and PII) would be encrypted using AES-256. 2 Encryption in Transit: I would enforce TLS/SSL to secure communication between microservices and between clients and services. 3 Token-based Authentication: I would use OAuth 2.0 for secure user authentication, with JWT (JSON Web Tokens) for stateless, secure authorization between services. 4 Access Control: I would implement Role-Based Access Control (RBAC) to enforce strict permissions on who can access or modify certain resources.`
  - Question: What measures would you take to prevent DDoS attacks, data breaches, and insider threats?
    `Answer: To mitigate DDoS attacks, I would implement rate limiting at the API gateway level and use CDNs with built-in DDoS protection. To prevent data breaches, I would enforce strict encryption policies and use firewalls to block malicious traffic. For insider threats, I would use auditing and monitoring tools to log and detect suspicious activities, combined with least privilege access policies.`



8. Latency

- Definition: The delay between a request and the corresponding response.
- Discussion Points:
  - Question: How would you reduce latency in a high-traffic system?
    `Answer: To reduce latency in a high-traffic system, I would implement: 1 Caching: Use in-memory caches like Redis or Memcached to store frequently accessed data, reducing the need to repeatedly query databases. 2 CDN (Content Delivery Network): Serve static content (e.g., images, CSS, JavaScript) through a CDN like Cloudflare or AWS CloudFront to deliver assets from locations geographically closer to users. 3 Database Optimization: Ensure efficient query execution by using indexes on frequently accessed tables and optimizing SQL queries. For NoSQL databases, use partitioning/sharding to distribute load. 4 Asynchronous Processing: Offload non-critical tasks (e.g., sending emails, logging) to background jobs via message queues like RabbitMQ to free up resources for time-sensitive requests.`
  - Question: Discuss caching strategies (e.g., Redis, Memcached) and database query optimization.
    `Answer: For caching strategies, I would: 1 Application-level Caching: Use Redis or Memcached to cache API responses or session data for quick access without hitting the database. 2 Database Query Caching: Use write-through caching, where data is written to both the cache and the database simultaneously, to ensure the cache stays up-to-date. Alternatively, implement read-through caching to pull data from the database only if the cache is missed. 3 Query Optimization: Optimize queries by using database indexes, reducing the number of JOINs, and avoiding SELECT * queries. Additionally, leverage read replicas to offload read-heavy workloads.`

---

9. Disaster Recovery

- Definition: Strategies to restore functionality after a catastrophic event (e.g., data center failure, major security breach).
- Discussion Points:
  - Question: What disaster recovery plans would you put in place (e.g., backup strategies, multi-region deployments)?
    `Answer: I would implement the following disaster recovery plans: 1 Multi-region Deployment: Deploy critical services and data across multiple geographic regions to ensure that even if one region experiences a failure, another region can take over with minimal downtime. 2 Automated Backups: Schedule regular automated backups of databases and critical stateful services, with data replication across multiple regions. For example, use Amazon RDS with multi-AZ deployments or Google Cloud Spanner for high availability. 3 Recovery Time Objective (RTO) and Recovery Point Objective (RPO): Establish RTO and RPO goals, ensuring that systems can be restored within acceptable time frames and with minimal data loss. 4 Failover Mechanisms: Use global load balancers and DNS failover solutions like AWS Route 53 to automatically route traffic to backup regions or instances during an outage.`
  - Question: How would you handle failover to ensure minimal downtime?
    `Answer: To handle failover, I would implement active-active or active-passive failover mechanisms across multiple regions. A global load balancer like AWS Route 53 or Google Cloud Load Balancing would detect if a region goes down and reroute traffic to the next available healthy region. The system would maintain replicated databases in each region, and synchronous data replication (for strong consistency) or asynchronous replication (for eventual consistency) would ensure minimal data loss during failover.`

Thus, by addressing these NFRs comprehensively, you show that you are not only focused on building functional systems but also on their performance, resilience, security, and efficiency, which are key to a well-designed system in real-world scenarios.

II.) NFR fulfillment @ BT : Currently we achived few of the NFR as below (PERSONAL NOTES) : 10/20/24:

1. Security :
   - Definition: Measures to protect the system and its data from unauthorized access, attacks, or breaches. Main concepts : Authentication, Authorization and Encryption.
   - Messages are signed using ECDSA algorithm. It allows entities to prove their identity and ensure data integrity without the need for a central authority. Next, signed messages can use HTTPS/TLS for encrypted communication.
   - All incoming requests have their SSL termination (https/tls) at ALB (Per Gauthier)
   - Authentication and Authorization: We're using OAuth tokens (such as JWTs) to authenticate and authorize requests.
   - Mutual TLS (mTLS) for Service-to-Service Communication: Mutual TLS (mTLS) ensures that both client and server authenticate each other via certificates, securing communication between microservices.
   - Application based Rate-limiting: Requests are also Rate-limited per user based on leaky bucket algorithm to safeguard services from DDoS.
   - Logging, Monitoring, and Auditing: Helps to detect suspicious behavior or breaches. Service Isolation and Network Security : Isolate microservices in different network zones or Virtual Private Cloud (VPC)s to minimize exposure to the public internet. Use firewalls or security groups to restrict traffic between different parts of the system.
   - Encryption (Data in Transit and at Rest) : Additionally messages between uWallet FE and bridge-wallet service are encrypted (msgs are signed using ecda signature)
   - Secrets Management: Use AWS Secrets Manager instead of hardcoding sensitive information like API keys, passwords, or tokens in your code.
   - To mitigate DDoS attacks, I would implement rate limiting at the API gateway level and use CDNs with built-in DDoS protection. To prevent data breaches, I would enforce strict encryption policies and use firewalls to block malicious traffic.
2. Performance Improvement (microservice level)
   - Definition: The responsiveness and speed of the system under load. Measured by Latency & Throughput.
   - Removal of deadlocks by sorting requests w/ pub-address. Reusing the connections, Single connection per API request (See Dan's Changes), Better exception handling.
   - Multithreading : Offloading public-address to threads and other works to aync-threads.
   - Beside microservice -level performance we below :
   - We have added Redis for overall system performance.
   - Additionally, I would optimize database queries using indexes and minimize network latency by colocating services in the same region
   - For high-throughput : user-facing requests can be processed in real-time, while background jobs (such as data processing or notifications) are offloaded to the queue. For batch operations
3. Fault Tolerence :
   - Defination : The system's ability to continue functioning even when some components fail.
   - To recover from failures in graceful manner - We don't have circuit breakers and bulkheads to limit the damage caused by service failures or unintentional overloads. INSTEAD in our eco-system. Kube manages the rescheduling of pods and the launch of nodes when individual pods fail (as reported by the readiness and health

probes). That's how we handle pod/process level failures and guarantee uptime. If an entire kube service is down, say because we turn it off, or the aws region is down, then currently there is no fallback. That traffic is lost at the alb.
    - I would also implement retries with exponential backoff for transient failures.
    - Implement graceful degradation by reducing functionality (e.g., disabling non-essential features) when the system is under heavy load, while ensuring core services remain available. Example running archiverToS3 at non-peak hours.
    - We also take periodic snapshots of Database and Redis via AWS managed services.

4. Scaling :
    - Definition: The ability of the system to handle increasing loads or to be expanded to accommodate growth.
    - Load Balancing:: Adopting Load balancer which would redirect incoming traffic on round robin fashion to internal services. Load balancers can use various algorithms, such as round-robin or least connections, to distribute traffic.. We used Ngnix for SSL termination and rate limiting and routing requests to endpoints.
    - Auto-scaling:: Services use to auto-scale during peak hours by kubernetes.
    - Increased Availability: Multiple AZ in same Region - provided more redundancy and fault tolerence.
    - Database Sharding: Well we thought on using Vetess for sharding for our mysql, but then we started with RDS - Master-slave that helped to segregate read / Write traffic. (FYI - In reality we only used master DB and archived tabled periodically to S3 for Read/Data-analysis purposes.)
    - Asynchronous Processing: Adopting asynchronous communication/processing for exchange and simplex transitions. Via watcher in exchange and simplex watcher which would wait for events and make further processing. This allows the system to quickly respond to user requests while processing intensive tasks separately.
    - Caching: Adopted Caching via Redis for faster reterival of frequent data instead of reterivals from database .
    - Containerization and Orchestration: Use containerization (e.g., Docker) and orchestration tools (e.g., Kubernetes) to deploy and manage scalable and resilient applications. Containers provide consistency across different environments and simplify the scaling process.
    - Database Optimization: Optimize database queries, use appropriate indexing, and consider database caching to improve database performance. Choose the right database architecture for your workload, whether it's relational, NoSQL, or a combination.
    - Monitoring and Optimization: Implement robust monitoring to identify performance bottlenecks and areas for improvement.

5. Availibility :
    - For high availability, I would use geo-redundant architectures where services are replicated across multiple data centers or regions (Note: Currently we are in ony one region unfortunately). A load balancer (e.g., AWS Elastic Load Balancer or NGINX) would route traffic to healthy instances. I would implement auto-scaling to ensure the system can handle fluctuating loads and leverage failover strategies to route traffic to backup instances if the primary fails.

FIAT ON-RAMP: We can highlight relevant Work, Scope, Impact, achivements and Challanges.

- The system is designed with a clear API structure and well-defined data models, addressing both functional and non-functional requirements. Below are the key NF considerations:

- Asynchronous Communication: The system supports asynchronous interactions using both gRPC and REST APIs. Responses are returned immediately after submission to the backend, with the backend processing independently, integrating with Binance.com and the Tron blockchain. Services are designed to be stateless, promoting scalability and ease of maintenance.

- Security: Security is a critical focus. Messages are signed using ECDSA algorithm. It allows entities to prove their identity and ensure data integrity without the need for a central authority. Next, signed messages can use HTTPS/TLS (Specially bridge-wallet REST server in Universal Wallet) or WSS (websocket server in Decentralized Bidding Project) for encrypted communication. HTTPS/TLS termination occurs at the Application Load Balancer (ALB), ensuring encrypted message exchanges. Mutual TLS (mTLS) with certificate-based authentication is implemented for secure inter-service communication (bridge-wallet to exchange/ledger). Additional layers of security include rate-limiting (leaky bucket algorithm), IP whitelisting, and a fraud detection model to identify suspicious or outlier transactions.

- Scaling: Auto-scaling is achieved via Kubernetes, especially for stateless services like the ledger and exchange, which handle fluctuating workloads during peak hours. Traffic is routed through Load Balancer (Route 53). Master-slave architecture is used for databases, with write operations directed to the master database. Stateless serivces (gRPC ensures that interactions between services are non-blocking, efficient, and scalable)

- Monitoring and Alerting: The system is continuously monitored using Prometheus and Grafana, with integrated Slack alerts for real-time notifications. In case of service failure, Slack alerts are triggered immediately. Kubernetes handles automatic rescheduling of pods and launches new nodes if necessary, ensuring service uptime through readiness and health probes.

- Performance: Recent transaction hashes are stored in Redis, using payment IDs as keys. Write-through caching policies are applied, with session IDs set to a 6-hour TTL, transaction data to 24 hours, and other long-term data to 1 year TTL.

- Data Consistency: To maintain consistency, a payment ID (in Fiat-onRamp project), Withdrawal_hash/Deposit_hash (universal wallet project) is used as an idempotency key at the application level. A reconciler job ensures proper transaction processing. MySQL is used at the database level, providing ACID properties for consistent data handling.

- Fault Tolerance: A backoff counter is implemented for transient failure handling, with reconciling threads in the wallet service retrying failed transactions. Readiness and health probes are used to monitor service health and uptime.

- Maintainability: A CI/CD pipeline is configured between GitHub and the build system, allowing for automatic build triggers, improving the development workflow.

- Integration: The wallet service (ds_wallet) is integrated with multiple backend services (gRPC server for the ledger, REST server, Horus Gateway, reconciler) using Protocol Buffers (protobuf) and REST APIs.

- Testing: Comprehensive testing includes unit tests, API tests, and ensuring code coverage, helping to maintain a robust codebase.

- Future Improvements: Consider transitioning to webhooks for transaction status updates to eliminate the need for polling, improving system efficiency.

- Challenges: One significant challenge was delivering tokens to users' off-chain balances, as Binance lacks direct access to the internal ledger. This required developing an exchange-service to transfer tokens from the hardware wallet to users' accounts.

- Impact: The system has streamlined in-app purchases for 20 million users, resulting in a trading volume increase to 100K transactions per month. Binance has also seen significant user-base growth as a result.

Same NF consideration for other projects as well :

1. Bittorrent Speed - Onboarding + Bidding via ds_wallet <-->ledger/ Airdrop. Blockchain transaction (Deposit/Withdrawal) via exchange/watcher.
2. Fiat On-Ramp - Done above
3. Universal Wallet - Same points as above. Golang REST server implementing exchange.
4. DeCentralised Bidding - Use of auth-server (REST server) for JWT-based authentication, Bid-server (websocket server) for bidding messages.
5. LLM based project for assisting CS team. Improving customer response time, RCA and reducing dependecy on engineering
6. Data-pipeline for DAU/MAU. MR scripts. Upcoming Kafka-flink upgrades for increasing bench-pings.
7. Monotoring, logging and Alerting System using Prometheus and Grafana.
8. Mailgun Integration
9. Migration from Postgres to mySQL for Sharding via Vetess tool. Application changes via aiomysql library.
10. Integration w/ ML models for Fraud detections.

11. Share with Lite - Referral System and Reward processing system. Schema design.

FYI: MOST Important/frequent Interview Questions Asked :
++

Refer 1st section here : https://leetcode.com/discuss/study-guide/5935001/Leadership-Principles-(SRE)

III.) QUESTIONS BASED ON MY PROJECTS : 10/20/24: [**MUST READ**]
++

Refer Section-3 at https://leetcode.com/discuss/study-guide/5935001/Leadership-Principles-(SRE)

IV.) TRADE-OFFS: 10/20/24
++++++++++++++++++++

1. Security Trade-offs:

Choice: Use of ECDSA for signing messages and mTLS for service-to-service communication.

- Trade-off: While ECDSA ensures a high level of security for identity and data integrity, it comes with computational overhead, especially when verifying signatures at scale. mTLS provides enhanced security but requires certificate management, which can introduce operational complexity.
- Decision Rationale: You opted for the additional computational cost to ensure data integrity and secure internal communications in a distributed microservices architecture, thus minimizing attack vectors.

Choice: Use of OAuth with JWTs for authentication.

- Trade-off: JWT tokens are stateless, which makes scaling easier, but they can grow large, impacting performance when embedded in every request. The inability to revoke tokens easily could be a security risk in case of token theft.
- Decision Rationale: You chose JWTs because their stateless nature fits well with microservices, and the trade-off of slightly larger payloads is acceptable compared to the benefits of scalability and ease of use.

Choice: Rate-limiting (token bucket) and DDoS protection via CDN.

- Trade-off: Rate limiting ensures protection from abuse, but it can potentially block legitimate users during traffic spikes or false positive DDoS detection. [import "golang.org/x/time/rate" - Provides Token bucket based Rate-limiting]
- Decision Rationale: This strategy was chosen to protect system resources from being overwhelmed, prioritizing system availability over the risk of blocking a few legitimate users.

2. Performance Optimization Trade-offs:

Choice: Redis caching for frequently accessed data.

- Trade-off: Redis reduces database load and improves read speeds but introduces eventual consistency issues where the cached data may not immediately reflect changes in the database.
- Decision Rationale: The trade-off was justified because high throughput and low-latency responses were crucial for user experience, and eventual consistency was acceptable for most use cases (e.g., transaction history or user session data).

Choice: Asynchronous processing for resource-intensive tasks.

- Trade-off: Async processing ensures real-time user responsiveness but can delay background tasks, introducing complexity in tracking task completion or errors.
- Decision Rationale: The decision to prioritize user-facing performance over task completion time was key for enhancing user experience, ensuring quick responses while processing data in the background.

Choice: Database optimization with RDS Master-Slave architecture.

- Trade-off: This improves read scalability but introduces the need for handling **replication lag** and complexities of maintaining **synchronization** between master and slave nodes.
- Decision Rationale: This architecture was chosen because the read-heavy nature of the system benefited from offloading reads to slave nodes, improving performance without overwhelming the master database.

3. Fault Tolerance Trade-offs:

Choice: Kubernetes for auto-rescheduling of pods.

- Trade-off: Relying on Kubernetes for pod recovery and rescheduling ensures fault tolerance at the infrastructure level but could lead to short-term service disruption if an entire service or region goes down.
- Decision Rationale: While there's no complete fallback in the case of a region-wide outage, Kubernetes provided sufficient resilience at the microservice level, balancing fault tolerance with the complexity of managing cross-region architectures.

Choice: Retries with exponential backoff for transient failures.

- Trade-off: Retries can increase resource consumption and slow down response times if not handled properly. In extreme cases, too many retries could exacerbate system load during failures.
- Decision Rationale: You implemented this strategy because handling transient failures gracefully with retries was essential to prevent sporadic network issues or external service unavailability from causing cascading failures.

4. Scaling Decisions and Trade-offs:

Choice: Auto-scaling using Kubernetes.

- Trade-off: Auto-scaling ensures the system can handle traffic surges, but scaling too quickly can introduce overhead in terms of resource consumption and deployment times. There's also a delay in responding to sudden spikes.
- Decision Rationale: The trade-off between the slight lag in scaling and the need for real-time responsiveness was acceptable because the system required flexibility to handle unpredictable workloads during peak hours.

Choice: Load balancing with round-robin.

- Trade-off: Round-robin distributes traffic evenly but may not account for varying service loads (e.g., different services may process requests at different speeds), potentially leading to inefficient resource utilization.
- Decision Rationale: Round-robin was chosen because the system had well-defined microservice endpoints, and traffic patterns were relatively consistent, meaning that sophisticated load-balancing algorithms weren't necessary at the initial stages.

Choice: Database sharding versus RDS Master-Slave architecture.

- Trade-off: Sharding would have offered better scalability for **write-heavy loads** but would introduce significant complexity in terms of database management and query optimization. The master-slave architecture was simpler but not as scalable for write operations.
- Decision Rationale: You chose the master-slave approach initially, considering that **read-heavy workloads were the priority,** and you preferred the simplicity and cost-effectiveness it provided at that time. Sharding could be introduced later as the system scales further.

V.) PROS & CONS OF MICROSERVICES (vs Monolithic Architecture) : 10/22/24
++

Pros of Microservices

1. Scalability
   Pro: Each microservice can be scaled independently based on demand. This allows for more efficient resource utilization, as only the necessary components need to be scaled, reducing infrastructure costs and improving performance for critical services.
   Example: A high-traffic login service can be scaled up independently without having to scale the entire application.

2. Independent Development and Deployment
   Pro: Microservices enable teams to work on different services independently, allowing faster development cycles and more frequent deployments. This supports DevOps practices like Continuous Integration/Continuous Deployment (CI/CD).
   Example: A team working on the payment service can deploy updates without affecting the user profile or order services.

3. Fault Isolation
   Pro: Failure in one service is isolated from others, preventing the entire system from crashing. This isolation improves the resilience and fault tolerance of the overall application.
   Example: If the recommendation service goes down, it won't affect the core functionality of the product checkout service.

4. Technology Flexibility
   Pro: Different microservices can use different programming languages, frameworks, and databases based on their specific needs, allowing teams to choose the best tool for each task (also known as polyglot persistence).
   Example: A real-time analytics service can use Python and a NoSQL database, while the main application can use Java and a relational database.

5. Easier to Understand and Maintain
   Pro: Microservices are smaller and focused on a single responsibility, making them easier to understand, maintain, and debug. This modular approach reduces complexity compared to a monolithic architecture.
   Example: Understanding the behavior of a microservice that only handles user authentication is simpler than understanding a monolith handling user authentication, order management, and payments together.

6. Improved Security
   Pro: Security can be applied more granularly to each microservice. Sensitive services (e.g., payment processing) can be secured independently with stricter policies.
   Example: The authentication service can use stronger encryption and authorization mechanisms without affecting the entire system.

7. Faster Time to Market
   Pro: Teams working on independent microservices can deliver features more rapidly since they can develop, test, and deploy each service separately without waiting for the entire system to be ready.
   Example: Launching a new feature in the billing service doesn't require the approval of changes in the product catalog or user profile services.

Cons of Microservices

1. Increased Complexity
   Con: While each service may be simple, the overall system architecture becomes more complex as the number of microservices grows. Managing the interactions between many services (including dependencies, data consistency, and communication) can be challenging.
   Example: Orchestrating communication between dozens or even hundreds of microservices, managing distributed transactions, and maintaining consistent data become complex tasks.

2. **Operational Overhead**
   Con: Microservices require extensive infrastructure, monitoring, and management. Managing multiple services, including deployment, scaling, and troubleshooting, requires sophisticated tooling and operational expertise.
   Example: Maintaining **separate CI/CD pipelines, monitoring, and logging for each microservice** increases the complexity of operations.

3. Distributed Systems Challenges
   Con: Microservices rely on network communication, which introduces **latency and the possibility of network failures**. Handling issues like service discovery, network timeouts, retries, and load balancing requires careful design.
   Example: A slow or unreliable network connection can degrade the performance of a microservices-based system, causing communication issues between services.

4. **Data Consistency**
   Con: Achieving data consistency across distributed microservices is difficult. Since each microservice manages its own database, **coordinating updates and ensuring consistency** across services becomes complex, especially in transactional workflows.
   Example: In an e-commerce system, updating the inventory in one service and processing a payment in another service may lead to inconsistencies if not properly managed.

5. Cross-Cutting Concerns
   Con: Implementing cross-cutting concerns like logging, monitoring, security, and error handling across all services can be challenging. It often requires building or integrating shared libraries and tools.
   Example: Applying a unified logging mechanism across 20+ microservices requires additional coordination and infrastructure.

6. Testing Complexity
   Con: Testing a microservices architecture is more complicated than testing a monolithic application because you need to test not only individual services but also their interactions. Integration and end-to-end testing require coordination across multiple services.
   Example: A change in one service might break functionality in another, making it harder to create comprehensive tests that cover all possible interactions between services.

7. **Inter-Service Communication Overhead**
   Con: Microservices communicate over the network using protocols such as HTTP/REST, gRPC, or message queues. This introduces **overhead due to serialization, deserialization, and network latency compared to direct in-process communication in monolithic systems.**
   Example: A call from the user service to the order service may take longer due to network overhead, even if both services are fast individually.

8. Deployment and Versioning Issues
   Con: Managing deployments of multiple microservices becomes complex, especially when dealing with interdependent services. Keeping track of service versions and backward compatibility is essential but challenging.
   Example: If the user service requires changes that break compatibility with the order service, both services need to be deployed in sync, complicating the deployment process.

9. Latency and Performance Overhead

Con: Since microservices often communicate over the network, the round-trip time (RTT) between services can introduce latency. Additionally, **serialization/deserialization and API calls can reduce overall performance**.
Example: In a monolith, a function call is quick, whereas in microservices, an API call between two services might incur additional latency.

Summary

- Pros:
  - Scalability
  - Independent Development and Deployment
  - Fault Isolation
  - Technology Flexibility
  - Easier to Maintain and Debug
  - Improved Security
  - Faster Time to Market
- Cons:
  - Increased Complexity
  - Operational Overhead
  - Distributed Systems Challenges
  - Data Consistency Issues
  - Cross-Cutting Concerns Management
  - Testing Complexity
  - Inter-Service Communication Overhead
  - Deployment and Versioning Challenges
  - Latency and Performance Issues

VI.) CHALLANGES IN DISTRIBUTED SYSTEM DESIGN: 10/30/24
++

In a distributed systems design interview, discussing the challenges shows a solid understanding of how these systems operate and the complexities involved. Here are some critical challenges and considerations to discuss:

1. Network Reliability and Latency:
   Network issues (like packet loss, latency, or timeouts) can disrupt communication between nodes. Discuss the impact of unreliable networks and the importance of designing for eventual consistency, retries, and handling split-brain scenarios.
2. Data Consistency:
   Ensuring data consistency across distributed nodes is challenging, especially with systems prioritizing availability and partition tolerance (as per the CAP theorem). You could discuss strategies like quorum-based replication, eventual consistency, and multi-version concurrency control (MVCC).
3. Scalability and Load Balancing:
   Achieving horizontal scalability while ensuring the system can handle increased loads effectively. Load balancers, sharding, and partitioning strategies help, but each introduces complexity in handling failovers and rebalancing.
4. Fault Tolerance and Resilience:
   Handling node failures gracefully without disrupting overall system functionality is critical. Techniques like replication, failover mechanisms, and using distributed consensus algorithms (e.g., Raft or Paxos) can improve fault tolerance.
5. Concurrency and Synchronization:
   Concurrent access to shared resources across multiple nodes can lead to race conditions and data inconsistency. Techniques like distributed locking, vector clocks, or Lamport timestamps can help maintain order and consistency.
6. Security and Data Privacy:
   Distributed systems often involve data exchange across networks, exposing them to risks like man-in-the-middle attacks or unauthorized access. Strategies here include encryption, secure authentication, token-based authorization, and compartmentalizing access based on need-to-know principles.
7. Monitoring and Observability:
   Ensuring visibility into system health is complex due to the distributed nature of components. Discuss using tools for centralized logging, tracing (e.g., OpenTelemetry), and monitoring (e.g., Prometheus, Grafana) to detect and troubleshoot issues in real-time.
8. Data Sharding and Partitioning:
   Sharding and partitioning data effectively across nodes requires careful planning. This includes deciding partition keys, handling rebalancing when scaling, and managing the increased complexity in querying and maintaining data consistency across partitions.
9. Distributed Consensus and Coordination:
   Coordination between nodes is essential, especially for tasks like leader election or maintaining a consistent state. Discuss consensus algorithms like Paxos or Raft, used in systems like Zookeeper and etcd, and their challenges in handling network partitions and latency.
10. Testing and Deployment:
    Testing distributed systems is challenging due to unpredictable failures and inter-node communication issues. It's helpful to discuss strategies like chaos engineering, integration tests with simulated network failures, and canary deployments to validate changes gradually.
11. Cost Management:
    Operating and scaling distributed systems can become expensive, especially with cloud infrastructure. Optimizing resource usage, data transfer costs, and storage while maintaining performance can help manage expenses.

Each of these challenges can open the door to discussing specific solutions and trade-offs, which would demonstrate a well-rounded understanding of distributed systems design.

VII.) Best Practices for designing Microservices:
++++++++++++++++++++++++++++++++++++

Designing microservices-based services requires careful planning to ensure scalability, reliability, and maintainability. Here are some best practices that can guide you in developing robust microservices architectures:

1. Service Decomposition

- Single Responsibility Principle: Each microservice should focus on a single business capability or domain (e.g., payment service, notification service). This ensures that the service is easier to maintain and scale.
- Bounded Contexts: Define clear boundaries for each service, ensuring they manage their own data and logic without relying on other services' internal implementations

Bounded Contexts: Define clear boundaries for each service, ensuring they manage their own data and logic without relying on other services' internal implementations.

2. API Design

- Use REST/GraphQL for Communication: Microservices should communicate via well-defined APIs, usually over HTTP/HTTPS using REST, GraphQL, or gRPC for high-performance needs.
- Versioning: Always version your APIs to avoid breaking changes affecting consumers.
- Consistency: Ensure consistent naming conventions and structure across all APIs to make them predictable and easier to use.

3. Data Management

- Decentralized Data Management: Each microservice should own its database to avoid tight coupling. Services should communicate with each other using APIs, not through shared databases.
- Eventual Consistency: In distributed systems, ensure eventual consistency, especially when data is spread across multiple services. Use techniques like distributed transactions or the Saga pattern when needed.

4. Service Communication

- Synchronous vs. Asynchronous Communication:
- Use synchronous communication (e.g., HTTP REST, gRPC) when real-time responses are required.
- Use asynchronous messaging (e.g., RabbitMQ, Kafka) for decoupling services, handling spikes in traffic, and improving fault tolerance.
- Service Discovery: Use service discovery tools (e.g., Consul, Eureka) to dynamically locate services at runtime rather than hard-coding service addresses.

5. Security

- Authentication and Authorization: Secure APIs using OAuth2 or JWT tokens. Ensure that each service is authenticated, and only authorized users can access them.
- Encrypt Communication: Use TLS to encrypt all communications between services, ensuring data integrity and confidentiality.
- API Gateways: Use an API gateway (e.g., Kong, NGINX, Istio) to centralize access control, rate-limiting, authentication, logging, and other cross-cutting concerns.

6. Resilience and Fault Tolerance

- Circuit Breakers: Use circuit breaker patterns (e.g., Hystrix) to stop cascading failures in case one service goes down or becomes unresponsive.
- Retries and Timeouts: Set appropriate retry mechanisms with backoff strategies and timeouts to prevent waiting indefinitely for responses from a failed service.
- Fallback Mechanisms: Define fallback strategies (e.g., default responses) when a dependent service is unavailable.
- Health Checks: Implement health checks to monitor service status and integrate them into orchestrators like Kubernetes for automatic service restarts and scaling.

7. Observability

- Logging: Implement centralized, structured logging (e.g., ELK Stack, Fluentd) across services for easy debugging.
- Tracing: Use distributed tracing (e.g., Jaeger, Zipkin) to trace requests across multiple services and understand how a request flows through the system.
- Metrics and Monitoring: Monitor microservices using metrics (e.g., Prometheus, Grafana) to track performance, request rates, latency, and errors.
- Alerts: Set up alerts based on metrics and logs to be notified of anomalies or failures (e.g., via Prometheus Alertmanager or PagerDuty).

8. Scalability

- Auto-scaling: Use container orchestration tools like Kubernetes or Docker Swarm to automatically scale services based on demand.
- Stateless Services: Design services to be stateless where possible, offloading session data to distributed stores (e.g., Redis) to improve scalability.
- Load Balancing: Use load balancers to distribute requests across instances of microservices for horizontal scaling.

9. Database Design

- Database per Service: Each service should own and manage its own database (polyglot persistence). This decouples services and reduces contention on shared databases.
- Event Sourcing and CQRS: Use event sourcing or CQRS (Command Query Responsibility Segregation) patterns where necessary to manage complex domain data changes and queries

10. DevOps Practices

- CI/CD Pipelines: Implement continuous integration and continuous delivery (CI/CD) pipelines to automate testing and deployment of microservices (e.g., Jenkins, CircleCI).
- Containerization: Package each service in containers (e.g., Docker) to ensure consistency across environments.
- Infrastructure as Code: Use infrastructure-as-code (IaC) tools (e.g., Terraform, Ansible) to automate the provisioning and management of infrastructure.

11. Service Contracts

- Consumer-Driven Contracts: Use tools like Pact to manage and verify service contracts between different microservices, ensuring that changes in one service do not break others.
- Backward Compatibility: Ensure new versions of services maintain backward compatibility to allow for incremental deployment and gradual migration.

12. Testing Strategies

- Unit Testing: Each microservice should have comprehensive unit tests for its internal logic.
- Integration Testing: Test interactions between services using real or mocked dependencies.
- Contract Testing: Ensure API contracts between services remain valid over time.
- Chaos Testing: Use chaos engineering (e.g., Chaos Monkey) to test the system's resilience under unexpected failures.

13. Deployment Strategies

- Blue-Green Deployments: Minimize downtime by deploying new versions of a service to a separate environment and switch traffic over only after validation.
- Canary Releases: Gradually release updates to a small percentage of users, allowing for a staged rollout and monitoring before a full-scale release.
- Rolling Deployments: Slowly replace instances of a service one by one with the new version to minimize downtime.

14. Event-Driven Architecture

- Publish/Subscribe: Use event-driven patterns where services emit events when significant actions occur (e.g., order placed), and other services subscribe to relevant events to react.
- Event Streaming: Use event streams (e.g., Kafka) to facilitate high-throughput, low-latency communication between services that need real-time updates.

15. Data Handling in Distributed Systems

- Idempotency: Ensure that service operations are idempotent to avoid duplicate actions in case of retries.
- Data Partitioning: Use horizontal partitioning/sharding strategies for databases to distribute load across multiple servers.

16. Security and Governance

- Security Audits: Regularly perform security audits to identify vulnerabilities in the microservices ecosystem.
- Data Privacy: Ensure compliance with data privacy regulations (e.g., GDPR, HIPAA) across services, especially when handling sensitive data.
- API Quotas and Rate Limiting: Implement quotas and rate limiting to prevent abusive usage of APIs.

By following these best practices, you can build microservices that are scalable, resilient, secure, and easy to manage. This ensures that your system can handle growing demands and complex business needs while maintaining high availability and performance.

VIII.) SCALING THE STATEFUL SERVICES: 10/22/24
++++++++++++++++++++++++++++++++++++++

Scaling stateful services presents unique challenges compared to scaling stateless services because the service instances need to maintain and manage their state across multiple requests or sessions. However, there are several strategies and best practices for scaling stateful services effectively. Below are the common approaches:

1. **Externalizing State** (Move State Out of the Service)

- Description: The most common way to scale stateful services is to externalize the state by moving it to an external system (like a database, object storage, or distributed caching system) while keeping the service itself stateless.
- How it works: The service no longer holds the state internally, allowing multiple instances of the service to work independently and share access to the external state. Databases, key-value stores (e.g., Redis, Memcached), or persistent storage (e.g., S3) can be used to store session or application data.
- Benefits: This approach simplifies scaling, as the service becomes effectively stateless and can scale horizontally.
- Tools:
    - Relational databases (MySQL, PostgreSQL)
    - NoSQL databases (Cassandra, MongoDB)
    - Distributed caches (Redis, Memcached)
    - Object storage (AWS S3)
- Example: Store user session data in Redis or a database.
  The service fetches the session data from Redis when required, making the service stateless but still able to handle sessions.

2. **Sticky Sessions** (Session Affinity)

- Description: In this approach, a load balancer routes requests from a specific user/session to the same service instance. This ensures that all state for a given user or session remains on the same instance.
- How it works: The load balancer uses a hashing mechanism based on the client's session identifier (cookie or token) to route all requests for that session to the same service instance. This maintains consistency of the state within that instance.
- Challenges: While this allows you to maintain the state within the service, it can lead to uneven load distribution, as some instances may handle more sessions than others.
- Tools:
    - AWS Elastic Load Balancer (ELB) with sticky sessions
    - Nginx/HAProxy with session affinity
- Example: A shopping cart stored in memory on a service instance remains accessible as long as the load balancer directs all user requests to the same instance.

3. **State Replication**
   Description: Replicate the state across multiple service instances. Each instance maintains a copy of the state so that any instance can handle any request.
   How it works: When the state changes, those changes are propagated to all the replicas. This approach allows each instance to handle requests without needing to go to an external state store.
   Challenges: It's crucial to ensure data consistency across replicas, especially under high loads. Replication can introduce latency if the system waits for consistency.
   Tools:
   Distributed systems with replication (e.g., Apache Zookeeper, Consul for service coordination)
   Master-slave replication in databases
   Example:A distributed system that replicates user data across multiple nodes so that any node can handle the user's requests.

4. **Stateful Microservices with Consistent Hashing**

- Description: Use consistent hashing to ensure that a request for a particular state is always routed to the same instance.
- How it works: Consistent hashing ensures that data is distributed across nodes efficiently. This approach minimizes disruption when nodes are added or removed and keeps state with the appropriate instance.
- Tools:
    - Akka Cluster for distributed systems with consistent hashing
    - Hashing algorithms for load balancers (Ring Hash, etc.)
- Example: In a stateful microservices architecture, user sessions or data objects can be consistently routed to the same instance using consistent hashing.

5. Sharding (Partitioning State)

- Description: Split the state between multiple instances by sharding the state across different nodes, where each shard is responsible for a subset of the data.
- How it works: Requests are routed to the correct shard based on some consistent hashing mechanism. For example, users could be partitioned by user ID or session ID.
- Benefits: Distributes state across instances, allowing for parallelism and better scaling.
- Challenges: Sharding introduces complexity in managing the shards and handling failures. It's also essential to manage the balance of shards dynamically.
- Tools:
    - Apache Kafka for state sharding in messaging systems
    - Distributed databases (Cassandra, MongoDB) with built-in sharding
- Example: Split user sessions across multiple instances based on user ID, where users with ID 0-999 go to instance 1, and users with ID 1000-1999 go to instance 2.

6. Stateful Containers with Orchestration (Kubernetes StatefulSets)

- Description: Use container orchestration platforms like Kubernetes to manage stateful services. StatefulSets in Kubernetes are specifically designed to manage stateful applications, where each instance has a persistent identifier and storage.
- How it works: Kubernetes manages individual instances (pods) of your service, ensuring they maintain their identity and attach to persistent storage. This allows horizontal scaling while maintaining state.
- Benefits: Kubernetes handles pod failure recovery, scaling, and persistent storage orchestration, which makes managing stateful workloads easier.
- Tools:
    - Kubernetes StatefulSets
    - Persistent Volumes (PV) and Persistent Volume Claims (PVC) in Kubernetes
- Example: Deploy a stateful service like a database using Kubernetes StatefulSets, where each replica gets a persistent storage volume that it can use to maintain state.

7. Event Sourcing & CQRS (Command Query Responsibility Segregation)

- Description: Event sourcing involves storing changes to the application state as a sequence of events. CQRS separates the responsibility of updating (command) and querying (read) the state.
- How it works: State changes are stored as events in an event store. When a service needs to update the state, it generates an event. The service can be scaled by replaying these events to reconstruct the state when needed.
- Benefits: This allows easier scaling, as you can distribute the processing of events across services and rebuild the state dynamically.
- Tools:

    - Event stores (Kafka, EventStore)
    - CQRS frameworks for managing read and write operations
- Example: A financial application stores all transactions as events. The system can replay these events to reconstruct the current account balance when needed.

Key Considerations When Scaling Stateful Services:

1. Consistency: Ensure the data is consistent across replicas, especially in distributed environments.
2. Availability: Use replication and fault tolerance techniques to avoid data loss in case of instance failures.
3. Partitioning/Sharding: Ensure effective partitioning to distribute load evenly across instances.
4. Failover and Recovery: Implement robust failover and recovery mechanisms, such as data replication, to avoid data loss and ensure availability.
5. Data Storage and Latency: Depending on the storage solution (e.g., external databases or caches), latency can become a concern. Use in-memory caching or optimized storage solutions.

By following these strategies, you can scale stateful services effectively while maintaining performance and consistency.

IX.) Scale communication between two microservices (grpc based) : 10/20/24:
++

Scaling communication between two gRPC-based microservices involves addressing several aspects, such as network efficiency, load balancing, failure handling, and resource optimization. Here's a detailed guide to scaling this communication:

1. Load Balancing
   Load balancing ensures that requests are distributed evenly across instances of a microservice to prevent overloading any single instance.

- Client-Side Load Balancing:
  gRPC natively supports client-side load balancing. You can implement a load balancing policy at the gRPC client level to distribute requests across multiple instances of a backend microservice. Common strategies include:
    - Round-robin: Each request is routed to the next available instance in a circular fashion.
    - Least connections: Requests are routed to the instance with the fewest active connections.
    - Consistent Hashing: Ensures that requests with the same key (like user ID) are routed to the same instance, useful when caching.

Example in gRPC (with DNS or service discovery):

```
// Set up client-side load balancing with round-robin policy
conn, err := grpc.Dial("my-grpc-service:50051", grpc.WithInsecure(), grpc.WithBalancerName("round_robin"))
```

For larger-scale systems, consider integrating with service discovery systems (like Consul or Eureka) or Kubernetes' DNS-based load balancing, where the client will query the registry for available microservice instances.

- Envoy Proxy for gRPC Load Balancing:
  Instead of managing client-side load balancing, you can use Envoy as a reverse proxy that handles load balancing and service discovery. Envoy sits between the gRPC client and server, acting as a smart layer that forwards requests to available instances based on various strategies.

2. Horizontal Scaling
   When you horizontally scale your microservices by adding more instances, you need to ensure the communication channels (between gRPC clients and servers) can scale dynamically.

- Auto-scaling via Kubernetes:
  In Kubernetes, gRPC microservices can scale automatically based on CPU/memory usage or custom metrics like request latency, number of open connections, or RPC rates. Kubernetes' Horizontal Pod Autoscaler (HPA) can increase or decrease the number of pods running a microservice based on traffic demands:

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: my-grpc-service-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: my-grpc-service
  minReplicas: 2
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 70
```

As pods scale up and down, the service discovery or DNS system dynamically updates the list of available instances.

3. Circuit Breaking and Failure Handling
   When scaling, some instances of a service may experience failure or high latency. To prevent cascading failures across microservices, circuit breaking can help isolate problematic services.

- gRPC with Circuit Breakers via Envoy:
  Envoy can be used as an intermediary between gRPC services to implement circuit breakers. When a service instance starts returning errors or takes too long to respond, Envoy trips the circuit breaker and stops routing requests to that instance temporarily.

```
circuit_breakers:
  thresholds:
    max_connections: 1000
    max_pending_requests: 100
    max_requests: 500
```

This ensures that a failure in one service doesn't cause failures across the system and provides more predictable performance under load.

- Retry Logic:
  Implement retry logic in the gRPC client to handle transient errors or timeouts. gRPC has built-in retry policies, which can be configured with a backoff mechanism to retry failed requests:

```
opts := []grpc.DialOption{
  grpc.WithRetry(func() retry.Backoff { return retry.DefaultBackoff }),
}
conn, err := grpc.Dial("my-grpc-service:50051", opts...)
```

4. Streaming and Asynchronous Processing
   Scaling communication for real-time or high-throughput scenarios can be enhanced using gRPC streaming. gRPC supports three types of streams:

- Unary: Standard request/response model.
- Server-side Streaming: The server sends multiple responses for a single request.
- Client-side Streaming: The client sends a stream of requests to the server.
- Bidirectional Streaming: Both the client and server can send streams of messages.

For scenarios involving large data sets or real-time updates (e.g., streaming logs or bidirectional communication), gRPC streaming can be very efficient compared to traditional request/response models. With streaming, the connection is kept open, reducing the overhead of establishing new connections and handling batch requests.

Example of bidirectional streaming

```
stream, err := client.MyServiceMethod(ctx)
for {
  // Sending and receiving messages asynchronously
  err := stream.Send(&Request{})
  resp, err := stream.Recv()
}
```

To scale streaming services, ensure the services are deployed across multiple instances and use load balancing to distribute streams evenly.

5. Observability: Monitoring and Tracing
   When scaling microservices, it's crucial to monitor the communication between services to detect bottlenecks and optimize performance.

- Distributed Tracing:
  Implement OpenTelemetry or Jaeger for distributed tracing to track gRPC calls across microservices. This will give you insight into the latency between services, allowing you to identify bottlenecks in your service-to-service communication.

Example with OpenTelemetry:

```
tracer := otel.Tracer("grpc-client")
ctx, span := tracer.Start(context.Background(), "grpc-call")
defer span.End()

// Making the gRPC call
_, err := client.MyMethod(ctx, &Request{})
```

- Metrics and Logging:
  Use Prometheus to track gRPC-specific metrics such as:
  - Request rates
  - Latency
  - Error rates
  - Active connections

Envoy, if used as a proxy, can also expose rich gRPC metrics that you can scrape via Prometheus. Set up alerts for unusual spikes in response times or connection errors to address scaling issues proactively.

6. Data Consistency and State Management
   In large-scale systems, ensuring data consistency during high traffic can be tricky. Since gRPC is stateless by default, communication between services typically assumes eventual consistency.

Caching:
Use distributed caching (like Redis) between services for frequently accessed or computed data to reduce direct database hits. Redis can be used to cache responses from microservice A that are required by microservice B, thus reducing the load on both services.

Example of Redis for caching responses

```
cacheKey := "grpc-response-key"
cachedData, err := redisClient.Get(cacheKey).Result()
if err == redis.Nil {
  response, err := grpcClient.MyMethod(ctx, &Request{})
  redisClient.Set(cacheKey, response, time.Minute*10)
}
```

Message Queuing:
For scenarios where eventual consistency and asynchronous communication are acceptable, you can introduce message queues (e.g., Kafka) between services. This decouples services and allows for better fault tolerance and traffic management.

7. Security
   Securing gRPC communication is essential, especially at scale.

TLS:
Use TLS to encrypt traffic between gRPC services. This ensures that the communication is secure even when scaling across different environments, such as multiple data centers or cloud regions.

```
creds, err := credentials.NewClientTLSFromFile("path/to/cert.pem", "")
```

```
conn, err := grpc.Dial("my-grpc-service:50051", grpc.WithTransportCredentials(creds))
```

Mutual TLS (mTLS):
For additional security, especially in zero-trust architectures, implement mTLS. Both the client and server validate each other's certificates, ensuring mutual trust.

OAuth 2.0 or JWT:
For user authentication and authorization, implement token-based mechanisms like OAuth 2.0 or JWT (JSON Web Tokens) in gRPC. The client sends tokens with each gRPC call, and the server validates them.

Additionally:

Summary of gRPC Security Approaches:

- TLS/SSL Encryption: Ensures encrypted communication.
- Mutual TLS (mTLS): Provides authentication on both sides.
- Token-Based Authentication: Use tokens (like JWT) in metadata.
- OAuth2: OAuth2 tokens can be passed for access control.
- Interceptors: Add custom security logic to requests.
- Role-Based Access Control: Fine-grained access control based on roles.
- These methods can be combined to secure gRPC services effectively, depending on the use case.

X.) SYSTEM DESIGN QUESTIONS FOR SENIOR ROLES : 10/20/24:
++

1. Question: Describe a time when you had to improve the scalability of a system. What challenges did you face, and how did you overcome them?
   Answer: In my current role, I was tasked with improving the scalability of a microservices-based system. One of the main challenges was handling increasing traffic during peak hours without affecting the overall system performance. To address this, I introduced Kubernetes-based auto-scaling. The stateless services were set to scale based on CPU and memory usage. Traffic was distributed using a load balancer, which employed round-robin algorithms. Additionally, I optimized our database operations by introducing Redis for caching frequently accessed data, reducing the load on our primary database. We also considered sharding strategies but instead adopted a master-slave architecture that allowed us to split read and write operations, improving scalability without the immediate need for sharding. The main challenge here was ensuring minimal downtime during these changes, which we achieved through careful planning and gradual rollout.

2. Question: How do you ensure the reliability of a system that handles critical operations?
   Answer: Ensuring reliability in a system handling critical operations requires a combination of proactive monitoring, fault tolerance mechanisms, and solid architectural design. In my current project, we implemented health and readiness probes within Kubernetes to monitor the status of pods and auto-reschedule them when needed. This way, even if individual components fail, the system continues functioning with minimal downtime. We also leveraged multi-AZ (Availability Zones) deployment to ensure high availability, so even if one zone goes down, traffic can be routed to another. Additionally, we built redundancy into our architecture by taking regular snapshots of our database and using Redis backups to ensure quick recovery in case of failure. The challenge was managing these snapshots and backups without affecting system performance, which we addressed by scheduling these operations during non-peak hours.

3. Question: Can you talk about a situation where you had to improve the performance of a system under heavy load?
   Answer: One instance that comes to mind involved optimizing the performance of a high-throughput microservices-based system that was experiencing high latency during peak times. The first step I took was to remove deadlocks by sorting incoming requests and reusing connections, which significantly improved the speed of the API. 2. We also adopted asynchronous processing using a queuing system for batch operations and background jobs, ensuring that real-time requests weren't blocked by long-running processes. Redis caching was introduced to offload frequent data reads from the database, reducing latency. 3. Another key optimization was implementing multithreading for certain operations, offloading tasks like public address generation to asynchronous threads. This improved throughput without increasing the load on individual services. The main challenge was optimizing these changes while maintaining system stability, which we solved through rigorous load testing before deployment.

4. Question: How have you handled fault tolerance in a system with high reliability demands?
   Answer: Fault tolerance is a critical aspect of any high-reliability system. In my experience, we handled pod/process-level failures using Kubernetes' self-healing capabilities, where pods were automatically rescheduled based on health and readiness probes. However, if an entire service or AWS region failed, there was no fallback mechanism in place. To improve this, we considered geo-redundant architecture but had to settle for scaling across multiple Availability Zones within the same region due to budget constraints. 2. Also For transient failures, we implemented retries with exponential backoff to handle temporary issues gracefully. I also integrated graceful degradation into the system, where non-essential features were disabled during heavy loads, ensuring that critical operations remained unaffected. The challenge here was maintaining system functionality without overloading it, which was resolved by setting priority levels for different types of services.

5. Question: Can you describe a challenging situation where you had to ensure both security and performance in a system?
   Answer: One challenging scenario I faced was ensuring security while optimizing system performance for a wallet service that handled financial transactions. Security was critical due to the nature of the transactions, so we implemented mutual TLS (mTLS) for service-to-service communication and OAuth tokens for authentication. Messages were signed using the ECDSA algorithm to ensure data integrity. 2. However, these security measures introduced additional overhead, leading to performance degradation. To balance security and performance, we implemented application-based rate limiting using the leaky bucket algorithm to safeguard against DDoS attacks without affecting legitimate users. We also optimized the system by caching JWT tokens and session data in Redis, which reduced the overhead of frequent token validation. This approach improved performance while maintaining high levels of security. The challenge was fine-tuning these configurations to avoid overburdening the system while ensuring top-notch security.

6. Question: Tell me about a time when you had to make a tradeoff between performance and reliability. How did you decide what to prioritize?
   Answer: In one of our systems, we faced a situation where we had to choose between maximizing performance and maintaining system reliability, particularly during a period of high transaction volume. Initially, we had designed the system to prioritize real-time processing of transactions, but as the load increased, some services began to fail intermittently. 2. To address this, we implemented graceful degradation, where non-critical services (like notifications or analytics) were temporarily disabled during peak load, allowing us to focus resources on the core transactional services. We also introduced circuit breakers to limit cascading failures from affecting other parts of the system. 3. Although this meant sacrificing some performance on non-essential features, it ensured the reliability of our core services and prevented downtime. The challenge was to make sure the degradation was transparent to the users, which we achieved by pre-communicating service limitations during high-load periods.

7. Question: Describe a situation where you had to implement a security solution that impacted system performance. How did you mitigate the performance impact?
   Answer: One notable case was when we introduced mutual TLS (mTLS) for service-to-service communication to enhance security. While mTLS provided a high level of security by ensuring that both the client and server authenticate each other, it introduced significant overhead due to the handshake process. 2. To mitigate the performance impact, I implemented connection pooling and session reuse mechanisms for frequently communicated services. This reduced the frequency of full TLS handshakes, improving latency without compromising security. We also introduced Redis-based caching for token management and session state, which allowed us to reduce repeated token validation processes for authenticated requests.3. By carefully balancing security and performance through these optimizations, we were able to improve response

times while maintaining the strict security requirements of the system.

8. Question: Have you ever encountered a situation where scaling a system caused unexpected challenges? How did you handle it?
   Answer: Yes, one such challenge arose when we implemented auto-scaling in our Kubernetes cluster for a microservices architecture. While auto-scaling helped handle increased load, we encountered an unexpected challenge with database bottlenecks. As the number of pods increased, the volume of database connections also rose, eventually causing connection pool exhaustion. 2. To address this, I introduced connection pooling strategies and optimized database queries to ensure the system could handle the increased demand. Additionally, we adopted database read replicas to offload read-heavy operations from the master database, which helped to balance the load effectively. 3. We also put rate-limiting mechanisms in place at the API gateway level to prevent a sudden surge in requests from overwhelming the database. The challenge here was managing the database scaling without affecting user experience, which we mitigated through these architectural changes and proper monitoring.

9. Question: Can you share an example of a time when you improved system availability? What was your approach, and what challenges did you face?
   Answer: In one project, we needed to improve system availability as our services were deployed in a single region, which posed a risk of significant downtime in case of regional outages. To address this, I proposed and implemented a multi-AZ (Availability Zone) architecture, where services were replicated across multiple zones within the same region. 2. We used an Elastic Load Balancer (ELB) to distribute traffic across healthy instances, ensuring high availability even if one AZ went down. Additionally, I set up health checks and failover strategies to reroute traffic to backup instances if the primary ones failed. 3. The main challenge was ensuring consistent data replication across zones while keeping latency low. We overcame this by optimizing database writes and introducing read replicas in each AZ to handle local reads, reducing cross-AZ latency. While we couldn't fully achieve geo-redundancy due to cost constraints, this approach significantly improved availability.

10. Question: How do you handle performance monitoring and optimization in large-scale systems? Can you give an example of how you used monitoring tools to identify and resolve a performance issue?
    Answer: Performance monitoring and optimization are crucial for maintaining system health, especially at scale. In our system, we used Prometheus for monitoring and Grafana for real-time visualization of performance metrics. This setup allowed us to track key metrics like CPU usage, memory consumption, request latency, and database throughput. 2. One issue we encountered was a sudden spike in response time during peak traffic hours. Upon investigating through Grafana dashboards, we identified a bottleneck in one of the microservices responsible for database-heavy operations. The service was performing inefficient queries that caused high CPU and memory consumption. 3. To resolve this, I optimized the database queries by adding appropriate indexing and reducing the number of database calls. I also introduced Redis as a caching layer for frequently accessed data, significantly reducing the load on the database and improving overall response times. We used Prometheus alerts to ensure that similar performance issues would be flagged early in the future.

11. Question: Can you share a time when you had to address a system failure? How did you ensure quick recovery and prevent it from happening again?
    Answer: In one instance, we experienced a system-wide outage due to a failure in our Kubernetes cluster caused by a misconfiguration in the auto-scaling policy. This resulted in excessive pod rescheduling and resource exhaustion, which brought down several key services. 2. To recover quickly, we immediately rolled back the configuration changes and rescheduled the pods manually to restore service availability. I also worked closely with the DevOps team to adjust the auto-scaling parameters to prevent over-scaling. 3. Post-recovery, we conducted a thorough root cause analysis and implemented a set of safeguards, including stricter health checks, more fine-tuned scaling policies, and better resource allocation strategies. We also set up real-time Slack alerts for critical service failures to ensure a faster response to similar incidents in the future.

12. Question: How do you handle NFR (Non-Functional Requirements) in your design process, especially when it comes to security, performance, and scalability?
    Answer: Non-functional requirements such as security, performance, and scalability are always a key part of my design process. For security, I focus on incorporating strong authentication (OAuth tokens, JWTs), authorization, encryption (HTTPS, TLS, mTLS), and DDoS protection mechanisms like rate-limiting. 2. For performance, I ensure that we leverage caching (e.g., Redis), optimize database queries, and use asynchronous processing for heavy tasks. We also monitor latency and throughput to make sure that the system can handle the required load. 3. For scalability, I design systems with auto-scaling capabilities using Kubernetes, stateless services, and load balancing across services and zones. Database sharding and master-slave architectures are considered when scaling read and write operations. 4. A key challenge is balancing these NFRs without causing trade-offs that might affect other parts of the system. For example, improving security shouldn't introduce significant latency, and enhancing scalability shouldn't compromise system reliability. Monitoring tools like Prometheus and real-time alerts help in making data-driven decisions to balance these requirements effectively.

13. Question: Tell me about a time when you needed to optimize system latency. How did you identify the sources of latency, and what actions did you take to resolve them?
    Answer: We noticed increased latency in our ledger services during peak transaction hours. After profiling the service, I identified that multiple calls to the database were the primary cause. To address this, I implemented in-memory caching using Redis to store frequently accessed transaction data, significantly reducing the number of database queries. I also optimized our database schema and introduced asynchronous processing for non-critical operations, which helped us further reduce latency and improve system responsiveness.

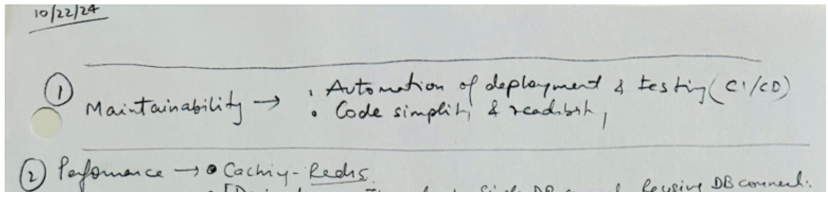
14. Question: Can you describe a time when you had to make a trade-off between system availability and consistency? How did you decide what to prioritize?
    Based on the Speed architectural diagram: The architecture includes services like the "Ledger Service" and an "Archive Service." These often require a balance between availability and consistency, especially in distributed systems. Answer: When designing the ledger service for high availability, we faced a trade-off between ensuring data consistency and maintaining availability during peak traffic. During times of heavy load, we prioritized availability by implementing eventual consistency. I introduced asynchronous processing and transaction logs that allowed us to ensure transactions were eventually processed, even if some data was temporarily out of sync. This approach helped us maintain service availability, while batch processes ensured data consistency was restored without affecting the user experience.

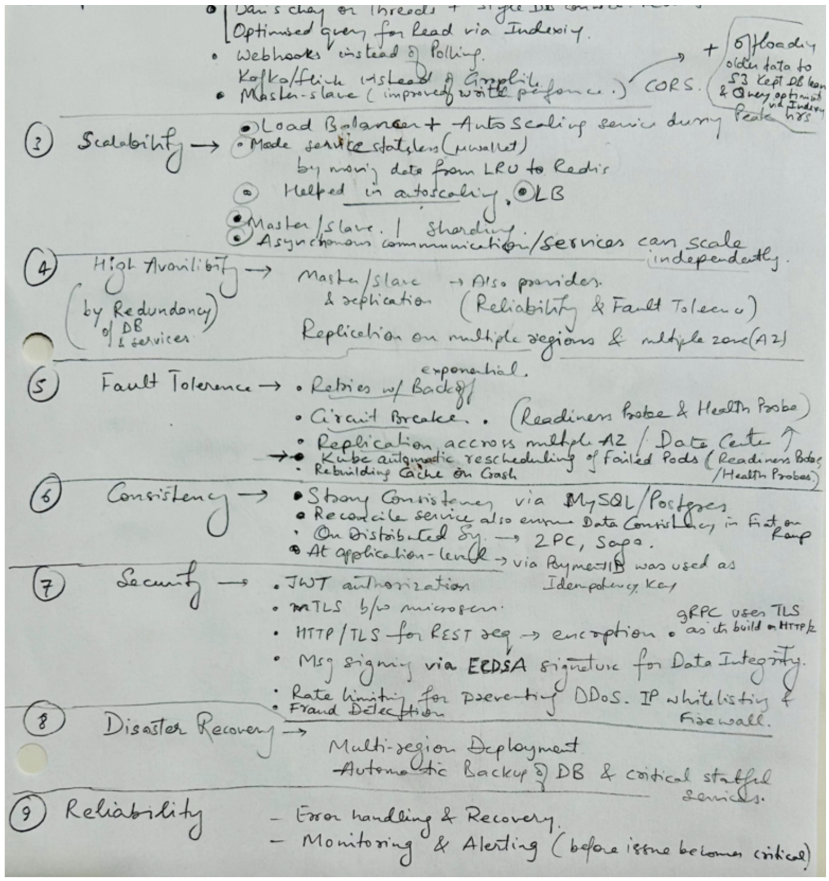
XI.) IMPLEMENTATION :
++++++++++++++++++

1. Consistent Hashing
2. Rate Limiter & LRU

ADDENDUM
++++++++++

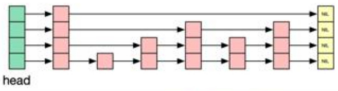
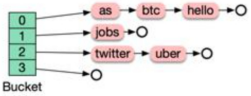
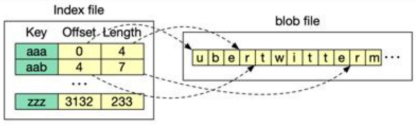
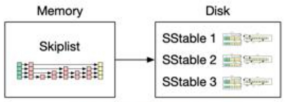
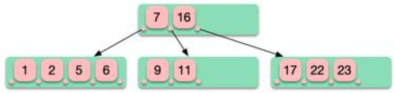
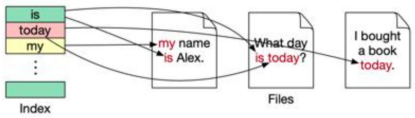
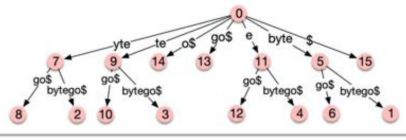
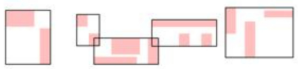
Summary of NFR @ BT

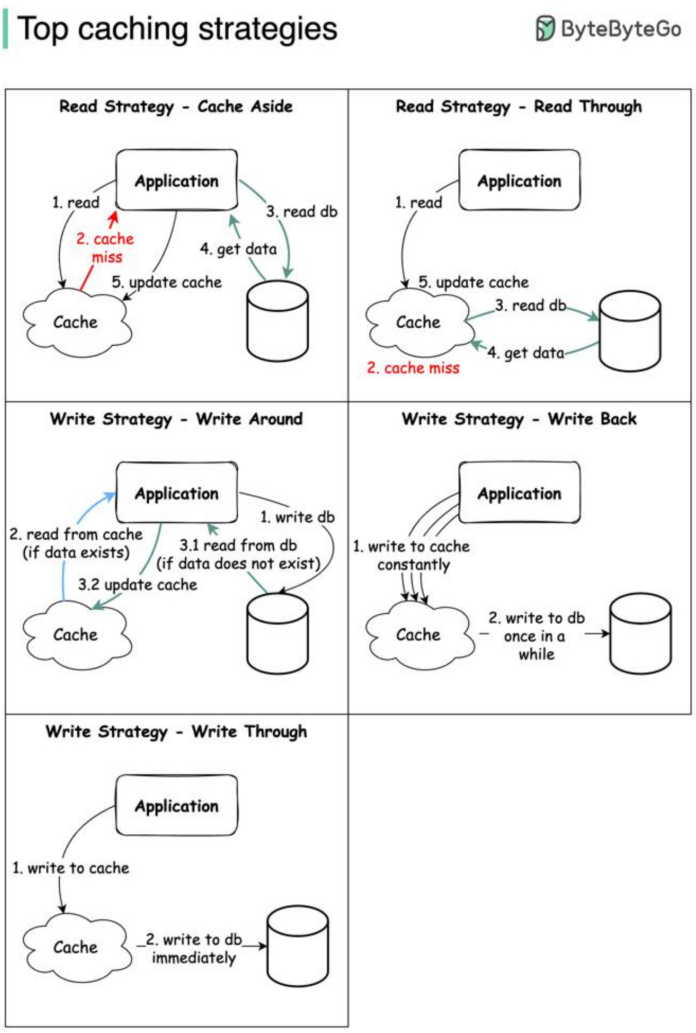




8 Data Structures That Power Your Databases

ByteByteGo.com

Types	Illustration	Use Case	Note
Skiplist		In-memory	used in Redis
Hash index		In-memory	Most common in-memory index solution
SSTable		Disk-based	Immutable data structure. Seldom used alone
LSM tree		Memory + Disk	High write throughput. Disk compaction may impact performance
B-tree		Disk-based	Most popular database index implementation
Inverted index		Search document	Used in document search engine such as Lucene
Suffix tree		Search string	Used in string search, such as string suffix match
R-tree		Search multi-dimension shape	Such as the nearest neighbor



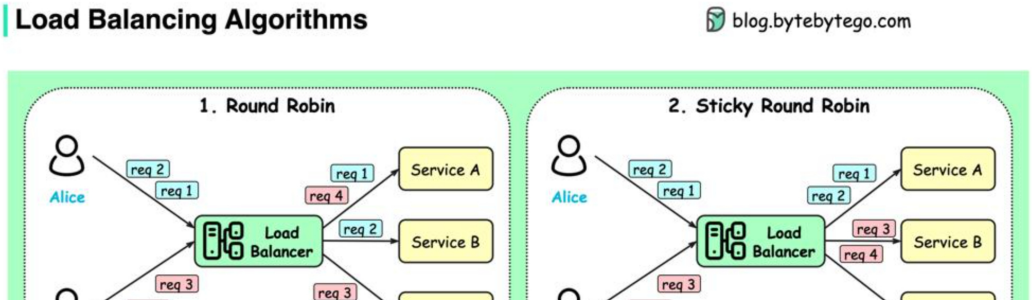
Load Balancing Strategies :

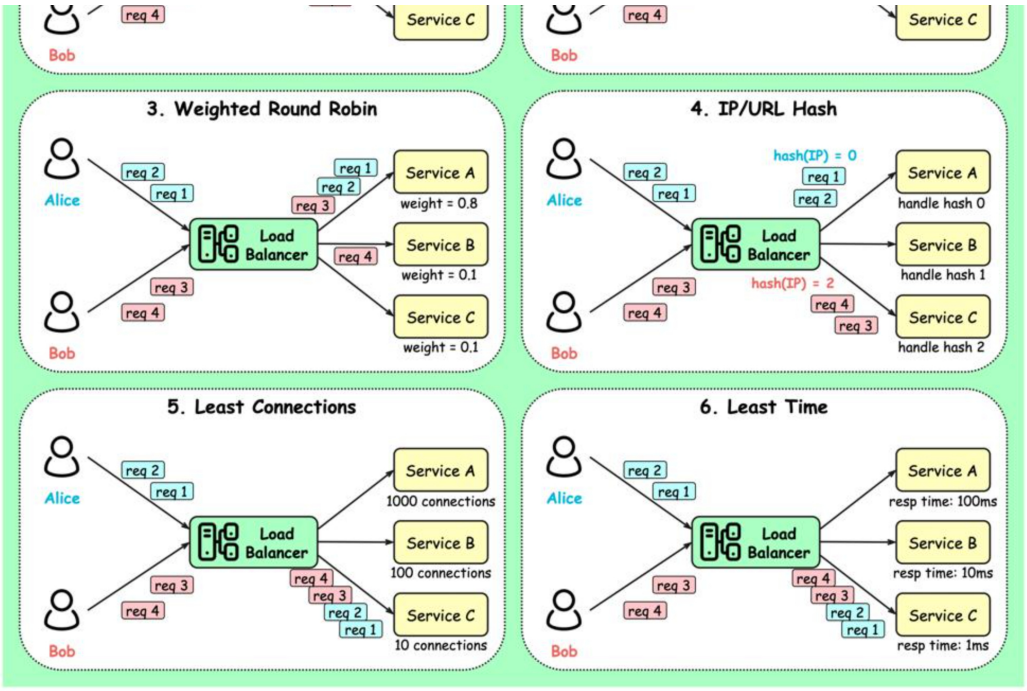
Static Algorithms

- Round robin
  - The client requests are sent to different service instances in sequential order. The services are usually required to be stateless.
- Sticky round-robin
  - This is an improvement of the round-robin algorithm. If Alice's first request goes to service A, the following requests go to service A as well.
- Weighted round-robin
  - The admin can specify the weight for each service. The ones with a higher weight handle more requests than others.
- Hash
  - This algorithm applies a hash function on the incoming requests' IP or URL. The requests are routed to relevant instances based on the hash function result.

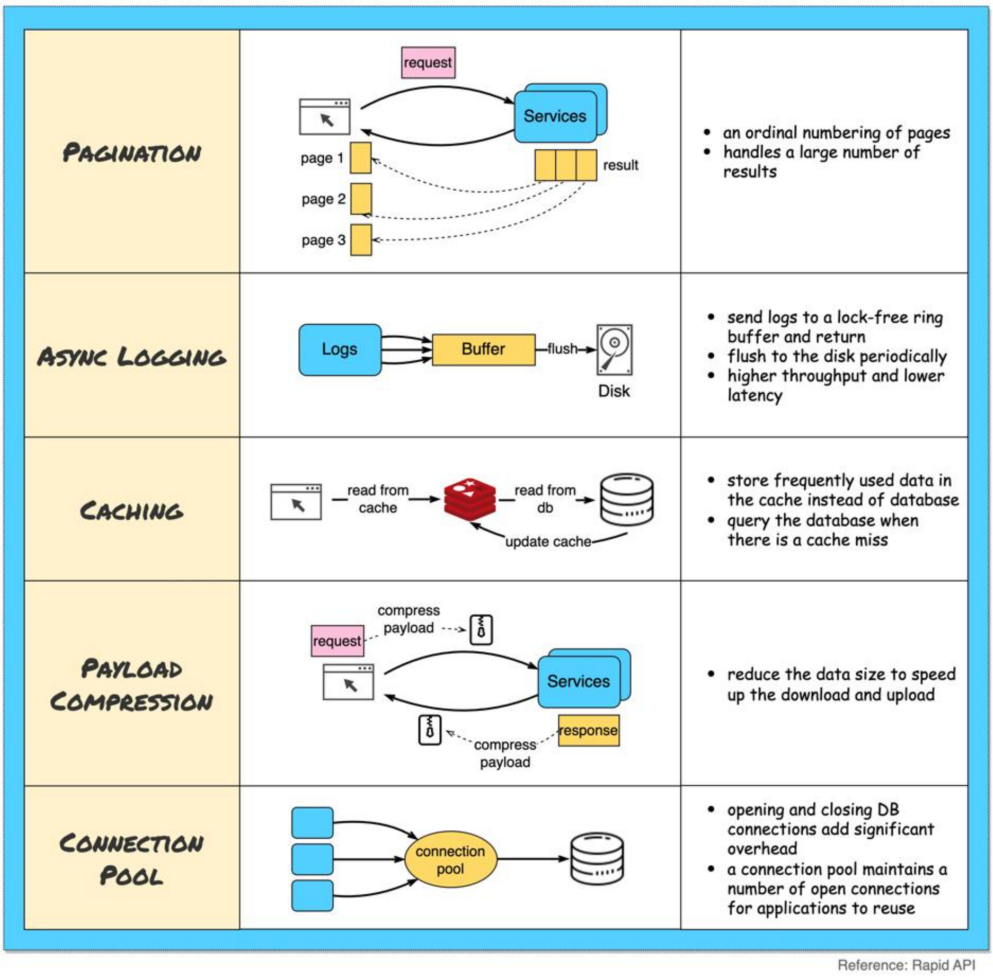
Dynamic Algorithms

- Least connections
  - A new request is sent to the service instance with the least concurrent connections.
- Least response time
  - A new request is sent to the service instance with the fastest response time.



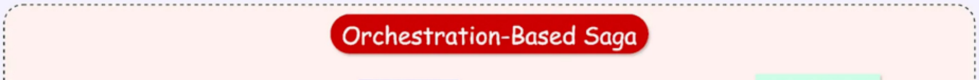


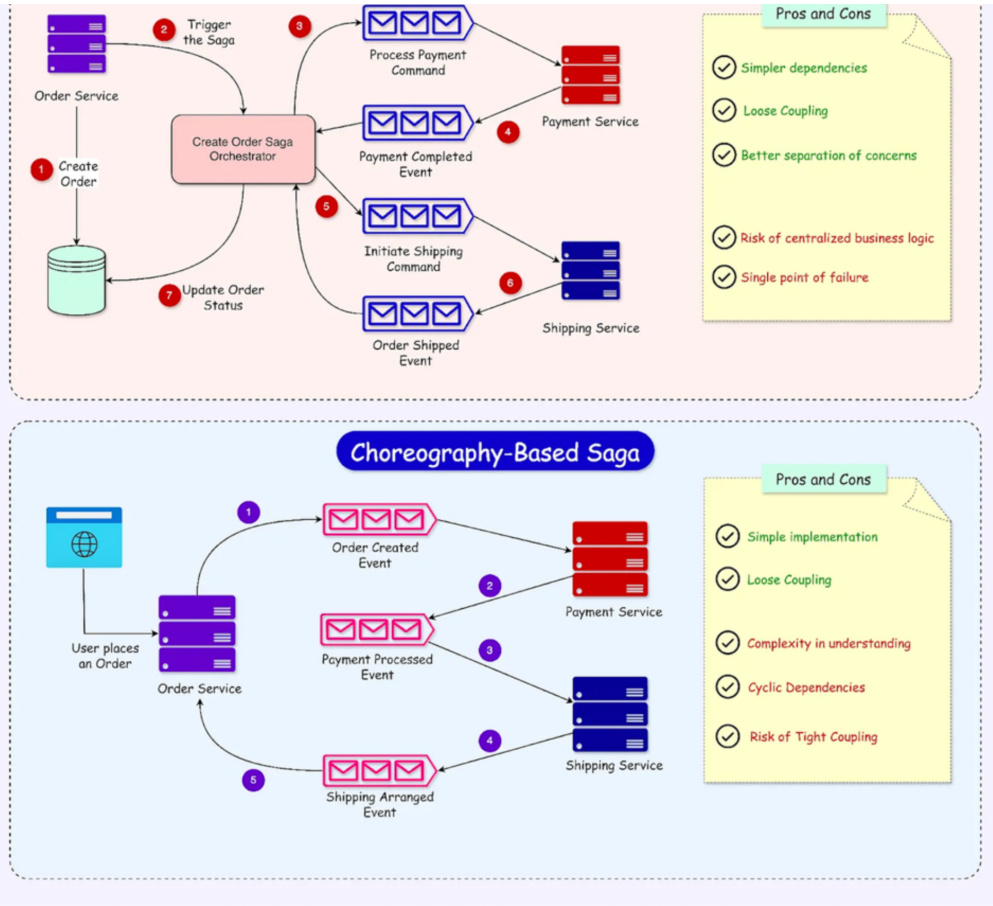
How to Improve API Performance?

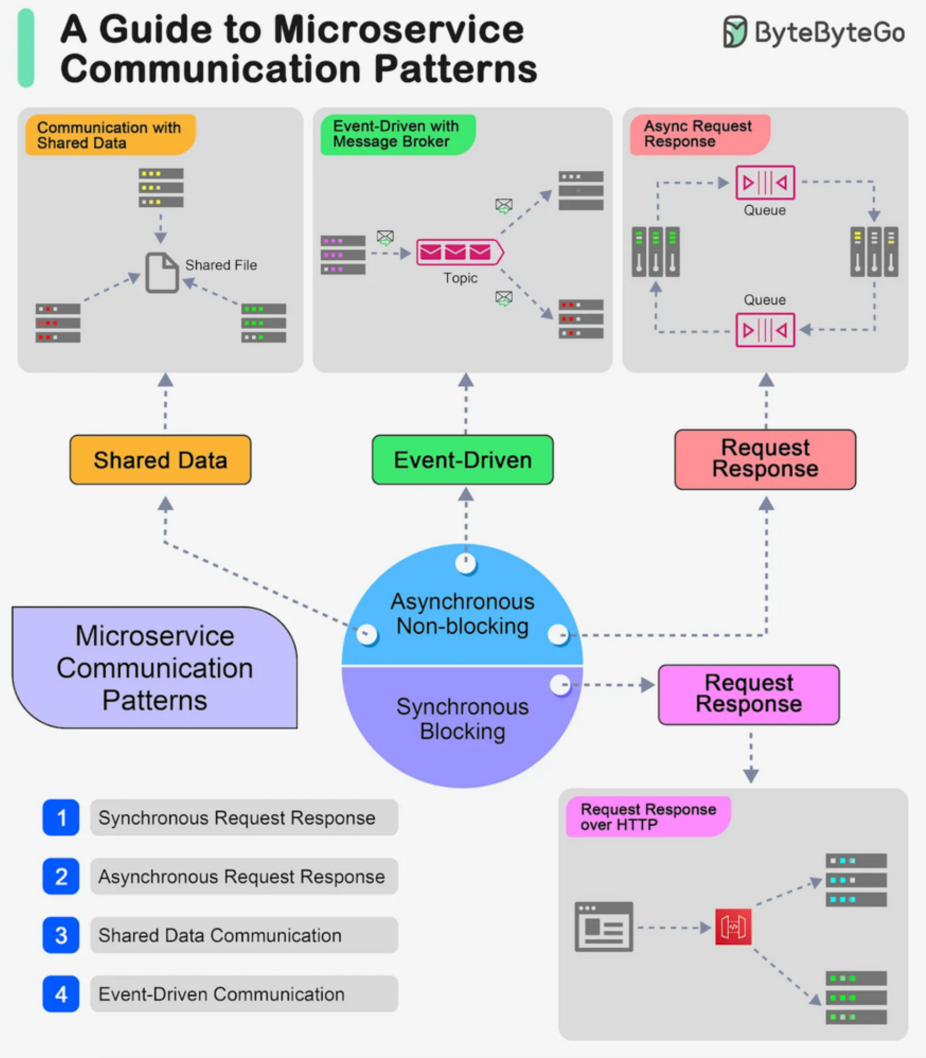


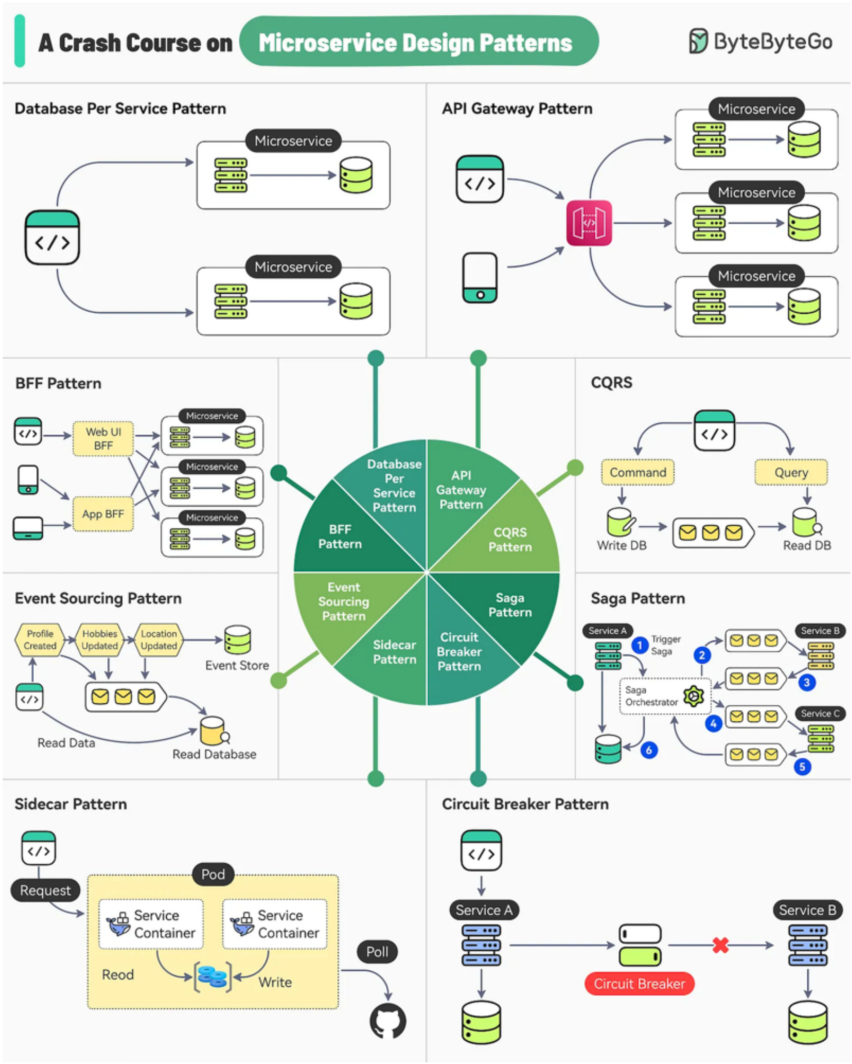
Reference: Rapid API

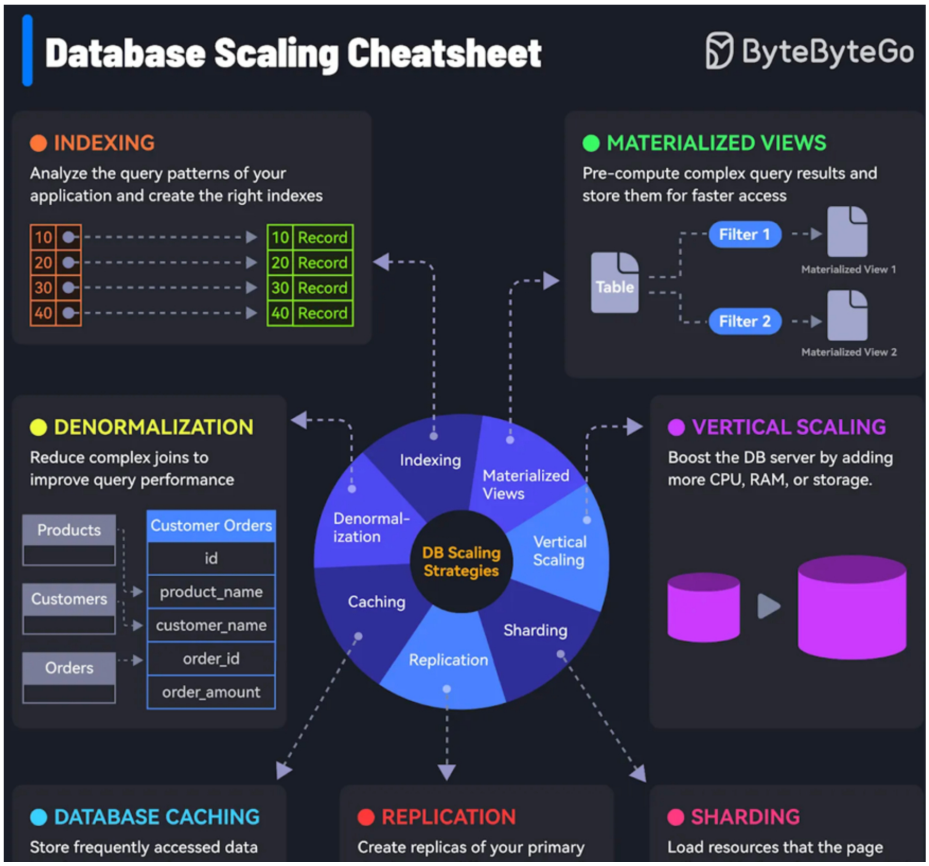
A Cheatsheet on Saga Pattern

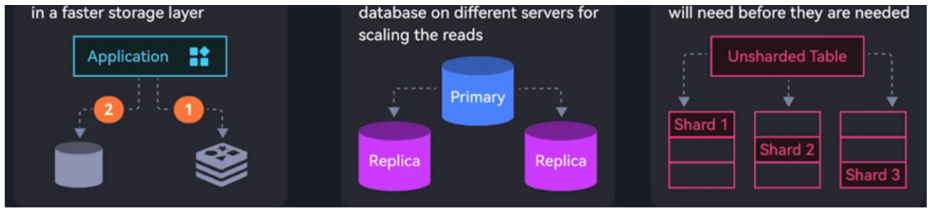


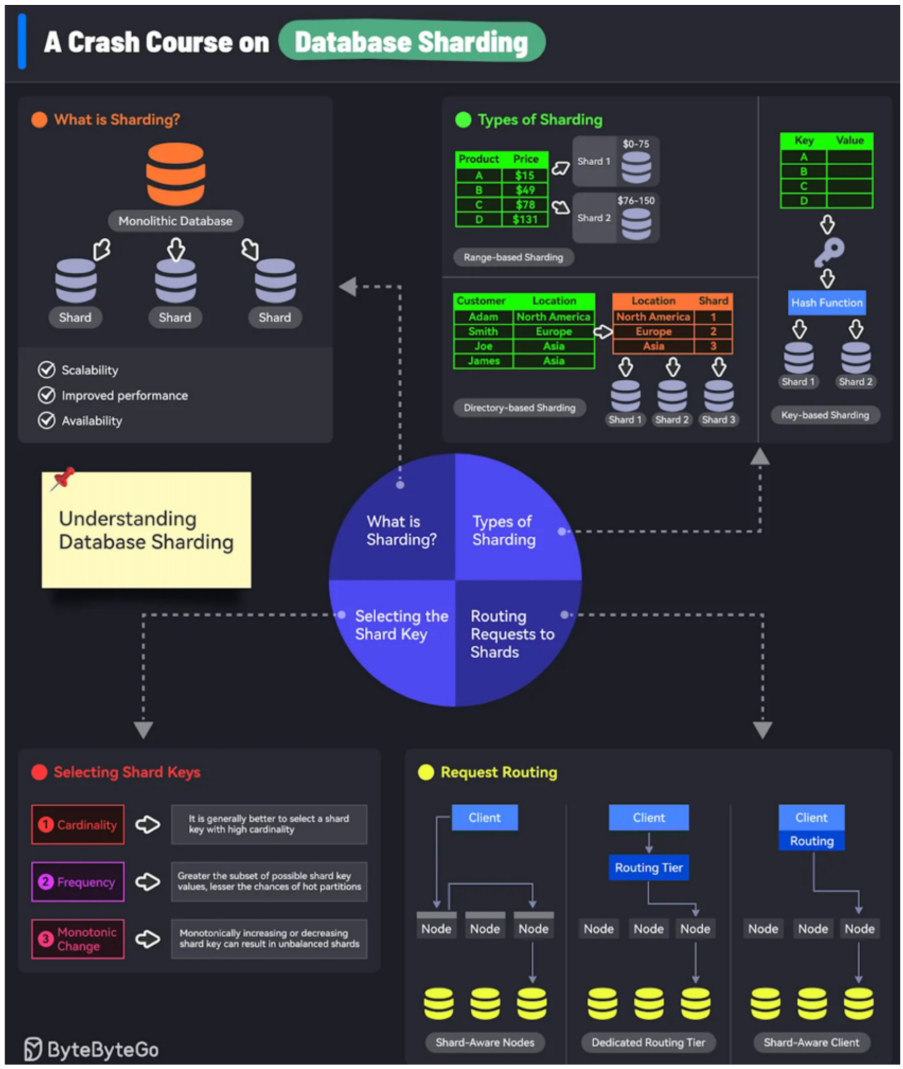

Order Service
Trigger the Saga
Create Order
Create Order Saga Orchestrator
Process Payment Command
Payment Service
Payment Completed Event
Initiate Shipping Command
Shipping Service
Order Shipped Event
Update Order Status
Pros and Cons
Simpler dependencies
Loose Coupling
Better separation of concerns
Risk of centralized business logic
Single point of failure
Choreography-Based Saga
User places an Order
Order Service
Order Created Event
Payment Service
Payment Processed Event
Shipping Service
Shipping Arranged Event
Pros and Cons
Simple implementation
Loose Coupling
Complexity in understanding
Cyclic Dependencies
Risk of Tight Coupling

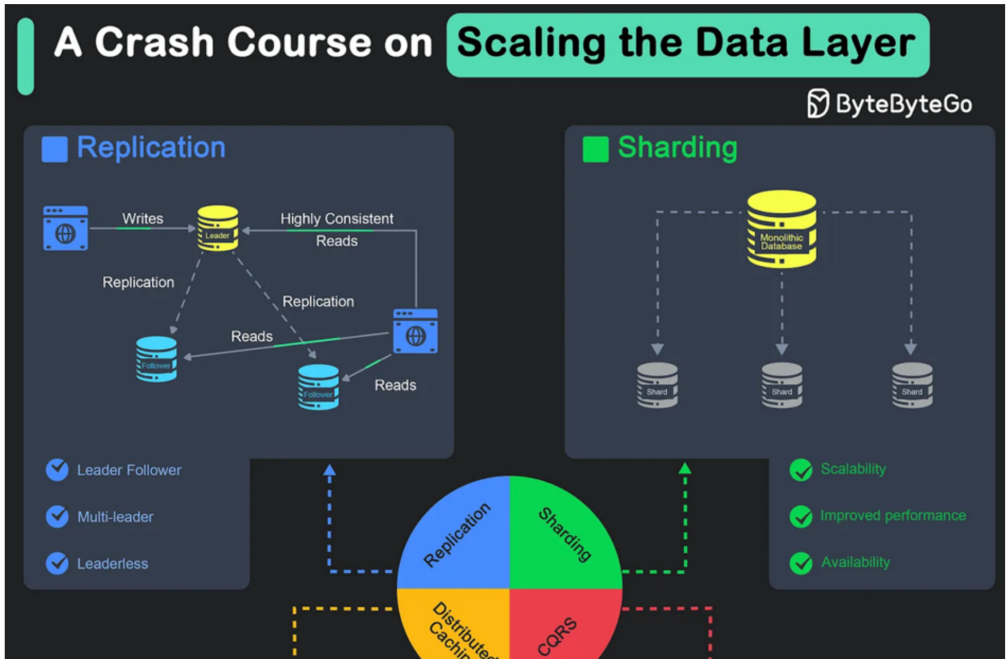

A Guide to Microservice Communication Patterns
ByteByteGo
Communication with Shared Data
Shared File
Event-Driven with Message Broker
Topic
Async Request Response
Queue
Queue
Shared Data
Event-Driven
Request Response
Microservice Communication Patterns
Asynchronous Non-blocking
Synchronous Blocking
Request Response
Request Response over HTTP
1 Synchronous Request Response
2 Asynchronous Request Response
3 Shared Data Communication
4 Event-Driven Communication

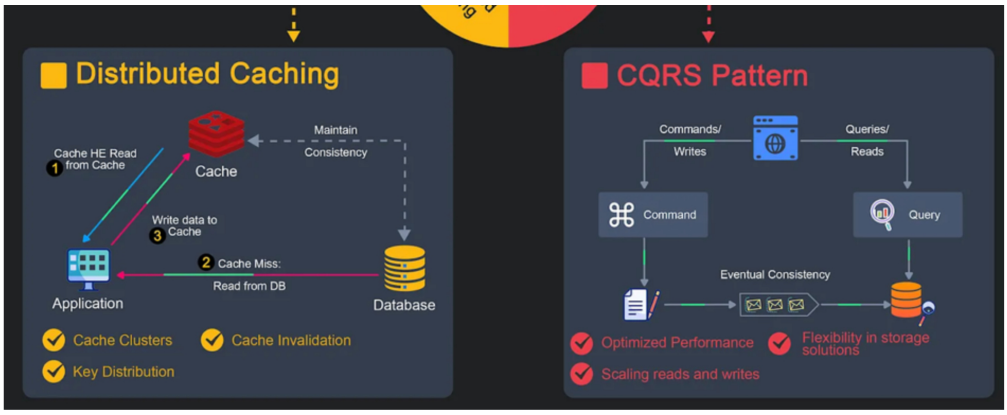

A Crash Course on
Microservice Design Patterns
ByteByteGo
Database Per Service Pattern
Microservice
Microservice
API Gateway Pattern
Microservice
Microservice
Microservice
BFF Pattern
Web UI BFF
App BFF
Microservice
Microservice
Microservice
Database Per Service Pattern
API Gateway Pattern
BFF Pattern
CQRS Pattern
Event Sourcing Pattern
Saga Pattern
Sidecar Pattern
Circuit Breaker Pattern
CQRS
Command
Query
Write DB
Read DB
Event Sourcing Pattern
Profile Created
Hobbies Updated
Location Updated
Event Store
Read Data
Read Database
Saga Pattern
Service A
Trigger Saga
Service B
Saga Orchestrator
Service C
Sidecar Pattern
Request
Pod
Service Container
Service Container
Reod
Write
Poll
Circuit Breaker Pattern
Service A
Circuit Breaker
Service B


Database Scaling Cheatsheet
ByteByteGo
INDEXING
Analyze the query patterns of your application and create the right indexes
10
20
30
40
10 Record
20 Record
30 Record
40 Record
MATERIALIZED VIEWS
Pre-compute complex query results and store them for faster access
Table
Filter 1
Filter 2
Materialized View 1
Materialized View 2
DENORMALIZATION
Reduce complex joins to improve query performance
Products
Customers
Orders
Customer Orders
id
product_name
customer_name
order_id
order_amount
Indexing
Materialized Views
Denormal-ization
DB Scaling Strategies
Vertical Scaling
Caching
Sharding
Replication
VERTICAL SCALING
Boost the DB server by adding more CPU, RAM, or storage.
DATABASE CACHING
Store frequently accessed data
REPLICATION
Create replicas of your primary
SHARDING
Load resources that the page


in a faster storage layer
Application
2
1
database on different servers for scaling the reads
Primary
Replica
Replica
will need before they are needed
Unsharded Table
Shard 1
Shard 2
Shard 3


A Crash Course on Database Sharding
What is Sharding?
Monolithic Database
Shard
Shard
Shard
Scalability
Improved performance
Availability
Types of Sharding
Product Price
A $15
B $49
C $78
D $131
Shard 1
$0-75
Shard 2
$76-150
Range-based Sharding
Customer Location
Adam North America
Smith Europe
Joe Asia
James Asia
Location Shard
North America 1
Europe 2
Asia 3
Shard 1
Shard 2
Shard 3
Directory-based Sharding
Key Value
A
B
C
D
Hash Function
Shard 1
Shard 2
Key-based Sharding
Understanding Database Sharding
What is Sharding?
Types of Sharding
Selecting the Shard Key
Routing Requests to Shards
Selecting Shard Keys
1 Cardinality
It is generally better to select a shard key with high cardinality
2 Frequency
Greater the subset of possible shard key values, lesser the chances of hot partitions
3 Monotonic Change
Monotonically increasing or decreasing shard key can result in unbalanced shards
Request Routing
Client
Node
Node
Node
Shard-Aware Nodes
Client
Routing Tier
Node
Node
Node
Dedicated Routing Tier
Client
Routing
Node
Node
Node
Shard-Aware Client
ByteByteGo


A Crash Course on Scaling the Data Layer
ByteByteGo
Replication
Writes
Leader
Highly Consistent Reads
Replication
Replication
Reads
Follower
Follower
Reads
Leader Follower
Multi-leader
Leaderless
Sharding
Monolithic Database
Shard
Shard
Shard
Scalability
Improved performance
Availability
Replication
Sharding
CQRS



LINUX COMMANDS :

```
1.) Networking and System Monitoring
    * netstat – Displays network connections, routing tables, and interface statistics.
    * ss – Shows socket statistics (similar to netstat but faster).
    * nmap – Network scanner for discovering hosts and services.
    * iftop – Real-time bandwidth usage of network interfaces.
    * htop – Interactive system monitor for CPU, memory usage, and process management.
    * ping – Tests connectivity between hosts.
    * traceroute – Shows the path packets take to reach a host.
    * curl – Transfers data from or to a server, supporting many protocols.
    * wget – Downloads files from the internet.
    * dig – Queries DNS information.
    * nslookup – Looks up DNS records.
    * arp – Displays or manipulates the system's ARP cache.
    * ip – Shows/manages IP addresses, routing, and devices.
    * route – Shows/manages IP routing tables.
    * nc (netcat) – Reads and writes data across networks.
    * iptables – Manages firewall rules.

2.) Disk and Filesystem Management
    * df – Displays disk space usage.
    * du – Shows disk usage of files and directories.
    * ncdu – Text-based disk usage analyzer.
    * rsync – File and directory synchronization and backup.
    * lsof – Lists open files and their associated processes.
    * mount / umount – Mounts and unmounts filesystems.
    * lsblk – Lists information about all available block devices.
    * blkid – Shows or changes block device attributes.
    * mkfs – Formats a disk with a specific filesystem.
    * fsck – Checks and repairs filesystem consistency.
    * e2fsck – Checks and repairs ext2/ext3/ext4 filesystems.
    * resize2fs – Resizes ext2/ext3/ext4 filesystems.

3.) System Resource Management
    * ps aux – Lists all running processes with details.
    * pkill – Kills processes by name.
    * nice / renice – Adjusts the priority of running processes.
    * sar – Collects and reports system activity information.

4.) Process Management
    * top – Displays live system processes.
    * kill – Terminates a process by ID.
    * killall – Terminates processes by name.
    * jobs – Lists background jobs in the current session.
    * bg / fg – Moves jobs to the background or foreground.
    * ps – Displays currently running processes.
    * pmap – Displays memory map of a process.

5.) File System and Disk Management
    * fdisk – Disk partitioning tool.
    * parted – Advanced disk partitioning.
    * tune2fs – Adjusts filesystem parameters on ext2/ext3/ext4 filesystems.

6.) Archiving and Compression
    * tar – Archives and compresses files.
    * zip / unzip – Compresses and extracts files in ZIP format.

7.) Security and Permissions
    * chmod – Changes file permissions.
    * chown – Changes file ownership.
    * iptables – Manages firewall rules.
    * ufw – User-friendly firewall for Ubuntu-based systems.

8.) System Information and Diagnostics
    * **dmesg** – Displays system messages (e.g., boot logs, hardware issues).
    * **iostat** – Provides CPU and I/O statistics.
    * strace – Traces system calls and signals for a command.
    * ltrace – Tracks library calls made by a command.
```

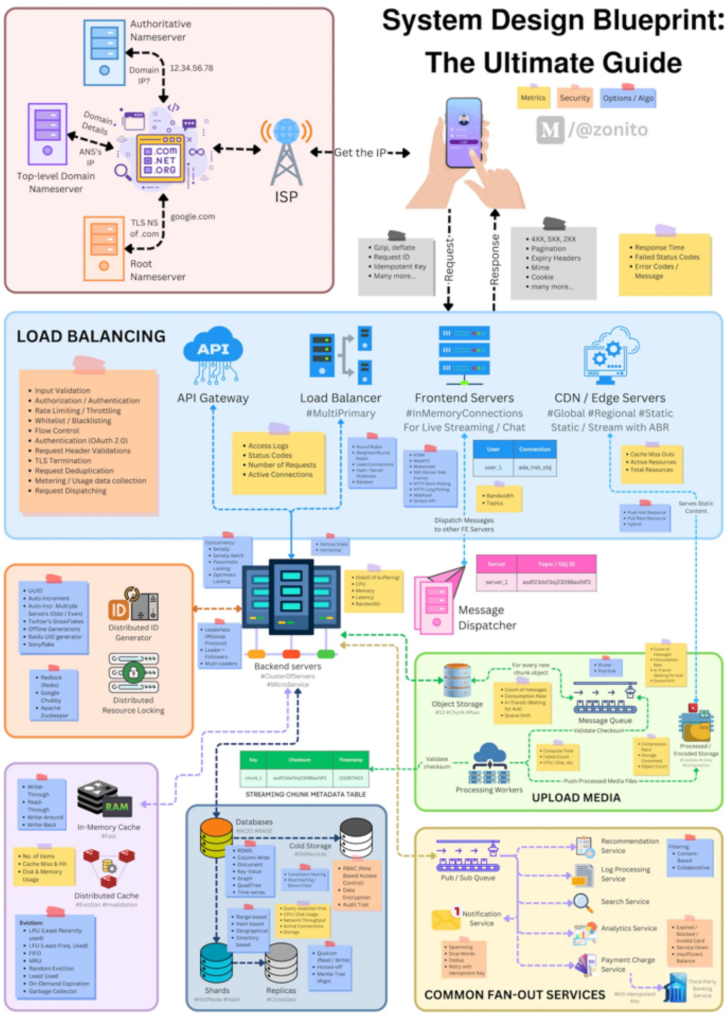
```
9.) Device and Hardware Control
    * ioctl – System call for device-specific input/output operations (used in programming).
    * mknod – Creates device files manually.
    * dmsetup – Device mapper tool for managing logical volumes.
    * hdparm – Disk parameter control for tuning HDD/SSD performance.
    * setpci – Configures PCI device settings.
    * modprobe – Manages kernel modules dynamically.
    * insmod / rmmod – Manually inserts and removes kernel modules.

10.) Kernel and System Configuration
    * sysctl – Configures kernel parameters at runtime.
    * ethtool – Configures network interface hardware settings.
    * perf – Performance analysis tool to profile applications.

11.) File and Directory Management
    * ls – Lists files and directories.
    * cd – Changes the current directory.
    * pwd – Displays the present working directory.
    * mkdir – Creates new directories.
    * rmdir – Removes empty directories.
    * cp – Copies files or directories.
    * mv – Moves or renames files or directories.
    * rm – Removes files or directories.
    * find – Searches for files and directories in a directory hierarchy.
    * locate – Finds files quickly using an index.
    * touch – Creates an empty file or updates an existing file's timestamp.

12.) Developer and Debugging Tools
    * gcc – GNU Compiler Collection for compiling code.
    * gdb – Debugger for programs written in C/C++.
    * make – Utility for building and managing projects.
    * valgrind – Tool for memory debugging, profiling, and leak detection.
    * strace – Traces system calls.
    * ltrace – Traces library calls.
    * objdump – Displays information about object files.
    * ldd – Lists shared libraries required by a program.

14.) System and Hardware Information
    * uname – Shows system information.
    * hostname – Displays or sets the system's hostname.
    * lscpu – Displays CPU architecture information.
    * lsusb – Lists USB devices.
    * lspci – Lists PCI devices.
    * dmidecode – Displays hardware information from the BIOS.
    * uptime – Shows how long the system has been running.
    * free – Shows memory usage.
    * **dstat** – Displays system resource usage in real-time.
    * **mpstat** – Shows CPU activity for each processor.
```

Consistent Hashing Concepts

Consistent hashing is a technique primarily used in distributed systems to distribute data evenly across a dynamic set of nodes (e.g., servers) while minimizing the amount of data that needs to be redistributed when nodes are added or removed. This approach is commonly used in caching, load balancing, and distributed databases, where node additions and deletions are frequent.

1. Basics of Consistent Hashing
    - In consistent hashing, all nodes and data items are mapped to a circular "hash ring." The primary steps involved are:
    - Hashing the nodes: Each node is assigned a position on the ring by hashing its identifier (e.g., its IP address or name).
    - Hashing the data keys: Each data item (e.g., a key-value pair) is also hashed to a position on the ring.
    - In this setup, each key is stored in the first node it encounters while moving clockwise from its hash position.
2. Adding Nodes
    - When a new node is added:
    - The new node is hashed and placed at its corresponding position on the ring.
    - Data that was previously stored in the next clockwise node now maps to the new node.
    - Only the keys that are within the interval between the new node and the next clockwise node need to be redistributed. This minimizes the number of keys that need to be moved, keeping changes as minimal as possible.
3. Deleting Nodes
    - When a node is removed:
    - The next node in the clockwise direction will take over the responsibility for the data previously managed by the removed node.
    - Only the data that was mapped to the deleted node needs to be moved to the next clockwise node.
    - This allows the system to recover gracefully, and again, only a minimal amount of data is redistributed.
4. Load Distribution and Virtual Nodes
    - One challenge in consistent hashing is that, if nodes are hashed directly, it might lead to uneven data distribution due to the randomness of hash positions. To counteract this, virtual nodes are often introduced:
    - Each physical node is represented by multiple virtual nodes, each with its own hash position on the ring.
    - By adding virtual nodes, we ensure a more even distribution, as each physical node handles multiple intervals on the ring, balancing out any “hotspots.”
5. Summary of Load Distribution:
    - Each key is stored on the first node clockwise from its hash position.
    - When nodes are added or removed, only the immediate neighboring keys need to be moved, minimizing redistribution.
    - Virtual nodes help evenly distribute data by spreading each physical node’s load across multiple locations on the ring.

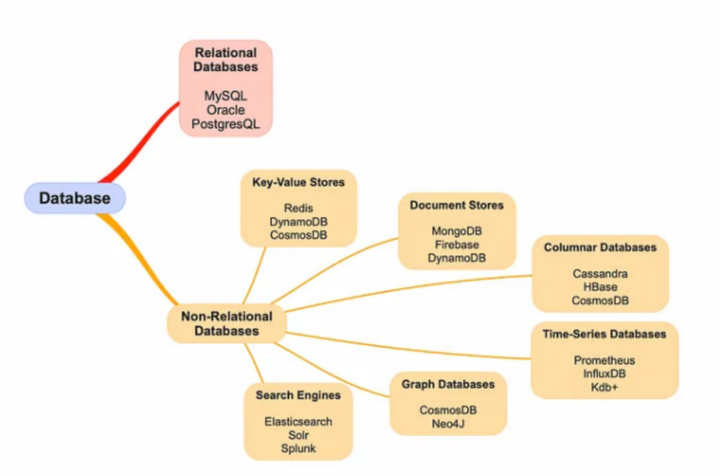
- This approach keeps consistent hashing efficient, scalable, and adaptable to frequent node changes.



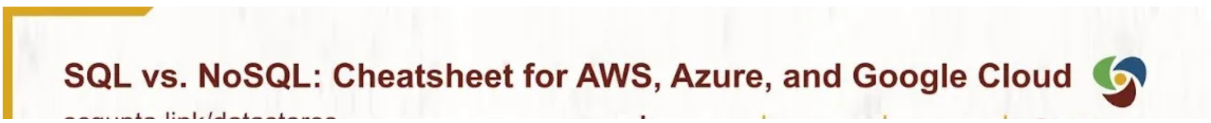
Database types :

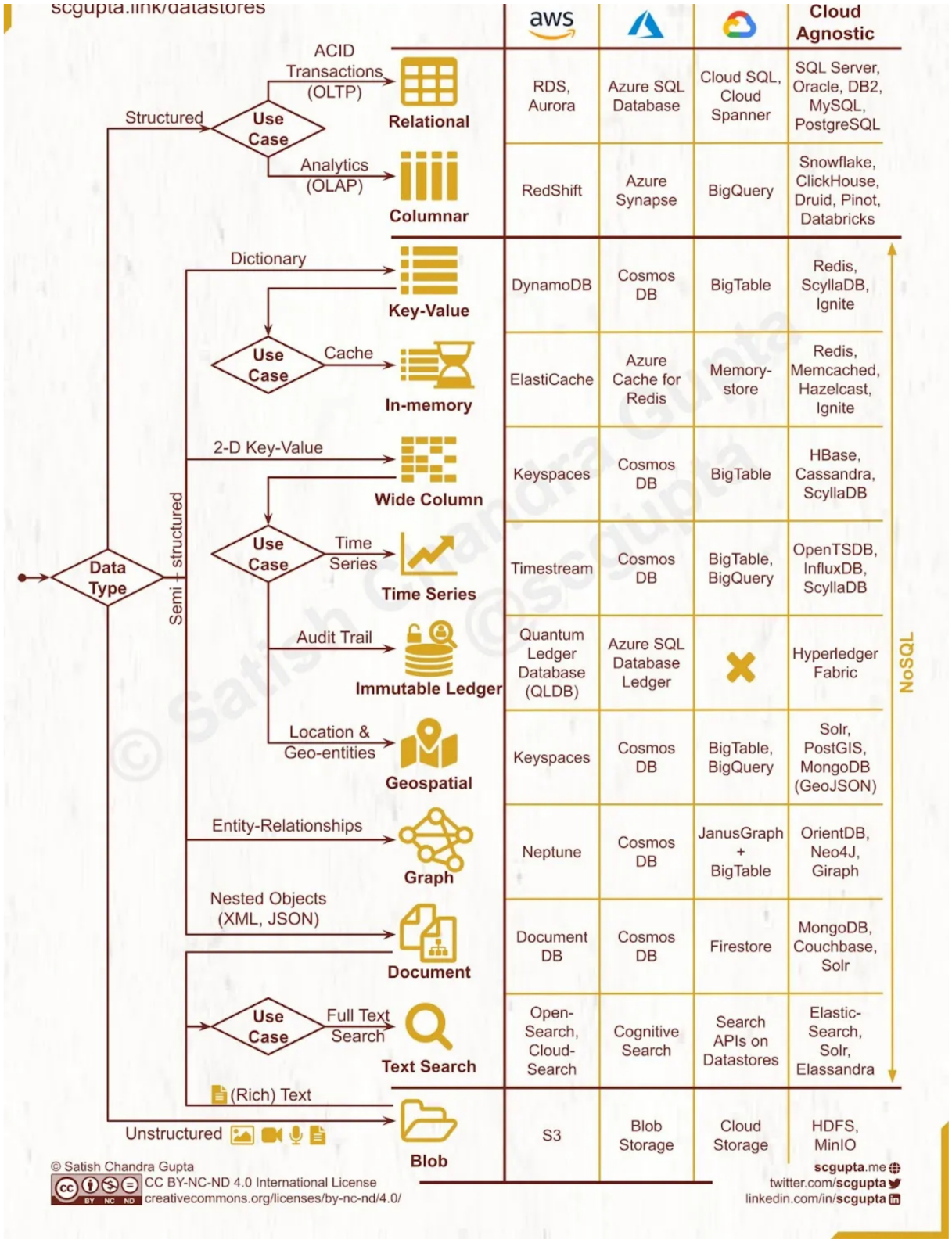
https://medium.com/@i.vikash/looking-for-a-database-for-your-next-system-design-d65df8a778c5

Let's quickly see the type of databases:-



DB CheatSheet : https://blog.bytebytego.com/p/understanding-database-types





Choose which database for your System Design.

Comparing B-Tree vs. LSM Tree

Aspect	B-Tree	LSM Tree
Read Performance	Faster random reads, lower latency	Higher latency (requires merging)
Write Performance	Slower due to random writes	Faster due to sequential appends
Storage Efficiency	Moderate	High (compaction + compression)
Use Case	OLTP, frequent queries	OLAP, logging, event streaming

Conclusion

- **For Read-Heavy Workloads**: Prefer B-Tree-based databases like MySQL, PostgreSQL, or specialized key-value stores.
- **For Write-Heavy Workloads**: Opt for LSM Tree-based systems like Cassandra, HBase, or RocksDB.
- **For Space Optimization**: Use LSM Tree-based engines with compression or columnar storage systems tailored for analytics.

Selecting a database involves not just workload type but also operational factors like scalability, consistency requirements, and latency tolerance.

2. Based on Scalability

Vertical Scalability:

- **Definition**: Scaling by adding more resources (CPU, RAM) to a single server.
- **Databases**: Relational databases (e.g., PostgreSQL, MySQL).
- **Consideration**: Limited by hardware.

Horizontal Scalability:

- **Definition**: Scaling by adding more servers to the cluster.
- **Databases**:
  - NoSQL databases (e.g., MongoDB, Cassandra, DynamoDB).
  - NewSQL databases (e.g., CockroachDB, Google Spanner) for distributed SQL.
- **Use Cases**:
  - High-concurrency systems with large datasets.
  - Example: Social media platforms.

3. Based on Consistency Requirements

Strong Consistency:

- **Definition**: All nodes reflect the latest write for every read.
- **Best Fit**:
  - Relational databases (PostgreSQL, MySQL with InnoDB).
  - NewSQL databases (CockroachDB, Spanner).
- **Use Cases**:
  - Financial transactions, inventory management.

Eventual Consistency:

- **Definition**: Nodes may be out of sync temporarily but converge eventually.
- **Best Fit**:
  - NoSQL databases (Cassandra, DynamoDB).
  - Suitable for distributed systems prioritizing availability.
- **Use Cases**:
  - Social media feeds, product catalogs.

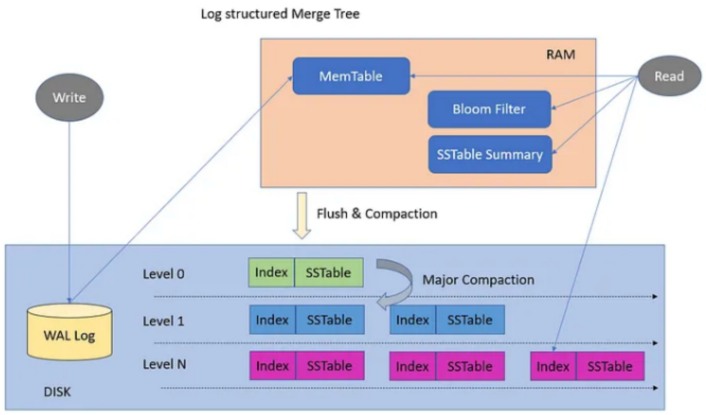
Tunable Consistency:

- **Definition**: Choose between strong and eventual consistency per query.
- **Best Fit**:
  - DynamoDB, Cassandra.

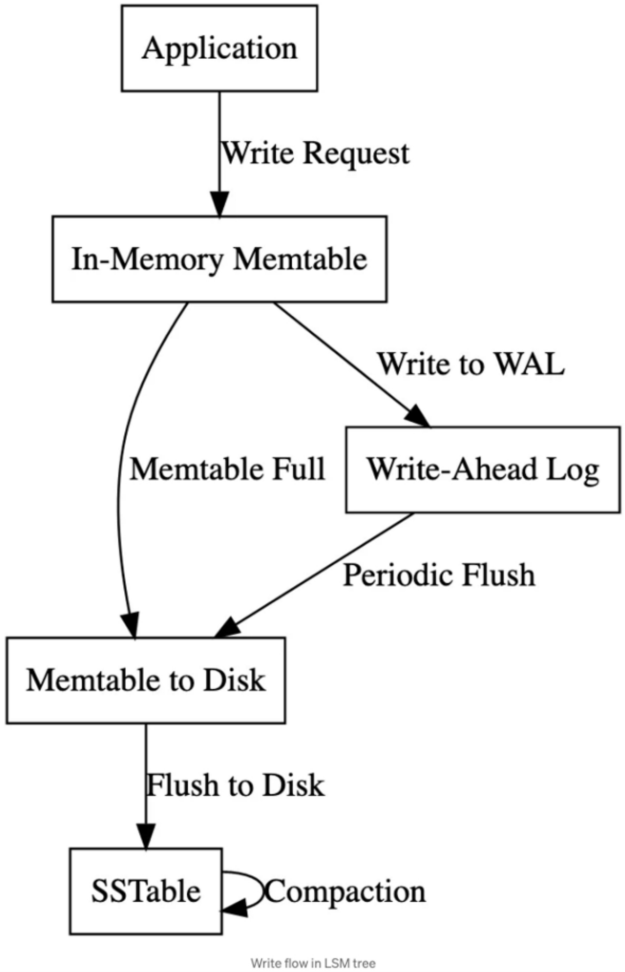
- **Use Cases:**
  - Flexible applications with mixed consistency requirements.

LSM Engine :

What is a LSM tree?



Overview of LSM Tree design



Write flow in LSM tree

Anrup14 ★ 20 October 21, 2024 8:41 AM

hi @adityakrverma , Can you please share the link for the part1 post for System design interview questions. I couldn't find it on your profile.

1 Reply