SYSTEM DESIGN CHEAT SHEET

This quick reference quick sheet I made covers the most important system design concepts, making it easy to review key points right before your interview.

Use Cases/Problems	System Design Questions	Component	What it solves	Caveats/Issues	Mitigations	Examples of Tools
- Unified API access: Centralizes client requests.	- Design an API gateway for microservices.	API Gateway	Single entry point, manages authentication and routing.	Can become a bottleneck, adds latency.	- Use multiple gateways with load balancing.	Kong, Apigee, AWS API Gateway
- Security: Manages authentication and authorization.	- Implement secure and scalable API access.				- Implement rate limiting and caching.	
					- Use circuit breakers and retries.	
- High traffic websites: Ensures uptime and balances load.	- Design a scalable web application.	Load Balancer across multiple redundant workers	Distributes traffic across workers, improves reliability and availability.	Single point of failure, adds complexity.	- Use multiple load balancers in different regions.	Nginx, HAProxy, AWS ELB
- Scalable APIs: Distributes incoming requests.	- Build a highly available online service.				- Implement health checks.	
					- Use DNS-based load balancing.	
- Financial transactions: Requires ACID compliance.	- Design a financial transaction system.	SQL Database	Strong ACID properties, structured data, complex queries.	Limited scalability, schema management.	- Implement sharding.	MySQL, PostgreSQL, MS SQL Server
- Complex queries: Needs structured and relational data.	- Create a scalable relational database.				- Use read replicas.	
					- Employ clustering and partitioning.	

- Large-scale data: Supports horizontal scaling.	- Design a large-scale user profile store.	NoSQL Database	Flexible schema, horizontal scalability, high performance.	Eventual consistency, limited transaction support.	- Use consistency settings (e.g., quorum reads/writes).	MongoDB, Cassandra, DynamoDB
- Unstructured data: Flexible schema adapts to changes.	- Create a scalable data storage solution.				- Design for idempotent operations.	
					- Implement conflict resolution strategies.	
- High availability: Ensures data is replicated and available.	- Design a data replication strategy.	Data Replication	Ensures data durability, to ensure system availability.	Increases costs, consistency issues.	- Use asynchronous replication.	AWS RDS standby (synchronous), AWS RDS Read Replicas (asynchronous), MongoDB Replica Set (asynchronous)
	- Implement a highly available database system.				- Implement conflict resolution.	
					- Use multi-master replication.	
- High read load: Reduces latency for frequent reads.	- Design a high-performance caching layer.	Cache	Reduces latency, decreases load on databases.	Cache consistency issues, potential for stale data.	- Implement cache invalidation strategies.	Redis, Memcached
- Session storage: Speeds up access to session data.	- Optimize read-heavy workload.				- Use Time-to-Live (TTL) settings.	
					- Employ write-through or write-back caching.	
- Real-time analytics: Requires fast data access.	- Design a real-time analytics system.	In-Memory Database	Extremely fast data retrieval, reduces latency.	Volatile storage, high memory cost.	- Enable persistence options.	Redis, Memcached
- Leaderboards: High-speed data retrieval is crucial.	- Create a fast leaderboard service.				- Use hybrid storage models (in-memory + disk).	
					- Implement data backup strategies.	

- Event streaming: Manages high-throughput data streams.	- Design a real-time event streaming platform.	Message Broker	Facilitates message exchange, supports multiple patterns.	Bottleneck potential, delivery guarantees.	- Use scalable brokers with partitions.	Apache Kafka, RabbitMQ, ActiveMQ
- Real-time processing: Facilitates real-time data flows.	- Implement a reliable messaging system.				- Implement backpressure handling.	
					- Monitor message broker performance.	
- Event-driven systems: Manages asynchronous events.	- Design an event-driven architecture.	Distributed Queue	Manages asynchronous communication, decouples components.	Message ordering and delivery guarantees.	- Use message brokers with strong ordering guarantees.	Apache Kafka, RabbitMQ, AWS SQS
- Microservices: Decouples service communication.	- Create a reliable task processing system.				- Implement idempotent message processing.	
					- Use message deduplication techniques.	
- Large applications: Enhances modularity and scalability.	- Design a scalable microservices architecture.	Microservices	Improves modularity, independent deployment.	Increased communication complexity.	- Use service meshes.	Docker, Kubernetes, Istio
- Continuous delivery: Facilitates independent deployment.	- Build a modular, independently deployable system.				- Implement standardized APIs.	
					- Use centralized logging and monitoring.	
- Microservices: Enables service discovery.	- Design a service discovery mechanism.	Service Registry	Tracks services and their instances.	High availability required, consistency	- Use distributed service registries.	Consul, Eureka, Zookeeper

- Dynamic environments: Tracks changing service instances.	- Implement dynamic service registration.			issues.	- Implement regular health checks.	
					- Use consensus algorithms for consistency.	
- Content-heavy sites: Improves load times for users.	- Design a content delivery system.	CDN (Content Delivery Network)	Reduces latency, improves load times.	Cache invalidation complexity, cost.	- Implement cache purging strategies.	Cloudflare, Akamai, AWS CloudFront
- Global reach: Distributes content across regions.	- Optimize a global website's performance.				- Use regional CDNs.	
					- Monitor CDN performance and hit rates.	
- Business intelligence: Centralizes analytics data.	- Design a data warehouse for analytics.	Data Warehouse	Centralizes data, supports complex queries.	High storage and maintenance costs.	- Use data compression and partitioning.	Amazon Redshift, Snowflake, Google BigQuery
- Historical analysis: Supports complex querying over large datasets.	- Build a scalable business intelligence platform.				- Implement data lifecycle management.	
					- Use cloud-based, scalable data warehouses.	
- E-commerce sites: Provides fast product search.	- Design a product search system.	Search Engine	Enables fast search over large datasets.	Indexing and maintenance required.	- Implement efficient indexing strategies.	Elasticsearch, Solr, Algolia
- Large datasets: Enables full-text search over extensive data.	- Implement a scalable search solution.				- Use distributed search architectures.	
					- Optimize search queries and relevance.	

- Media storage: Handles large files like images and videos.	- Design a scalable file storage system.	File Storage	Scales with data growth, handles unstructured data.	Backup and redundancy required, retrieval latency.	- Use distributed file systems.	AWS S3, Google Cloud Storage, HDFS
- Backup solutions: Stores and retrieves backups.	- Implement a reliable backup solution.				- Implement multi-region replication.	
					- Use lifecycle policies for data management.	
- Data warehousing: Prepares data for analysis.	- Design an ETL pipeline for a data warehouse.	ETL Pipeline	Facilitates data integration and analysis.	Complex to build and maintain.	- Use managed ETL services.	Apache Nifi, AWS Glue, Talend
- Data migration: Transforms data from multiple sources.	- Build a reliable data integration system.				- Implement monitoring and error handling.	
					- Use data validation and transformation tools.	
- System reliability: Monitors uptime and performance.	- Design a system monitoring solution.	Monitoring System	Tracks system health, enables alerting.	High overhead, potential noise.	- Use threshold tuning and anomaly detection.	Prometheus, Grafana, Datadog
- Issue detection: Alerts for anomalies and failures.	- Implement an alerting and dashboard system.				- Implement efficient data collection.	
					- Use centralized monitoring dashboards.	
- Debugging: Captures logs for issue diagnosis.	- Design a centralized logging system.	Logging System	Aids in auditing and troubleshooting.	Large data volumes, storage and querying.	- Use log rotation and retention policies.	ELK Stack, Splunk, Fluentd
- Compliance: Maintains audit trails.	- Implement a scalable logging and analysis solution.				- Implement centralized logging.	
					- Optimize log storage and indexing.	

- Secure applications: Manages user identity and access.	- Design a secure authentication system.	Authentication Service	Enhances security, manages user authentication.	Single point of failure, security measures needed.	- Use multi-factor authentication.	OAuth, Okta, Auth0
- Single sign-on: Centralizes authentication across services.	- Implement a single sign-on solution.				- Implement redundancy and failover.	
					- Use secure token storage and management.	
- Containerized apps: Automates container management.	- Design a container orchestration system.	Orchestration Tool	Automates deployment and management.	Adds complexity, learning curve.	- Use managed orchestration services.	Kubernetes, Docker Swarm, Mesos
- Microservices: Coordinates service deployments.	- Implement a CI/CD pipeline for microservices.				- Implement robust CI/CD pipelines.	
					- Use monitoring and scaling tools.	
- Dynamic applications: Centralizes config changes.	- Design a configuration management system.	Configuration Service	Centralizes configuration management.	Single point of failure, secure access needed.	- Use distributed configuration stores.	Consul, etcd, Spring Cloud Config
- Large systems: Manages configurations across services.	- Implement dynamic configuration updates.				- Implement encryption for sensitive data.	
					- Use versioning and rollback mechanisms.	
- Real-time dashboards: Aggregates live data feeds.	- Design a real-time analytics system.	Real-Time Data Aggregation	Enables real-time analytics and monitoring.	High complexity, data velocity issues.	- Use stream processing frameworks.	Apache Flink, Apache Storm, AWS Kinesis

- Monitoring: Provides instant insights from data streams.	- Implement a live data aggregation platform.				- Implement windowing and aggregation techniques.	
					- Monitor and scale processing infrastructure.	
- Microservices: Tracks requests across services.	- Design a distributed tracing system.	Distributed Tracing	Aids in debugging and performance monitoring.	High overhead, integration required.	- Use sampling to reduce overhead.	Jaeger, Zipkin, OpenTracing
- Performance tuning: Identifies bottlenecks and delays.	- Implement performance monitoring for microservices.				- Implement efficient trace storage.	
					- Use correlation IDs for request tracking.	
- Fault tolerance: Prevents system overloads.	- Design a fault-tolerant microservices system.	Circuit Breaker	Protects services from cascading failures.	Adds complexity, tuning needed.	- Use monitoring tools to detect failures.	Hystrix, Resilience4j, Istio
- Resilient services: Isolates failures in microservices.	- Implement circuit breakers for service reliability.				- Implement fallback strategies.	
					- Use retries and exponential backoff.	
- API management: Protects against request floods.	- Design an API rate limiting system.	Rate Limiter	Controls request rate, prevents abuse.	Can impact user experience.	- Use dynamic rate limiting.	Kong, Envoy, Nginx
- Fair resource allocation: Ensures fair usage policies.	- Implement a fair resource allocation mechanism.				- Implement user-based quotas.	
					- Use monitoring to adjust limits.	

- Periodic tasks: Automates recurring jobs.	- Design a job scheduling system.	Scheduler	Manages background jobs and tasks.	Requires monitoring, can become bottleneck.	- Use distributed schedulers.	Apache Airflow, Celery, Kubernetes CronJobs
- Batch processing: Manages large data processing tasks.	- Implement a reliable task processing system.				- Implement job prioritization.	
					- Use monitoring and retry mechanisms.	
- Microservices: Handles inter-service communication.	- Design a service mesh for microservices.	Service Mesh	Manages microservices communication.	Adds operational complexity.	- Use managed service meshes.	Istio, Linkerd, Consul Connect
- Observability: Provides insights into service interactions.	- Implement observability for service interactions.				- Implement automation tools.	
					- Use monitoring and observability tools.	
- Disaster recovery: Ensures data is safe and recoverable.	- Design a backup and recovery system.	Data Backup and Recovery	Ensures data durability, protects against data loss.	Resource-intensive, regular testing needed. Increases costs. if backups are not up to date there will be data loss. Backup may not be accessible.	- Use automated backup solutions.	Use native backup capabilites of the data store, or centralized backup products like AWS Backup, Google Cloud Backup, Veeam
- Data integrity: Maintains backups for compliance.	- Implement a reliable disaster recovery solution.				- Implement multi-region storage.	
					- Regularly test backup and recovery processes.	
- Social networks: Models complex relationships.	- Design a social network graph database.	Graph Database	Efficiently handles graph-based data and relationships.	Steep learning curve, non-graph query	- Use graph-specific optimizations.	Neo4j, Amazon Neptune, OrientDB

- Recommendation engines: Analyzes connected data.	- Implement a recommendation engine.			inefficiency.	- Implement hybrid models for different data types.	
					- Use indexing and caching for performance.	
- Big data analytics: Stores and processes vast data.	- Design a big data analytics platform.	Data Lake	Supports diverse data types and analytics.	Governance required, risk of becoming data swamp.	- Use metadata management.	AWS Lake Formation, Azure Data Lake, Hadoop
- Data warehousing: Prepares raw data for analytics.	- Implement a data lake for diverse data types.				- Implement data cataloging.	
					- Use data lifecycle policies.	
- Event-driven architectures: Processes data streams in real-time.	- Design a real-time data streaming system.	Data Streaming Platform	Facilitates real-time data processing.	High operational complexity.	- Use managed streaming services.	
- Analytics: Real-time insights from continuous data flow.	- Implement an event-driven architecture.				- Implement scaling strategies.	
					- Monitor and optimize processing.	