Enterprise Software Development

Complete Exam Study Guide - All Topics

1. Software Design Principles

OOP Principles

- **Abstraction:** Hiding how something is implemented
- **Polymorphism:** Many forms - child instance in parent class
- **Encapsulation:** Private, public & protected access modifiers
- **Inheritance:** Child-parent class relationship

SOLID Principles

- **S - Single Responsibility:** A class should have only one job
- **O - Open/Closed:** Open for extension, closed for modification
- **L - Liskov Substitution:** Derived class objects should be substitutable for base class without issues
- **I - Interface Segregation:** Class shouldn't be forced to implement methods it doesn't use
- **D - Dependency Injection:** Class should receive dependencies from external sources (abstraction) instead of creating them internally

Other Design Principles

- **DRY (Don't Repeat Yourself):** Long-term benefit, avoid code duplication
- **WET (Write Everything Twice):** Isolated, short-term - sometimes acceptable for prototyping
- **KISS (Keep It Simple, Stupid):** Simplified, essential code
- **YAGNI (You Aren't Gonna Need It):** No future assumptions - only code what you need!

Design Patterns

- **Singleton:** Single instance of cache, dbs, etc. Provides global access to class instances
- **Factory:** Clean, extendable code through interfaces

Code Smells (Anti-patterns)

- Variable naming (too generic)
- Primitive datatype obsession
- Dead code (not used anywhere)
- One container(class) = one domain(purpose)
- Middle man is bad (classes not providing any value)
- Long functions are bad (doing more than 1 thing)
- Long parameter lists should be reduced
- Overuse of comments
- Duplicate code is bad!

Refactoring: Changing code to make it easier without changing its logic. Kata-log for practice.

2. Architectural Styles

What is Architecture?

- **Martin Fowler:** "Architecture is about the important stuff. Whatever that is."
- Components of system, dependencies, relationships of components, control flow between components

Layered Architecture

Components with similar functionalities organized into horizontal layers. Separation of Concerns - each layer has specific responsibility.

The Four Layers

- **1. Presentation Layer:** UI/visualization, browser, customer interaction (React, React Native) - Front-end
- **2. Business Layer:** Application logic, data aggregation, computation, query requests
- **3. Persistence Layer:** Repositories/ORM mapping, adapters (S3, Elasticsearch, Kafka) - Back-end
- **4. Database Layer:** OLTP DB (Postgres/MySQL), Cache (Redis), Object storage (S3/GCS), db mgmt, policies
- **Utility Layer (Optional):** Data/string utilities, auditing, logging

Note: Communication between layers often builds to MVC pattern. Layers can communicate with other layers sometimes.
Redis: Hashmap as a database (used as Cache). **Elasticsearch:** Works best with Text data.

Monolithic Architecture

Single codebase, everything packaged and deployed as single unit. Client Side, Server Side, Database all in one place.

Advantages

- Easy development with one codebase
- Secure (closed system)
- Easy deployment (single executable)
- Debugging way easier (central logging)
- Can scale horizontally with load balancer + multiple threads/processes

Disadvantages

- Complex reasoning as application grows
- Lack of modularity
- Interdependence - error in any module affects entire application
- Scaling issues - must scale entire app even if only one functionality needs it
- Barrier to technology adoption

Microservices Architecture

Separate modules based on Business Domain. Independent services offering flexibility, easy updates, and easy to scale.

Key Characteristics

- Each service has its own datastore (decentralized DB)
- Other services don't have direct access to data - they request it from the respective service
- Independent deployment and scaling
- **Communication through:** HTTPS API, RPC, or Kafka (Event-driven)

Advantages

- Independent deployment and scaling
- Flexibility in technology choices
- Teams are responsible for developing AND operating their services ("You build it, you run it" - Amazon)

Disadvantages

- Issues often span multiple distributed services - harder debugging
- Higher network latency due to service communication
- More infrastructure required
- End-to-end testing is complex

Conway's Law

"Organizations design systems that mirror their own communication structure." Structure of system mirrors organization structure.

Serverless (AWS Lambda)

- Extreme version of Microservices - each function as a separate service
- Technically serverless = something we don't have to manage
- **Cold start:** Instance takes time to open & start becoming efficient

Key Infrastructure Concepts

- **Load Balancers:** Request divided among multiple servers (can be in Monolith too)
- **Heart-beat:** Manager checks periodically which server is healthy and working
- **Auto-scaling:** AWS automatically starts (allocates) more servers based on new requests
- **Rate-limit:** Set no. of requests per IP in a set amount of time

3. API Fundamentals

REST API Best Practices

- Resource-oriented design - exposes resources through URIs
- **HTTP Methods:** GET (fetch/read), POST (new write), PUT (update), DELETE (delete), PATCH (partial update)
- Proper status codes: 2xx success, 4xx client error, 5xx server error
- API versioning for backward compatibility (new version at every API change, diff deployments)
- Authentication & Authorization (logged in + permissions)
- APIs should retry if fails by n times
- **Idempotency:** Repeated requests produce the same effect without unintended side effects

OpenAPI Specification

- Specification document = API Contract
- Includes data schemas (structure) for request/response validation
- All API with its information (status codes, request, response, body, etc.)
- Document usually in .yaml file
- If document written nicely → can make boilerplate code automatically for API calls
- All we need to do is call and put the parameters - boilerplate has endpoint code, status checking, etc.

RPC (Remote Procedure Calls)

- Instead of sending a request to a resource (API), we call a method that retrieves data & does some business logic
- **LPC (Local Procedure Call):** Definition & calling on same PC
- **RPC (Remote Procedure Call):** Definition & calling on different PC
- **API:** Usually designed around resources (data). **RPC:** Usually designed for actions (methods)
- Server side does all computation for the methods
- Server & Client can have different architectures & languages - need Interface Description Language (IDL)
- **IDL structures define:** parameters, return types, etc. (uniform concept)
- From IDL, we create a proxy (Stub) responsible for all communications b/w Server & Client
- **Stub:** Proxy in client - client can call from server. Proxy ↔ Stub interchangeable
- **Parameter Marshaling:** Translation of local language to how remote PC would understand
- **Parameter Unmarshaling:** Unpacking on receiving end
- **HTTP vs Sockets:** HTTP breaks connection after transferring. Sockets → no connection breaking

gRPC (Google's Remote Procedure Calls)

- .proto file = IDL that generates stubs and takes care of all inner workings
- Proto files generate code for proxy so we can easily call methods in client
- IDL defined both in Client & Server - helps client call & server implement methods
- Build system compiles specification document → boilerplate code we can extend to build our controllers
- Boilerplate has all endpoint code, status checking, etc. - everything except business logic
- .proto (Protocol Buffers) - specifically used in gRPC for efficient binary serialization

gRPC Communication Models

- **Unary RPC:** Single request → Single response
- **Client streaming:** Multiple requests → Single response
- **Server streaming:** Single request → Multiple responses

- **Bidirectional streaming:** Both client and server send messages continuously

gRPC Advantages over HTTP/1.1 REST

- Uses HTTP/2
- Efficient serialization and deserialization
- Client-side load balancing

gRPC Disadvantages

- Lack of browser support and harder debugging
- Uses serialized Protobuf messages (not human-readable like JSON/XML)

HTTP/2 solved: Head-of-Line (HOL) blocking due to single outstanding request in HTTP/1.1

Execution Semantics

- **At-most-once:** No operation performed more than once, even on retries (may lose requests)
- **At-least-once:** With idempotent logic - ideal for financial transactions requiring high reliability
- **Exactly-once:** Hardest to achieve - requires careful design

Fault-Tolerant API Design

- **Timeout:** Should be on every API call (client side) - move on after n seconds
- **Retry:** After n retries move on (poison letter requests)
- **Circuit Breaker:** Opens after failure rate threshold, prevents cascade failures

Circuit Breaker States

- CLOSED → (failures exceed threshold) → OPEN → (wait duration) → HALF-OPEN → test → CLOSED or OPEN
- **Config:** failureRateThreshold (e.g., 50%), waitDurationInOpenState (e.g., 30s), slidingWindowSize (e.g., last 3 calls)

Design for Failure: Exceptions, heart-beat, circuit-breakers

Service Registry

- In microservice communication, used to track available service instances
- **Client-side discovery:** Client queries registry and selects instance
- **Server-side discovery:** Load balancer queries registry

MapReduce

Data on different machines, query runs parallel on all machines. Much faster (distributed systems).

4. Scalability Techniques

Load Balancing

Static Algorithms

- **Round Robin:** Distribute requests sequentially
- **Weighted Round Robin:** Some servers get more requests based on weight
- **IP Hash:** Same client always goes to same server

Dynamic Algorithms

- **Least Connections:** Send to server with fewest active connections
- **Weighted Response Time:** Based on server response speed
- **Resource-based:** Based on server CPU/memory

GeoDNS & CDN

- **GeoDNS:** Route users to geographically closest data center
- **CDN (Content Delivery Network):** Cache static content at edge locations
- **Multi-AZ (Availability Zone):** Replicate across multiple data centers for fault tolerance

12-Factor App Principles

Methodology for building modern, scalable, maintainable cloud-native applications.

5. Continuous Integration & Continuous Delivery

Testing Types

- **Unit Testing:** Testing all behaviors of a single component. Checking isolated functionalities.
- **Integration Testing:** Checking components together. Actual API calls checked here.
- **System Testing:** Everything together!
- **Mocking:** Mock an API internally and return hard-coded values. Good for testing when we don't want unnecessary API requests. Can mock successful/failure errors. Mainly for unit tests.

CI/CD Pipeline

- Automating infrastructure - needed for Microservices. Automatically QA & Prod!
- Integrate your functionality as soon as possible
- To minimize cost, devs usually build a pipeline for CI/CD & testing
- **Pipeline:** Whenever we merge or commit new code → Test → Setting dependencies → Testing & compiling → Build. PR = Pull Request
- **Key Practice:** Merge small changes ASAP (don't let code diverge too much)

Deployment Strategies

- **Blue-Green Deployment:** Two identical environments - switch traffic instantly
- **Canary Deployment:** Roll out to small percentage first, then expand
- **Canary duration/size:** How long to wait? How big is canary?

6. Infrastructure Provisioning & Configuration Management

Terraform Basics

- Infrastructure as Code (IaC) tool
- Declarative configuration - describe desired state

Key Commands

- **terraform init:** Initialize working directory, download providers
- **terraform plan:** Preview changes before applying
- **terraform apply:** Apply the changes to create/update infrastructure
- **terraform destroy:** Remove all managed infrastructure

Key Concepts

- **Providers:** Plugins for cloud platforms (AWS, GCP, Azure)
- **Resources:** Infrastructure components to create
- **State:** Terraform tracks current infrastructure state
- **Modules:** Reusable configuration blocks

7. Site Reliability Engineering (SRE)

What is SRE?

- Traditional ops: Disconnect between product ("launch anything anytime") and ops ("never change anything")
- Google's approach: Hire software engineers to run products, create self-running/repairing systems
- **50% Rule:** Google SRE teams have max 50% cap on "ops" work - rest must be development
- Goal: Eventually 50% ops time should decrease through automation

DevOps vs SRE

- **DevOps:** Collaboration between dev and ops for efficient software delivery
- **SRE = DevOps + Google Extensions:** SLOs, SLIs, SLAs, Error budgets, Blameless postmortems

Observability

- **DynaTrace:** Visibility checks, state of our system. To make graphs & stuff.
- **Prometheus:** Open-source monitoring system
- **System Observability:** Observing our system through Graphs, Histograms, etc.
- We need our system to expose data - CPU utilization, memory, internal stuff, API statistics
- **Logging:** File-based logging, console-based logging → Format & Visualize

Alerting

- Alert through Kafka - make different alerts by writing queries filtering different errors
- What if DB down? Error Rate % goes up → start getting ALERTS
- We can have a threshold (2% Error Rate) & show in same Dashboard (DynaTrace)

Managing Risk and Availability

- 99.99% vs 99.999% may not be distinguishable to users
- Balance cost, innovation, features, performance with reliability
- **Risk Tolerance:** Acceptable level of service downtime

Availability Formulas

- **Time-based:** Availability = Uptime / (Uptime + Downtime)
- **Downtime calculation:** D = T × (1 - A) where T = 525,600 minutes/year
- **Request-based:** Availability = Successful Requests / Total Requests
- **Allowed errors:** D = T × (1 - A) where D = allowed errors, T = total requests
- **Example:** 100,000 requests at 99.99% = 10 allowed errors (100,000 × 0.0001 = 10)

IMPORTANT: Availability Table (The Nines)

- **90% (one nine):** 36.5 days/year, 3 days/month, 2.4 hours/day
- **99% (two nines):** 3.65 days/year, 7.2 hours/month, 14.4 minutes/day
- **99.9% (three nines):** 8.76 hours/year, 43.2 minutes/month, 1.44 minutes/day
- **99.99% (four nines):** 52.6 minutes/year, 4.32 minutes/month, 8.64 seconds/day
- **99.999% (five nines):** 5.26 minutes/year, 25.9 seconds/month, 0.87 seconds/day

Cost-Benefit Analysis Example

- Proposed improvement: 99.9% → 99.99% = 0.09% increase
- Service revenue: $1M, Value of improvement: $1M × 0.0009 = $900
- **Decision:** If cost < $900, worth it; if cost > $900, not worth it

SLI (Service Level Indicator)

- **Definition:** A measurement of a property of your service important to users
- **Latency:** How long to return response
- **Error Rate:** Ratio of errors to all requests
- **Throughput:** Requests processed per second
- **Durability:** Likelihood data stored over long periods (storage services)
- **Availability:** Fraction of time service is usable
- **Examples:** availability %, p95 latency, error rate, throughput, crash-free sessions

Percentiles for Latency

- Same request gives different response times → distribution of values
- Mean not useful due to high variety
- **Median (p50):** Better at telling average case - reflects typical user experience without distortion from extreme values
- **p95:** 5 out of 100 requests take more than this time
- **p99:** 1 out of 100 requests take more than this time
- **Amazon uses p99.9 (1 in 1000):** Slowest customers = most valuable (largest accounts, most purchases)
- p99.99 (1 in 10000) too expensive and unpredictable
- Average can obscure tail latencies - typical request 50ms but 5% are 20x slower

SLO (Service Level Objective)

- **Definition:** A target for an SLI over a time window. The engineering goal.
- **Example:** "99% (averaged over 1 minute) of Get RPC calls will complete in less than 100ms"
- **Example:** "99.9% of requests must succeed over 30 days"
- SLI ≤ target SLO
- Unrealistic to insist SLOs met 100% of time - reduces innovation rate

Error Budgets

- Conflict between product team (launch features) and SRE (stability)
- Software fault tolerance: Too little = brittle product, too much = stable but unused
- Testing: Not enough = outages/data leaks, too much = lose market
- Push frequency: Every push is risky

Error Budget Solution

- Product Management defines SLO (sets uptime expectation per quarter)
- Actual uptime measured by monitoring system
- Difference = "budget" of unreliability remaining for quarter
- **Rule:** As long as uptime > SLO (error budget remains), new releases can be pushed

SLA (Service Level Agreement)

- **Definition:** Explicit or implicit contract with users including consequences of missing SLOs
- Consequences: Financial rebates/penalties
- **Easy distinction:** "What happens if SLOs aren't met?" - if no consequence, it's an SLO
- Crafting SLA requires business and legal teams
- SRE helps understand likelihood and difficulty of meeting SLOs
- **Be conservative:** Broader constituency = harder to change/delete unwise SLAs

8. Containers

Container Benefits

- **Self-contained:** Everything needed, no host dependencies
- **Isolated:** Minimal influence on host/other containers, increased security
- **Independent:** Deleting one doesn't affect others
- **Portable:** Run anywhere - dev machine, data center, cloud

Containers vs VMs

- **VM:** Entire OS with own kernel, hardware drivers, programs, applications
- Spinning up VM just to isolate single application = lot of overhead
- **Container:** Isolated process with all files it needs to run
- **Key Advantage:** Multiple containers share same kernel → run more applications on less infrastructure

Container Internals

CGroups (Control Groups)

Limit and monitor resource usage:

- Memory limits (prevent excessive memory usage)
- CPU quotas (restrict CPU time per process)
- I/O bandwidth (limit disk access)

Namespaces

Isolation for system resources:

- **PID namespace:** Isolates process IDs - processes inside container can't see/interact with processes outside
- **Mount namespace:** Each container has own file system view, preventing access to files outside scope

Copy-on-Write Storage

- Only copy data when modifications occur
- Reduces disk space usage and improves performance
- Technologies: Snapshotting FS, AUFS, ZFS

Docker Alternatives

- **Important:** Docker ≠ Containers
- **Alternatives:** Podman, CRI-O, Kaniko, runC
- **OCI (Open Container Initiative):** Standard image format

9. Event-Driven Architectures

Messaging Basics

- High-speed, asynchronous, program-to-program communication with reliable delivery
- **Messages:** Packets of data
- **Queues/Channels:** Logical pathways connecting programs
- **Sender/Producer:** Program sending message
- **Receiver/Consumer:** Program receiving message
- **Providers:** Amazon SQS, RabbitMQ, NATS, Redis
- **Kafka:** Event-streaming platform for real-time data pipelines and stream processing. Microservices subscribe to a topic which lets them know when data is received (event happened). Commit-less: all data is immutable.

Usage Examples

- Integrations between different systems
- Chat Apps
- Background tasks: image compression, PDF generation, email sending
- Ride Booking Apps
- Sensor Data from IoT devices

Messaging Patterns

- **Simple:** One consumer, one producer
- **Load Balancing:** Message delivered to only ONE of subscribed consumers
- **Fanout:** Message delivered to ALL subscribed consumers
- **Filter:** Subscribe to specific topics or subjects

Wildcards (NATS Syntax)

- **a.b.*** matches a.b.c, a.b.d, etc.
- **a.>** matches a.anything.anything...
- **a.*.b** matches a.x.b, a.y.b, etc.

Acks and DLQs

- All messages delivered are acknowledged by clients or get retried
- **DLQ (Dead Letter Queue):** If messages not acknowledged after configured retries → put into DLQ. All data that didn't get deserialized properly (order didn't get processed correctly).
- Special consumer decides what to do with failed messages
- **Important:** Because of retries, queues cannot guarantee exactly-once semantics
- Consumers must consider possibility of receiving message more than once
- **Serialization:** A two-way process b/w two systems to understand everything
- Individual orders go in DLQs. That order does not necessarily get re-processed.

Streaming vs Messaging

- **Streaming:** Message queue + ability to "replay" all messages
- Store data as continuous records in distributed log
- Consumers subscribe based on subjects/topics, process at their own pace
- Allows rewind, forward, restart message consumption
- Used as buffer for slow consumers
- Since streams persist, can be used as database or primary source of truth
- Other systems can build/rebuild view by reading stream
- Every record = "Event" → Event Driven Systems

Distributed Log Architecture

- Append-only log with built-in partitioning and replication based on topics
- All patterns (fan-out, filtering, wildcards) work for stream messages
- Consumers save sequence numbers to resume after errors, do batch reads, jump ahead/back
- Background jobs to archive or delete old records if configured
- **Streaming Providers:** Kafka, NATS Jetstream

10. Data Systems - Storage and Retrieval

Functions of Data System

- Store data, Retrieve when we want that data
- Need to decide which database to use based on workload
- **Analytical workloads:** NoSQL, Column Oriented
- **Transactional workloads:** RDBMS
- Within RDBMS: different storage engines (Log Structured vs Page Oriented)

Simple Database Concept

- Append-only logs very efficient for writes
- Search is inefficient → O(N)

Indexes

- Efficiently find values for particular key
- Additional data structure → needs extra storage
- One for each key
- **Trade-off:** Speed up reads but every index slows down writes (index on company name = size reduced, faster access)
- **Drawback:** Write would involve re-indexing (slow). Solution: Threading
- **Types:** Hash Indexes, SSTables, LSM Trees, B-Tree

Hash Indexes

- In-memory map of byte offsets on disk
- **Limitations:** Cannot put on disk, must fit in memory
- Range queries do not perform well (cannot find keys between kitty0000 and kitty9999)

SSTables (Sorted String Tables)

- Sorted logs based on keys
- Memtable based on sorted tree (e.g., red-black tree) maintained in-memory
- Periodically flushed to disk
- Read first looked up in memtable, then on disk
- Efficient for on-disk storage and range scans
- **LSM Tree (Log-Structured Merge Tree):** Algorithm for merging and compacting SSTables

B-Tree

- Most common and default indexing structure on many RDBMS
- Sorts logs like SSTables but breaks database into fixed-sized pages (4KB usually)
- Each page can refer to other pages, follow page to page until find data
- Highly optimized for on-disk storage
- **Branching factor:** Number of references to other pages in a page (hundreds in practice, ~500)
- **Capacity:** Four-level tree of 4KB pages with branching factor of 500 can store up to 256 TB

KEY COMPARISON

- **SSTables/LSM Trees: Faster for WRITES**
- **B-Tree: Faster for READS**
- **Other index types:** Fulltext, GIN, Multi Column

11. Replication and Partitioning

Why Distribute Database?

- **Scalability:** Data, read, or write load bigger than single machine
- **Fault Tolerance:** Continue serving if machines go down
- **Latency:** Improve latency by placing data close to users
- **NOT for:** Reducing replication lag

Replication vs Partitioning

- **Replication:** Keeping copy of same data on several different nodes
- **Partitioning (Sharding):** Splitting big database into smaller subsets (partitions) assigned to different nodes

Single-Leader Replication

- One replica = master/primary/leader (all writes go here)
- Other replicas = followers/standby/slaves
- Leader sends writes to replicas as log, replicas update their copy
- Reads can go to any replica, writes only to leader
- **Built-in:** MySQL, PostgreSQL, SQL Server, MongoDB, RethinkDB, Espresso, Kafka, RabbitMQ

Handling Node Outages

- **New follower:** Setup using latest snapshot, fetch all records since last replication
- **Follower disconnect:** Request replication logs since last known sequence, catch up
- **Leader failover (Difficult):** How to know leader failed? Choose new leader (elections, consensus). Tell followers. Deal with data loss, old leader returning, split brain.
- **What to do when replica just got created:** 1) Run all queries 2) Copy change logs (diffs) 3) Maintain snapshots of master's state at intervals then run those on new replica
- **Failover:** If master down, replica becomes master. Offset can be maintained if replica is down but master is still fine.
- **Priority List:** Leader election uses priority list
- **Split Brain Problem:** 2 leaders of DB - resolved by Coordinator/Zookeeper with Consensus Algorithms (Raft)

Replication Log Implementation

- **Statement-based:** SQL statements sent. Problem: Nondeterministic functions (NOW(), RANDOM()) generate different data. Autoincrements may be inconsistent.
- **Write-Ahead Log (WAL) shipping:** Use database log to reconstruct state on followers. Very accurate but very low level, requires same database version on all nodes.
- **Logical (row-based) log replication:** Separate log for replication, decouples storage engine. For each row: new column values saved. Good for replicating to external systems.

Read-After-Write Consistency

- Problem: User views data shortly after write, new data may not have reached replica yet
- **Solution:** User always sees their own updates. If few things editable by user, read those from leader. Track replication lag, only read from caught-up replicas.

Multi-Leader Replication

- Alternative when leader might go down
- Multiple leaders, both accept writes, replicate to separate followers
- Usually one leader per datacenter
- Leader in one datacenter acts as follower of leader in other

- **Not very common:** Should be avoided if possible
- **Special case:** Offline editing (Google Drive) = multi-leader replication

Write Conflicts

- Occur when two leaders concurrently update the same record

Conflict Resolution

- **Conflict Avoidance:** Always send same user's writes to same leader
- **Last Write Wins:** Most recent timestamp wins
- **Precedence Numbers:** Give priority based on node ranking (higher precedence wins)
- **Record conflict:** Save all information, deal with later
- **Use Code - On write:** Code logic resolves conflict as soon as detected
- **Use Code - On read:** Multiple records returned, client determines right version, writes winner back
- Automatic Resolution of conflicts

Leaderless Replication

- Inspired by Amazon's Dynamo system (NOT DynamoDB)
- **Used by:** Riak, Cassandra
- Reads and Writes rely on Quorum
- **IMPORTANT FORMULA: $w + r > n$**
- **Example:** $n=3$ nodes, $w=2$ (write to 2), $r=2$ (read from 2) → $2+2=4 > 3$ ✓
- **If $r=1$ and $w=1$ in 3-node system:** Lower latency but stale reads likely ($1+1=2$ is NOT > 3)

Partitioning (Sharding)

- **When needed:** Data becomes too large for single machine
- Decide which partition a record belongs to using rules
- Query on single partition → handled by single node
- Large complex queries spanning multiple partitions → run in parallel

Skew and Hot Spots

- Goal: Partition records evenly
- **Skewed partitioning:** Unfair - more data on some partitions than others
- **Hot Spot:** Single or few partitions receive disproportionate load

Partitioning Strategies

- **By Key Range:** Based on minimum and maximum key value. May need to adjust over time.
- **By Hash:** Hash of key distributes data more evenly, solves hot spot problem for related data

Quick Reference Summary

Key Formulas

- **Availability (time):** Uptime / (Uptime + Downtime)
- **Availability (requests):** Successful Requests / Total Requests
- **Allowed Downtime:** $D = T \times (1 - A)$
- **Quorum Rule:** $w + r > n$

Key Distinctions

- **SLI vs SLO vs SLA:** Measurement vs Target vs Contract with consequences
- **LSM Tree vs B-Tree:** Faster writes vs Faster reads
- **Replication vs Partitioning:** Same data on multiple nodes vs Different data on different nodes
- **REST vs RPC:** Resource-oriented vs Action/Method-oriented
- **Container vs VM:** Shared kernel vs Full OS
- **Messaging vs Streaming:** Message consumed once vs Can replay messages