Exam Topics

Exam topics:

- API Fundamentals (OpenAPI, gRPC, GraphQL, Circuit Breakers, Secrets Management)
- Architectural Styles: Monoliths, Microservices, Modular, Case Studies
- Scalability Techniques (CDNs, GeoDNS, Multi-AZ, Event-Driven)
- Continuous Integration & Continuous Delivery
- Infrastructure Provisioning & Config Management
- Data Systems Overview, CAP Theorem
- Scaling RDBMS & NoSQL Systems
- Back-of-the-Envelope Calculations, Latency Numbers, QPS

Exam Notes

Introduction

- Legacy code is code that has become old, brittle and difficult to change
- Over time code naturally ages and deteriorates according to evolving requirements
- Code becomes rotten because customers requirements always change, and software that isnt designed for flexibility struggles to adapt these changes
- Characteristics of Legacy Code
    - Small modifications can break unrelated parts of system
    - Lack of modularity, poor testing
    - Messy and difficult to maintain
- To prevent code from rotting, software developers must design software, that
    - Follow good coding practices, clean architecture, modular design, and regular refactoring
    - Ensure system can evolve smoothly with changing requirements
- Readable code is easier to maintain, debug and extend
- Descriptive names for variable, methods etc
- Proper indentation, spacing formatting
- Extensibility = ability of a software system to grow and adapt by adding new features or components without breaking or rewriting existing code
- How to design for extensibility?
    - Open/Closed principle: code should be open for extension but closed for modification
    - Interfaces/Abstractions
- Adding new methods means adding new sub classes
- Four Pillars of OOP
    - Polymorphism: same function/method name but behaves differently
    - Abstraction: focus on what an object does not how
    - Encapsulation: group related data or methods
    - Inheritance: create new classes based on existing ones, reusing and extending code
- S-Single responsibility: each class/module should have single responsibility
- O-Open/Closed principle: classes should be open for extension but not for modification
- L-Liskov Substituion: subclasses should be substitutable for their parent class without altering correctness
- I-Interface Segregation: Prefer many small, specific interfaces over one large , general purpose one
- D-Dependency Inversion: depend on abstractions, not concrete implementations
- DRY vs WET

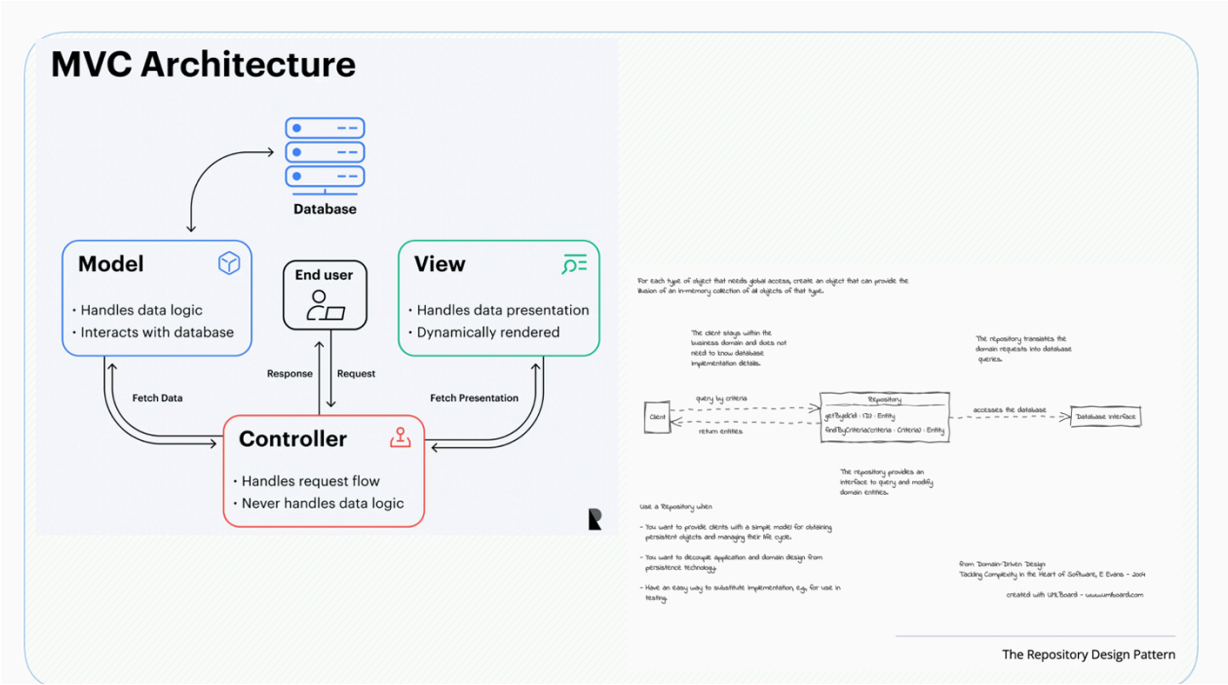
    - DRY (Dont Repeat Yourself): dont copy paste code again and again instead write it once and reuse it again
    - WET (Write Everything Twice): sometimes it's ok to write things twice
- KISS vs YAGINI
    - KISS - Keep it simple, stupid : simple designs are easier to understand
    - YAGINI - You arent gonna need it: dont build for hypothetical requirements only implement current features
- Code Smells
    - Indicators that code might need refactoring
    - Not always a bug but maybe indicate a design issue
    - Common examples
        - Duplicate codes
        - Improper names
        - Dead code
        - Long functions
        - Excessive comments
- Refactoring
    - Improving internal structure of code without changing external behaviour
- Testing
    - Validation vs Verification
    - Levels of testing
        - Unit testing: test small pieces of code
        - Component testing: test combined modules
        - System testing: test whole app
- Version control
    - System that tracks changes in code/documents over time
    - Central repository
    - You can see who changed, what got changed, when and why

Architectural Styles: Monoliths, Microservices, Modular, Case Studies

Building Complex Applications

- Building complex layers require understanding of multiple layers of complexity
- Tackling complexity (breaking down layers)
    - Layered architecture: a design pattern where modules or components with similar functionalities are organized into horizontal layers
        - Presentation layer
            - Contains all classes responsible for presenting UI/visualization to end user and handling browser communication logic

- Business layer
    - Contains core business logic and processing rules of application rules of application. This layer usually deals with data aggregation, computation and query requests.
    - Has main logic of application
- Persistence layer
    - This layer manages data storage and retrieval. This layer consists of both logical and physical aspects.
- Database layer:
    - This is where all applications resides and is managed



- A software architecture explicitly defines:
    - The components that make up system
    - How these components interact with each other
    - How control and communication are managed between these components
- Monolithic architecture
    - A software application built as single unit
    - Characterized by single codebase where all components are tightly coupled
    - The entire application is packaged and deployed as one indivisible entry
    - Components of monolithic application
        - Client server UI interface
        - Server side application
            - A single logical executable that handles all server processing

                - Has access to and manages server resources
                - Handles incoming HTTP requests
                - Executing core business/domain logic
                - Retrieving and updating from database
            - DBMS
        - Advantages:
            - Easier application development -> having single codebase simplifies initial coding and building process
            - Simple deployment -> only one directory to manage
            - Hassle free debugging & testing -> end to end testing easier as all components are together
            - Increased security -> architecture is closed system -> fewer exposed surfaces
        - Disadvantages
            - Complex reasoning -> difficult to pin point issues as all components are interconnected
            - Tight coupling -> a change in one module can have unexpected effects on other modules
            - Scaling difficulties -> inefficient scaling -> you cant scale high demand function you will have to scale entire application
            - Slower development speed -> larger codebase increases cognitive load
            - Reliability risk -> a single error in any module may bring down entire application
- Microservices
    - Architectural style that structures an application as a suite of small, independent services
    - Each service is built around specific functionality and can be developed, deployed and scaled independently
    - Characteristics
        - Services operates independently failure in one service does not necessarily crash the entire system
        - Each service runs its own process
        - Different services can be written in different programming languages
        - Each service can be released and versioned independently
        - Services can be replicated and granulated
    - Advantages
        - Enables teams to develop, debug, deploy their services independently

        - Specific services can be scaled horizontally to meet demand
            - Vertical vs Horizontal (simple)
            - Vertical scaling: Make one server more powerful (more RAM, CPU).
            - Horizontal scaling: Add more servers and distribute load.
            - Horizontal scaling is preferred in cloud systems because it:
                - increases fault tolerance
                - is cheaper
                - can auto-scale
                - avoids single-server limits
        - Allows teams to choose tech stack according to their services
        - A failure in one service is less likely to effect others
    - Disadvantages
        - The system as whole gets distributed, which is inherently more complex
        - A larger attack surface due to network communication
        - Increased overhead for operating multiple databases
- Convoys Law
    - The structure of software you build will inevitably mirror the structure of teams that build it
    - What it means:
        - Siloed teams (UI, Backend, DB) → produce layered monoliths.
        - Cross-functional teams (Orders team, Users team) → produce microservices
- Microservices Components
    - Backend services
    - API gateway
    - Service to service communication
    - Database per service
    - Load balancer
    - Independent deployment pipelines
    - Containerization
- Twelve Factor App Methodology
    - A methodology to build SaaS applications
    - Ensure apps are
        - Deployable on cloud
        - Continuously deliverable
        - Scalable without structural change
        - Maintainable via declarative automation
        - The 12 Factors

        - Codebase
            - One codebase tracked in version control, many deploys.
        - Dependencies
            - Explicitly declare dependencies; no implicit system-level installs.
        - Config
            - Configuration stored in environment variables, not in code.
        - Backing Services
            - Treat DB, cache, queues as attached resources; interchangeable.
        - Build, Release, Run
            - Strict separation of build stage → release stage → runtime.
        - Processes
            - Application is stateless; scale out by running multiple processes.
        - Port Binding
            - App self-contains web server; exposed via port binding.
        - Concurrency
            - Scale horizontally through process model, not threads.
        - Disposability
            - Fast startup and graceful shutdown to support scaling + resilience.
        - Dev/Prod Parity
            - Keep development, staging, and production environments as similar as possible.
        - Logs
            - Logs are event streams; don't manage log files inside the app.
        - Admin Processes
            - Run admin/maintenance tasks as one-off processes.
- Modular architecture
    - A middle ground between monolith and microservices
    - Application is still one deployable unit but internal structure is divided into independent modules
    - Each module is a self contained unit with
        - Its own domain logic
        - Clear interfaces
        - No direct database access outside its own boundaries
    - Communication happens through APIs
- Case Studies (typical examples to mention in exams)

- Case Study 1: Amazon (Monolith → Microservices)
    - Amazon started as a big monolithic codebase.
    - As traffic grew, scaling and deployments became slow.
    - They moved to a microservices model:
    - “Hundreds of small teams, each owning a service”
    - Independent deployments
    - Scaled individual services (e.g., search, recommendations)

  Lesson: breaking monolith improved scaling and team velocity.
- Case Study 2: Netflix
    - Originally ran on a monolithic app in data centers.
    - Outages and scaling issues pushed them to migrate to AWS.
    - Adopted microservices + cloud-native practices:
    - API Gateway (Netflix Zuul)
    - Circuit Breakers (Hystrix)
    - Chaos Engineering
    - Auto-scaling

  Lesson: microservices helped achieve global reliability + performance.
- Case Study 3: Careem
    - Started with a Python/Django monolithic backend.
    - Scaling challenges grew as they expanded across MENA.
    - Migrated toward microservices with domain-based services like Booking, Dispatch, Pricing, Payments, and Captain Service.
    - Adopted independent deployments, service isolation, and horizontal scaling of heavy-load services (e.g., Dispatch).

  Lesson: microservices improved reliability, faster releases, and region-wide scalability.

Load Balancing

- Load balancing distributes incoming traffic across multiple instances of service so no single service is overwhelmed
- Why it’s important?
    - All servers get balanced work
    - Failed servers are skipped
    - Requests sent to best performing server
    - System stays alive even if one node crashes
    - Easy to add more servers horizontally
- Static Load Balancing (fixed rules, no real time awareness)
    - Static algorithms do not consider server health, load or performance

    - Round Robbin
        - Requests go to servers in rotation
        - Very fast
        - Doesnt consider server health or workload
    - Weighted round robbin
        - Each server is given a weight based on capacity
        - Example
            - A: weight 3
            - B: weight 2
            - C: weight 1
        - Higher-weight servers receive more requests.
    - IP Hash (Consistent Hashing)
        - Client IP or key → hashed → mapped to a specific server.
        - Ensures the same client usually hits the same server.
- Dynamic Loading
    - Dynamic algorithms react to current server state
    - They consider
        - Active connections
        - CPU/memory
        - Response time
        - Health checks

      This gives smarter routing

        - Least connections
            - Traffic sent to server with fewest open connections
        - Weighted less connections
            - Mix of capacity + connections
                - Each server has weight (capacity)
                - Among the servers with lowest connections, choose the one with highest weight
        - Weighted response time
            - Load balancer uses each servers response time as weight
            - The fastest responding server gets more requests
        - Resource based load balancing
            - LB chooses server with most free resources
- Health checks
        - Load balancer constantly checks server health
        - If a server fails remove it from rotation automatically
        - When server recovers add it back

SERVICE REGISTRY

- A service registry is a database that stores network location (IP+Port) of all running service instances in microservices system
- It enables service discovery so services can find each other dynamically.
- Why do we need it?
    - In microservices
        - Each service instance gets a dynamic IP/port (due to autoscaling, failures, redeployments).
        - New instances appear, old ones die.
        - Clients cannot hardcode IPs.
    - So a registry keeps track of:
        - which instances exist
        - where they are located
        - which ones are healthy
- How service discovery works (simple flow)
    - Service instance starts → registers itself with the registry (IP + port).
    - Registry stores instance in its database.
    - Client asks registry, “Where is Service A?”
    - Registry returns list of available instances.
    - Client or load balancer chooses one instance and sends the request.
    - When a service stops, registry removes it.
    - Instances periodically send heartbeat to stay registered.
- Client-Side Discovery
    - Flow
        - Client → queries registry
        - Registry returns addresses of instances
        - Client picks one (round robin, hash, etc.)
        - Client calls that service directly
    - Benefits
        - Flexible load balancing per client
        - Client knows all instances → can do smart routing
    - Drawback
        - Tight coupling: client must include registry logic
        - Must implement discovery logic for each language/framework
        - More complex client code
- Server-Side Discovery
    - Load balancer handles discovery.
    - Flow
        - Client → Load Balancer
        - Load balancer → queries registry to find instances

        - LB sends request to one of the instances
        - Instances still register/deregister with the registry
    - Benefits
        - Clients are simple (just call LB)
        - Centralized discovery logic
        - Common in cloud/Kubernetes
    - Drawback
        - Load balancer + registry must be highly available
        - Another system component to manage
- How Services Register
- Two standard patterns:
- 1. Self-Registration Pattern
    - Each service instance:
    - Registers itself (POST)
    - Sends heartbeat (PUT)
    - Deregisters on shutdown (DELETE)
    - Advantages
        - Simple
        - No extra components
    - Drawback
        - Services are tightly coupled to registry logic
        - Must implement registration in every language used
- 2. Third-Party Registration Pattern
    - A separate service registrar handles registration.
    - Registrar monitors running instances via:
        - polling (asking deployment system for instance list)
        - events (like in Kubernetes)
        - health checks
    - Registrar then:
        - registers instances
        - removes dead ones
    - Advantages
        - Services are decoupled
        - No registration logic inside service code
    - Drawback
        - Requires an extra highly available component
- Heartbeat Mechanism
    - Every service instance must periodically send a heartbeat:
        - “I’m alive”
        - Updates TTL (time-to-live)

    - If heartbeat stops → registry marks it dead → removes it.
- Service Registry Needs to Be:
    - Highly available
    - Up-to-date (fast registration + deregistration)
    - Distributed (to avoid downtime)

Scalability Techniques

- CDNs (Content Delivery Networks)
    - A CDN is a **network of edge servers** distributed worldwide that **cache and deliver content from the location closest to the user**.
    - Why CDN?
        - Reduce latency (closer server → faster response)
        - Reduce load on the origin server
        - Reduce bandwidth usage
        - Improve security
    - How it works?
        - User requests static asset
        - CDN edge node closest to user serves it
        - If not cached -> CDN pulls from origin -> caches it for next time
- GeoDNS
    - DNS returns different IPs based on user geographic location
    - How GeoDNS works?
        - Client requests DNS resolution (eg google.com)
        - DNS server checks client location
        - Returns IP of regions nearest data center
    - Why GeoDNS?
        - Sends user to nearest region -> low latency
        - Helps load balancing across regions
        - Enables region specific deployment
    - Example: using VPN → IP changes → DNS returns different region
- Anycast routing
    - Multiple servers share same IP address
    - How it works?
        - All edge servers broadcast same IP address
        - BGP chooses the nearest or low hop server
        - User automatically reaches that server
    - Why Anycast?
        - Faster routing globally
        - Automatic failover

        - Used by cloudfare, Google DNS
- CDN + GeoDNS + Anycast Together (Important)
    - Modern CDNs use GeoDNS to route the user to nearest region and Anycast to route within that region to nearest edge server.
    - Combo benefits
        - GeoDNS -> picks nearest region
        - Anycast -> picks nearest server inside that region
        - Super low latency + high reliability
- Availability zones and Multi-AZ Deployment
    - AZ = physically separate data centers within the same region.
    - Multiple data centres
    - Why multi-az?
        - Protects against power/network failure in one AZ
        - High availability
        - Fast failover
- Regions and Multi-region deployment
    - Regions = geographically distant data centers (e.g., Singapore, Virginia).
    - Why multi region?
        - Disaster recovery
        - Compliance requirements
        - Lower latency for global users
    - Challenges
        - Higher latency between regions
        - Expensive
        - Complex architectures
- Event driven architecture
    - How it works?
        - Instead of synchronous requests -> system use events + queues
        - Produces emit events
        - Consumers process events asynchronously
        - Systems uses auto-scale based on event load
    - Why it improves scalability?
        - Decouples services
        - Buffers spikes using queues
        - Auto scales consumer
        - Prevents overload on downstream services
    - Kafka lets user send and receive data in real time

API Fundamentals (OpenAPI, gRPC, GraphQL, Circuit Breakers, Secrets Management)

- OpenAPI is machine readable specification (YAML/JSON) describing REST endpoints
- Defines paths, parameters, request/response formats, error codes
- Enables contract first development
- REST APIs rely on JSON → human-readable but not strictly typed.
- OpenAPI solves inconsistency by enforcing a **standard contract** between client and server
- Key Concepts
    - Paths → endpoint URIs (/users, /orders/{id})
    - Methods → GET/POST/PUT/DELETE
    - Schemas → typed request/response bodies
    - Parameters → query, path, header
    - Responses → status codes + schema
- Why It Matters
    - Auto-generated API documentation (Swagger UI)
    - Auto-generate client/server code
    - Validates requests
    - Ensures team-wide consistency
- gRPC (Remote Procedure Calls)
    - A high performance RPC framework built on HTTP/2
    - Uses Protocal Buffers (protobuf) for serialization
    - Autogenerate client and server stubs from .proto files
    - How it works?
        - Client calls a local stub
        - Stub serializes parameters into binary protobuf
        - Message sent over network
        - Server stub unmarshals request and invokes service function
    - Bidirectional streaming
    - Ideal for microservices
    - Reliability is a challenge in RPC.
    - Failures must be handled with semantics:
        - At-Least-Once → Retry until success, but risk duplicate actions.
        - At-Most-Once → Fail fast, risk losing the action.
        - Exactly-Once → Ideal but very hard to guarantee.
- GraphQL
    - A query language for APIs where clients specify exactly the fields they want
    - Benefits

        - Avoid over fetching
        - Avoid under fetching
        - Single endpoint
        - Strongly typed schema
    - Components
        - Query -> read
        - Mutation -> write
        - Resolver -> function that fetches actual data
        - Schema -> types and data fields
- Circuit Breakers
    - RPC failures like
        - Client crash
        - Server crash
        - Dropped packets
        - Slow network
    - Circuit breakers guard against these.
    - What circuit breakers do?
        - Prevent cascading failures by **stopping calls to an unhealthy service**.
        - States
            - Closed → system normal
            - Open → stop calls immediately, fail fast
            - Half-Open → test a few calls to see if service recovered
        - Purpose
            - Avoid timeouts
            - Improve resilience
            - Limit damage from failing dependencies
    - Circuit breakers are often implemented with libraries like Resilience4j.
    - Key parameters:
        - failureRateThreshold → e.g., open if 50% fail.
        - waitDurationInOpenState → how long to stay open before testing.
        - slidingWindowSize → number of past calls used to calculate failure rate.
- Secrets Management
    - Why its needed?
        - API needs authentication with
            - API Keys
            - Tokens
            - DB Credentials
            - SSH Keys

    - Goals
        - Protect secrets at rest and in transit
        - Rotate credentials easily
        - restrict access using policies
    - Where Secrets Are Stored (Best Practice)
        - AWS Secrets Manager
        - AWS Parameter Store
        - HashiCorp Vault
        - Azure Key Vault
        - GCP Secret Manager
        - Kubernetes Secrets

Continuous Integration & Continuous Delivery

- Continuous Integration is the practice where developers merge their code into a shared main branch frequently (usually daily).
- Every merge automatically triggers:
    - a build
    - automated tests
- This helps detect integration issues early.
- Why CI is needed?
    - When developers work in isolation for so long
        - Team mates can change same files -> merge conflicts
        - Tests changes -> your local code becomes outdated
        - Bugs are caught late on staging/proudction
- CI Workflow
    - Developer writes code in local environment
    - Meanwhile, teammates keep updating central repository
    - Developer pulls updates and integrates them locally
    - Developer rebuilds, and reruns tests locally
    - Developer pushes code to repo
    - CI server builds and tests the combined code
    - If successful notify developers, if not fix immediately
- Best CI practices
    - Put everything in version control
        - Always commit to main branch
        - Use feature branches delete them after merge
        - Prefer text based files
    - Automate the build
        - One command should build whole system

        - No manual steps in build process
    - Make build self testing
        - Automated tests must run on every build
        - Green build = all tests passed
        - Red build = broken functionality
    - Everyone commits daily
        - Frequent commits help detect early conflict
        - Each push should handle automated build and test
- Linters: Tools that check for code style issues and poor programming practices automatically
- Continuous delivery extends CI to ensure software is:
    - Always deployable
    - Tested for production readiness
    - Packaged automatically
    - Deployment is not always automatic
    - Cycle
        - Build software - code with deployability in mind
        - Deployable software - every built software is deployable
        - Prioritize deployability
        - Automated feedback - CI system provides feedback about failures
        - Push button deployment - any version can be deployed to any environment easily
    - Zero downtime deployment
        - Blue Green deployment
            - Two identical environments blue and green
            - Users are on green -> new version deployed to blue
            - Switch router instantly to blue
            - Keep green for rollback if something fails
        - Canary release
            - Release new version to small % of users first
            - Detect bugs without affecting all users
            - If stable -> roll out to more users gradually
    - Continuous deployment = every change that passes automated tests is automatically deployed to production

Infrastructure Provisioning & Config Management

- Infrastructure provisioning is process of creating computing resources automatically using code instead of manually
- This follows IaC

    - Infrastructure is defined using files (YAML, JSON etc)
    - Stored in version control
    - Fully reproducible
    - Enables automation in CI/CD pipeline
- Why IaC?
    - Avoid manual configuration -> prevent mistakes
    - Avoid environment drift
    - Create stable test environments repeatedly
    - Provision and destroy infra on demand using cloud APIs
- Key benefits
    - Consistency across environments
    - Faster provisioning
    - Scalability on cloud platforms
    - Infrastructure becomes part of software delivery
- What it does?
    - Avoid manual configuration
        - No editing configs manually
        - Ensures prod, staging, dev environments **look the same**.
    - Deliver stable test environments
        - Cloud dynamically creates environments from IaC definitions.
    - Automate across any cloud
        - Same IaC can work on AWS, Azure, GCP, VMWare
    - Use declarative definitions
        - You describe desired state
        - Tools figure out how to apply
- Tools:
    - Terraform
        - Most widely used
        - Uses HCL files
        - Cloud and on prem support
        - Reusable, modular, version controlled
    - AWS Cloud Formation
        - Native to cloud providers
        - JSON/YAML templates define resource graphs
- Configuration management automates installation, setup, updates and configuration of already provisional resources
- Provisioning = creates servers
- Configuration management = configures them after creation
- **Traditional vs Modern**
    - Old: custom bash scripts

        - Modern: **Ansible, Chef, Puppet**, cloud-init, Packer images.
- Challenges
    - Configuration drift
        - Environments diverge because manual tweaks, hotfixes, inconsistent updates
        - Solution → IaC + automation ensures every environment matches production.
    - Independencies
        - Microservices have many distributed settings
        - Changing one service affects the other
    - Data management complexity
        - Configurations rely on shared state
        - Hard to maintain consistency across environments
- Modes of configuration
    - Agentless
        - Controller connects to servers via SSH
        - Applies changes and exits
        - Example -> Ansible
        - No agents installed → easy, lightweight.
    - Agent based
        - Agents run on each managed server
        - Always ensure correct configuration
        - Examples -> Puppet, chef
        - Continuos monitoring and enforcement
    - Image building
        - Preconfigure everything in a machine image
        - When VM/container starts its already configured
        - Example -> Packer

Site Reliability Engineering (SLOs, SLIs, SLAs)

- Why SRE?
    - Traditional orgs had separate ops teams managing deployments manually
    - Google noticed a disconnect
        - Product teams -> want rapid release
        - Ops teams -> wants stability
    - SRE solves this by using software engineers to run production systems
- What SRE engineers do?
    - Automate tasks normally done manually
    - Build systems that are
        - Self healing

        - Self managing
        - Highly observable
- SRE vs DevOps
    - DevOps -> faster delivery, collaboration
    - SRE -> Googles practical implementation of DevOps with
        - SLIs
        - SLOs
        - SLAs
        - Error budgets
- SRE = DevOps + Mathematics + Reliability Engineering
- Managing reliability and risks
    - Increasing reliability costs money
    - Must balance:
        - Innovation speed
        - Features
        - Cost
        - User expectations
    - **Most measurable risk metric = unplanned downtime**
- Downtime=Total Time (T)×(1−Availability A)
- Availability=Total requests/Successful requests
- SLIs - service level indicators
        - Quantitative metrics that measure service quality.
        - Common SLI
            - Latency: response time
            - Error rates
            - Throughput
            - Durability
        - Latency must be described using **percentiles**, not just averages.
    - p50 (median): 50% requests faster than this.
    - p95: slowest 5% performance.
    - p99: slowest 1%.
    - p99.9: slowest 0.1%.
    - Used because averages hide tail latency.
- SLOs - service level objectives
    - A target for an SLI
    - E.g “99% of GET calls must complete in **<100ms**, measured over 1 minute.”
    - SLOs should not be 100%.
    - Why?
        - Stops innovation

        - Too costly
        - Unrealistic
- Error budgets
    - Error budget = **Allowed amount of unreliability** based on SLO.
    - Why use error budgets?
        - Balances reliability vs feature velocity
        - Objective way to decide:
        - Can we release new features?
        - Or must we freeze releases and stabilize?
- SLAs- service level agreements
    - Formal contract with users
    - Includes penalties
    - SLA contain SLOs + legal consequences
    - Difference from SLO:
        - SLO = internal engineering goal
        - SLA = external business contract
- Monitoring
    - Monitoring answers:
        - What happened?
        - Why did it happen?
        - What will happen?
    - Used for:
        - Long-term trends
        - Comparison between versions
        - Debugging
        - Alerting
        - Dashboards
- If you monitor only 4 metrics, choose these:
    - Latency - time to serve a request
    - Traffic - demand -> RPS
    - Errors - explicitly or implicit failures
    - Saturation - how full the system is
- Alerting principles
    - Alerts are urgent
    - Warnings
    - Only alert when
        - Actionable
        - Urgent
        - User visible

Data Systems Overview, Scaling RDBMS & NoSQL Systems

- Databases often need to scale beyond one server because of:
  - Scalability
    - Dataset too large for one machine
    - Write or read head too high for single CPU/disk/network
  - Fault tolerance
    - If one machine fails system must continue working
    - Replicas allow failover
  - Latency
    - Serving users in different regions
- Replications vs Partitioning
  - Replication = copy of same data stored on multiple nodes
  - Used for
    - Fault tolerance
    - Higher read throughput
    - Availability
  - Partitioning = data is split into smaller subsets and each subset goes to different nodes
  - Used for
    - Scaling read + write load
    - Handling extremely large dataset
  - Benefits:
    - Writes scale linearly
    - Reads scale
    - Storage capacity increases
    - Queries can run in parallel
- Synchronous Replication
  - The leader waits for followers to confirm the write before saying “success” to the client.
- Asynchronous Replication
  - The leader immediately responds “success,” and followers update later.
- Replication of small datasets
  - If all data fits on one machine, we choose among:
    - Single leader replication
    - Multi leader replication
    - Leaderless replication
- Single leader replication
  - How it works?
    - One node = leader
    - All writes go to the leader

        - Leader sends replication logs to followers
        - Followers update their copy
        - Reads can go to any node
    - Follower failure handling (replica)
        - New follower bootstraps using leader snapshot
        - Then replays replication logs to catch up
    - Leader failure handling (master)
        - Challenges
            - Detecting leader failure
            - Electing new leader
            - Updating followers to track new leader
            - Handling split-brain
- Types of replication logs
    - Statement based
        - Sends SQL statements
        - Issues: nondeterministic (NOW(), RAND()), auto-increments may differ
    - WAL Shipping (Write Ahead Log)
        - Replicate strong engine's log
        - Very accurate but low level
        - Requires identical DB version
    - Logical / Row based replication
        - Replication specific log
        - Records before after row values
        - Easier for cross system replication
- Read your own write consistency
    - Problem
        - User writes data → reads from a lagging replica → sees old data.
    - Solution
        - Read user editable data from master (leader)
        - Or track replication lag and route to caught up replica
        - Or use session consistent threads
- Multi leader replication
    - Used in
        - Multi datacenter servers
        - Offline capable apps
    - How it works?
        - Multiple leaders accept writes
        - Each leader replicates to others
        - Each leader has its own follower (replica)

    - Problems
        - Conflicting writes
        - Hard to implement
        - Should be avoided unless necessary
- Conflicts in multi leader replication
    - Send all writes from same user to same leader
    - But if leader fails -> conflict unavoidable
    - Methods:
        - Last write wins: use timestamps or version numbers
        - Leader with higher priority wins
        - Later reconcile manually or through business logic
        - Resolution in code
            - On write automatically picks correct version
            - On read return multiple versions
- Partitionining + Replica
    - Each partition can also be replicated
    - This gives:
        - Scalability + availability
        - But increases complexity
- Skew and Hot Spots
    - Skew uneven distribution of data across partitions
    - Hot spot a single or few partitions receive most traffic
    - Causes
        - Poor partitioning
        - Sequential IDs
        - Geographic concentration
    - Solutions
        - Harsh partitioning
            - Hash(key) → partition
                - Good for balanced distribution
                - Great for avoiding hot spots
                - Harder for range queries
        - Random partitioning
        - Better key design
        - Key-Range Partitioning
            - Partition by continuous ranges (e.g., A–F, G–M, N–Z)
            - Good for range scans
            - Bad for hot spots (e.g., timestamps)
- RDBMS vs NoSQL Scaling
    - Traditional RDBMS Scaling

        - Vertical scaling (bigger servers)
        - Add read replicas
        - Hard to scale writes
        - Strong consistency
    - NoSQL Scaling
        - Horizontal scaling
        - Partition-first architecture
        - Distributed writes
        - Eventual consistency
        - Good for massive datasets

Other Notes

- Redis is a **fast, in-memory NoSQL database** used to speed up applications by caching frequently accessed data.
- **Kafka is a system that lets different services send and receive huge streams of messages in real time.**
- Hash indexes are good for:
    - Exact key lookups
    - Checking if a key exists
    - Equality queries (WHERE email = 'x')
    - Because a hash function maps a key directly to a bucket.
- Hash indexes are BAD for:
    - Range queries (id > 100, price BETWEEN 10 AND 20, ordered scans)
- Why?
    - Because hashing destroys ordering — adjacent values do not hash to adjacent buckets.
- B-Trees use **ordered keys**; hash indexes are a separate data structure.
- B-Trees update pages **in place** (not append-only).
-

MCQs

We are implementing a URL shortening app. Which data store is most appropriate for a high-read URL redirection lookup?
Select all that apply.
RDBMS
Redis
Distributed File System
NoSQL Key Value Store

What is the primary goal of horizontal partitioning (sharding) in a database system?
Increase security by isolating user data
Improve write throughput and scalability
Ensure strong consistency across nodes
Eliminate the need for indexing

In a replicated system, which of the following statements are true?
Select all that apply.
Replication helps with high availability
Replication guarantees consistency across all nodes at all times
Replication can be synchronous or asynchronous
Replication eliminates the need for backups

In range-based sharding, what potential problem can occur with uneven data distribution?
Hotspots
Data corruption
Deadlocks
Stale reads

Partition tolerance is
ability to recover lost data when a disk is lost
waiting of requests while partitioning data
ability to recover from network failures
number of reads a partition can tolerate

Which of the following could reduce the load on a load balancer?
Message Queues
CDNs -> Clients hit the CDN instead of your load balancer
API Gateway

Out of the following, which is the most appropriate for alerting about downtime?
Email
Slack messages
Phone calls
Sms

Which of the following is one of the four golden signals of monitoring?
Throughput
CPU Load
Traffic
Disk I/O

If your 95th percentile latency is 800ms, what does that mean?
95% of requests took more than 800ms
95% of requests took less than or equal to 800ms
All requests took less than 800ms
The average latency is 800ms

Your service has an availability SLO of 99.9% uptime per 30-day month. How much total downtime is allowed in that month before the SLO is violated?
43.2 minutes
4 hours 22 minutes
7 hours 12 minutes
3 minutes

Downtime=Total Time (T)×(1−Availability A)

Downtime = (30x24x60) x (1-0.999) = 43.2

Which of the following best describes the relationship between SLI, SLO, and SLA?

SLI is a business contract, SLO is the measurement, SLA is the target

SLA is the actual measured metric, SLI is a goal, SLO is the legal commitment

SLI is the measured value, SLO is the target, SLA is the formal contract with consequences

They are all synonyms used in different industries

You define an SLO of 99.95% availability for an API endpoint. You measure availability using successful request responses (HTTP 200s) out of total requests. What is the SLI in this scenario?

99.95%

The total number of HTTP requests

The ratio of successful HTTP 200 responses to total requests

The agreement between you and the customer

Which of the following is a key difference between an SSTable and a B-Tree?

SSTables store data in a balanced tree structure

B-Trees support only append-only writes

SSTables require data to be sorted and written immutably

B-Trees use hash functions to index data

Explaination: This is the defining property of SSTables (Sorted String Tables):

Data is sorted

Files are immutable

Used in LSM-tree storage engines (Cassandra, RocksDB)

Which of the following operations is typically not efficient with a hash index?

Exact key lookups

Range queries (e.g., WHERE id > 100)

Checking existence of a specific key

Equality filtering (e.g., WHERE email = 'x')

You're building a leaderboard and need to fetch the top 100 scores in descending order. Which data structure is best suited for this?

Hash Index

SSTable

Bloom Filter

B-Tree

Which of the following is the most recommended way of replicating database changes in modern data systems?

Sending SQL Statements

WAL Shipping

Logical (row based) replication

Which of the following is solved by randomly hashing the key and assigning the partition based on the hash?

Skew

Hotspot

What does continuous integration solve?

Allows fellow developers to see the changes from others as soon as possible

Helps with running tests automatically

Runs integrations with other systems

Which branch should run end-to-end tests?

feature branch

release branch

main branch

Which of the following shows steps involves in continuous delivery?

Code -> Lint -> Unit Test -> Integration Test -> Publish Artifact

Code -> Lint -> Unit Test -> Integration Test -> Deploy

Code -> Lint -> Unit Test -> Integration Test -> Publish Artifact -> Deploy

Which of the following is impossible without monitoring in place?

Canary Deployment

Blue Green Deployment

Which one of the following is a way to give human readable name to a Docker image during build?

docker build --name nginx

docker build --label nginx

docker build --tag nginx

Which of the following Dockerfile instructions determine what gets executed when a container starts?

Select all that apply.

CMD

ENTRYPOINT

RUN

ARG

Which of the following a CDN relies on?

Select all that apply.

Anycast

Edge Load Balancers

GeoDNS

Edge functions

A load balancer is also a front proxy

True

False

GraphQL operates over plain text HTTP without requiring special protocols.

True

False

Which of the following can be rolled back quicker if something fails?

Blue Green

Canary

Which one of these is more expensive in terms of infrastructure cost?

Blue Green Deployments

Canary Releases

Continuous delivery is about delivering every change to production as soon as it passes all tests.

True

False

When doing continuous integration, developers run end-to-end integration tests on their systems before merging to the central repository.

True

False

CI is pre-requisite of CD

True

False

In addition to running containers, docker-compose can also build container images.

True

False

Which of the following best describes the difference between ARG and ENV in a Dockerfile?

ARG values are only available during build time, while ENV sets container environment variables

ARG is for system variables, while ENV is for application variables

ARG is permanent, while ENV can be changed at runtime

There is no difference between them

Load balancer is also a reverse proxy since it always has to proxy the request between backend and the user, hence hiding the backend from the user like a proxy.

True

False

We have a mix of old and new servers. Old servers have less capacity than the new servers. You are tasked with architecting load balancer in front of these servers.

On line 1, write static load balancing or dynamic load balancing depending on what is appropriate.

On line 2, write in a single sentence the algorithm we may use for this setup. Do not exceed one sentence.