1.1 Software Engineering Overview: Definition

IEEE Standard Glossary of Software Engineering Terminology: The app of a systematic, disciplined, quantifiable approach to dev, operation, & maintenance of software.

1.2 Software Engineering Overview: Origins

Sys engineering during WWII, focusing on building complex sys through work processes, optimisation methods & risk management

1.3 Software Engineering Overview: Key Contributions

Coined by Margaret Hamilton (1963–64) at MIT during Apollo missions. Aimed to elevate software to same level of respect as hardware engineering. Gave legitimacy. ***NATO Software Engineering Conferences (1968/69)***: Friedrich L. Bauer addressed "software crisis", proposed systematic approaches to ensure reliability & efficiency. ***1966 (ACM)***: Anthony Oettinger emphasised recognition of software engineering as engineering profession w/o boundaries between "scientific" & "Biz" apps. ***1972 Redefinition***: Redefined as establishment & use of sound engineering principles to economically produce reliable software for real machines

1.4 Software Engineering Overview: Software Crisis 1.0

What?: Circa 1960s. Challenges in **time, cost, & quality** due to lack of scalable programming practices & rapid increase in computational power. ***Why?***: Emergence of software complexity made traditional dev unmanageable. Growth in computer power outpaced capabilities of software engineering. **Edsger Dijkstra**: Early programming was manageable, but w more powerful computers, programming became equally gigantic in complexity. ***Response***: **R&D in Software Engineering**: **1. Process models**: Structured software dev methodologies. **2. Software Architecture**: High-level design frameworks. **3. Design principles & patterns**: Standard solutions to common problems. **4. Testing methods**: Systematic validation approaches. **5. Software Design Patterns**: Reusable design templates for specific issues.

1.5 Software Engineering Overview: Software Crisis 2.0

What?: Modern Era. Disparity between available technological resources & ability to meet user expectations. Exponential growth in complexity of sys & user demands ***Why?***: **Push factors**: Advances in hardware & cost reductions, proliferation of data & devices. **Pull factor**: Increase demands of digital natives who expect seamless, high-quality experiences. ***Response***: Need for software engineering practices to continuously adapt & innovate to bridge gap between capabilities & expectations

2.1 Software Dev Process: Categorising Software

By Computation & Response: **1. Real-time**: Requires immediate processing & response (e.g., autonomous vehicles, robotics) **2. Concurrent**: Handles multiple processes simultaneously (e.g., chat apps, multiplayer gaming) **3. Distributed**: Components are spread across multiple sys & communicate over a network (e.g., cloud storage, content delivery networks).

By Nature of Code & Data: **1. Open-source software**: Publicly available source code that can be modified (e.g., Linux, Apache, PostgreSQL) **2. Open-content Sys**: Content is collaboratively created & shared (e.g., Wikipedia, OpenStreetMap).

By Deployment Mode: **1. Embedded Sys**: Integrated into devices for specific functions (e.g., IoT devices, medical equipment) **2. Desktop apps**: Designed for standalone use on personal computers (e.g., Microsoft Office, Photoshop) **3. Edge Sys**: Running on devices near data source to minimise latency (e.g., industrial sensors, smart home sys) **4. Cloud-native Sys**: Designed specifically for cloud env, leveraging distributed architectures (e.g., Netflix, Kubernetes)|

Edge Computing Case Study

Definition: Balance computation between centralised cloud & localised edge devices. **Use Cases:** Autonomous vehicles, smart cities, real-time facial recognition. **Challenges:** Limited processing power in edge devices (e.g., cameras, smartphones), battery life, & heat dissipation. **Advantages:** Reduced latency, improved privacy, & localised decision-making

Cloud Computing Case Study

Definition: Hosted on external data centres & accessed over internet. **Characteristics**: Includes vendor-managed resources like networking, storage, & runtime env ***Models*** : **1. IaaS:** Infra as a Service (e.g., AWS, Google Cloud) provides virtualised computing resources. **2. PaaS:** Platform as a Service (e.g., AWS Elastic Beanstalk, Heroku) provides platforms for app dev & deployment **3. SaaS:** Software as a Service (e.g., Google Apps, Salesforce) provides fully managed software solutions for end-users.

Cloud Native Case Study

Definition: Software designed to maximise scalability & flexibility in cloud env. **Features**: Immutable infra (e.g., declarative infra w tools like Terraform); Microservices-based architecture (e.g., splitting apps into independently deployable services); API-driven for integration (e.g., REST, GraphQL); Service Mesh for service-to-service communication (e.g., Istio); Containers for portability (e.g., Docker); Dynamically managed using orchestration tools (e.g., Kubernetes). **Dev Practices**: CI/CD pipelines, DevOps practices, server-less computing (e.g., AWS Lambda). **Stack**: **1. Infra:** Physical & virtualised resources (e.g., VMs, Kubernetes clusters). **2. Provisioning:** Tools for resource provisioning (e.g., Terraform, Ansible). **3. Runtime:** Manages workloads (e.g., container runtimes like Docker). **4. Orchestration & Management:** Automates deployments & scaling (e.g., Kubernetes, OpenShift). **5. Application Dev:** Tools for defining & building apps (e.g., Helm, Skaffold). **6. Observability:** For monitoring & analysis (e.g., Prometheus, Grafana, ELK stack)

Monolithic Design Example: Traditional web apps w tightly coupled architecture (e.g., single server running all components) ***Cloud-native Design Example***: Modern architectures w microservices, API gateways, distributed data storage (e.g., an e-commerce platform w microservices for inventory, payments, & recommendations).

2.2 Software Dev Process: Deploying Software

Definition: Activities that make software available for use after dev. Deployment bridges acquisition of software & its execution. **Impacts**: *Quality Attributes (3.4).* **Issues**: **1. Driven by Software Crisis 2.0**: 1) Advances in hardware 2) Reduction in hardware costs & increased processing power 3) Proliferation of devices 4) Infinite data availability & demand. **2. Need to Rethink Deployment**: 1) Exploit hardware advancements 2) Accommodate varied devices 3) Manage interconnected devices, users, & apps 4) Efficiently handle data. **Challenges**: **1. Integration of Internet & Advances**: Virtual worldwide marketplace creates concerns in shifting to cloud-native architectures. Affects portability. **2. Large-Scale Content Delivery**: Transfers of data/info between end-points creates concerns w location, number of end-points, quality of service. Affects availability & performance **3. Heterogeneous Platforms**: OS/browser/platform diversity & combinations creates concerns w coexistence of platforms & interoperability between platforms. Affects interoperability **4. Dependency & Change Management**: Multiple components have interdependencies - "uses" relationship creates concerns w Handle configurations, version changes(within & outside components), & compatibility. Affects maintainability **5. Coordination Among Components**: Computations among components need coordination creates concerns w API designs & synchronous/asynchronous communication. Affects performance **6. Security Concerns**: Privacy, authentication, authorisation, integrity. Affects security & usability.

Deployment Stages: From Code → Executable → Final Environment. How to transition between these stages?

Pattern Detection Case Study - Vulnerability Detection: Vulnerabilities observed to be caused by single repeating pattern. **Solution**: Exploit ML (+ expert knowledge) to identify patterns. **Mechanism Components**: Encoded expert knowledge. Large volumes of weak evidence. Pattern detection. Deductive & inductive reasoning. Create higher-level features. **Flow**: Code snippet → Probability of vulnerability. Code snippet → CodeQL → Features → ML model → Alert / Non-alert

Boosting Productivity Case Studies: **1. Code efficiently**: Method autocompletion. Boilerplate code generation. Try various approaches quickly. **2. Tackle new problems, creatively**: Generate & explain code, configuration & documentation. **3. Code translation**: Convert code from one language to another. E.g., Legacy code → Python. **4. Writing maintainable code**: Detect code vulnerabilities. Improve code quality & enforce good practices. **5. Better testing**: Generate unit tests for your scripts. Improve test coverage

3.1 SRS: Overview

Definition: **IEEE Standard Glossary of Software Engineering Terminology** - 1) condition or capability needed by user to solve problem or achieve an objective 2) condition or capability that must be met or possessed by sys or sys component to satisfy contract, standard, spec, or other formally imposed documents 3) documented representation of condition or capability as in (1) or (2). **Sommerville & Peter Sawyer** - Reqs are Spec of what should be implemented. They are descriptions of how sys should behave or of sys property or attribute. They may be constraint on dev process or of sys.

Examples: **Usage Centric**: As user, I can upload an image into sys so that I can showcase my photography skills to world. **Product Centric**: The sys will support range of graphic file formats up to 20 mb in size; [Refinement] The sys will support following file formats: jpeg, png, tiff, bmp; [Refinement] The sys will support maximum resolution of 5184 × 3888 pixels

3.2 SRS: End-to-End

Sources: **1. Documents:** Descriptions of current or competing products; standards/regulations; help desk problem reports. **2. Interviews:** Methods w focus groups; use case workshops; Product champions as key customer representatives. **3. Questionnaires & Marketing Surveys:** Pilot to clarify qn, can be challenging to craft clear ones. **4. Event-Response Tables:** Identify external stimuli & describe sys responses. **5. Prototyping:** Useful across reqs gathering, design, & implementation phases. **6. Observation – Watch Users Do Their Jobs:** Workflow diagrams; conducting "day in life" studies; analyse information available to users when performing tasks.

Process: **1. Elicitation**: Discover reqs via interviews, workshops, event-response tables, focus groups, prototypes, watching users, sys interface analysis, UI analysis, document analysis. **2. Analysis**: Analyse, decompose, derive, understand, negotiate, identify gaps. Clarifies Elicitation. **3. Specification**: Written & illustrated reqs for comprehension & use. Close gaps in analysis. **4. Validation**: Confirm that we have correct set of reqs. Rewrites spec. Reevaluates Analysis. Confirms & Corrects elicitation.

Outcome: **1. Rights, Responsibilities & Agreements**: All major stakeholders confident of dev within balanced schedule, cost, functionality & quality. **2. SRS**: A set of precisely stated services & constraints that software must satisfy. A complete description of behaviour of software. **3. Change Control**: Process to ensure that changes to product or sys are introduced in controlled & coordinated manner.

SRS Case Study: **IEEE 830-1993 - Recommended Practice for SRS** - {1. Introduction - 1.1 Purpose, 1.2 Document Conventions, 1.3 Intended Audience & Reading Suggestions, 1.4 Project Scope, 1.5 References 2. Overall Description - 2.1 Product Perspective, 2.2 Product Features, 2.3 User Classes & Characteristics, 2.4 Operating Environment, 2.5 Design & Implementation Constraints, 2.6 Assumptions & Dependencies 3. System Features - 3.1 Functional Reqs 4. External Interface Reqs - 4.1 User Interfaces, 4.2 Hardware Interfaces, 4.3 Software Interfaces, 4.4 Communications Interfaces 5. Nonfunctional Reqs - 5.1 Performance Reqs, 5.2 Safety Reqs, 5.3 Security Reqs, 5.4 Software Quality Attributes 6. Other Reqs - Appendix A: Glossary; Appendix B: Analysis Models; Appendix C: TBD List} **SRS Components**: Interfaces; Functional Capabilities; Performance Levels; Data Structures/Elements; Safety; Reliability; Security/Privacy; Quality; Constraints & Limitations **SRS vs. Product Backlog**: **1. Product Backlog**: Repository of JTBD; Facilitate prioritisation of work; Facilitate planning. **2. SRS**: In-depth description of software product to be developed; Direct/Indirect reqs of sys; Only tells what JTBD. **Good Qualities**: Accurate; Complete; Modifiable; Ranked; Testable; Traceable; Unambiguous; Verifiable; Valid

3.3 SRS: Types

1. Business: Why organisation is implementing sys, e.g., reduce staff costs by 25%. **2. User**: Goals user must be able to perform w product, e.g., check for flight using website. **3. Functional**: Behaviour product will exhibit, e.g., passengers shall be able to print boarding passes. **4. Quality Attributes**: How well sys performs, e.g., mean time between failure ≥ 100 hours. A type of non-functional reqs. **5. System**: Hardware or software issues, e.g., invoice sys must share data w purchase order sys. Affects functional reqs. **6. Data**: Describes data items or structures, e.g., product number is alphanumeric. **7. External Interfaces**: Connections between sys & env, e.g., must import files as CSV. Affects functional reqs. **8. Constraints**: Limitations on design & implementation choices, e.g., must be backward compatible. **9. Business Rules**: Actual policies/regulations. Constrain biz, user, & functional reqs.

3.4 SRS: Quality Attributes

Different apps have varying quality attributes. **1. Embedded Software**: Performance, efficiency, reliability, robustness, safety, security **2. Web Applications**: Availability, integrity, interoperability, performance, scalability, security, usability **3. Desktop & Mobile Software**: Performance, security, usability.

listening to consume. Represent state change or record of an action. Immutable & Ordered in sequence.

Push vs. Pull Communication: **1. Push**: Source actively sends data to broker for consumers. **2. Pull**: Consumers request data from broker as needed.

Understanding Architectures: 1) Divide & Conquer 2) Abstraction 3) Increase Cohesion by ensuring related elements work closely together. 4) Reduce Coupling by minimising interdependencies between components. 5) Reuse Components 6) Flexibility ensures adaptability for future changes.

General Approach: Start with System Context - Identify scope, users, & dependencies. Add Containers - Highlight logical units like apps, databases, & file sys; Include key technology choices. Zoom in on Components - Show decomposition into components & their responsibilities; Identify interactions & technology decisions. Provide Code-Level Details - Drill down into implementation as needed for precision. Can Combine steps for simplicity.

Decomposition, Componentizing & Packaging: **1. Horizontal Slicing**: Design by layers. Layers include controllers, services, & repositories. **2. Vertical Slicing**: Design by feature. Focus on specific features, integrating all relevant layers.

Modularity: **Benefits**: Shorter dev time. Better flexibility. Improved comprehensibility. **Principles**: Decompose large sys into smaller, manageable chunks w well-defined interfaces (APIs). Allocate functions to modules & specify interactions via APIs. **Key Idea**: Manage complexity by breaking sys into modular components.

Cohesion: **1. Functional**: Performs one computation w/o side effects. **2. Layer**: Related services grouped; strict hierarchy between higher & lower levels. **3. Communicational**: Operates on same data. **4. Sequential**: Procedures executed in sequence w one output feeding next input. **5. Procedural**: Procedures called one after another. **6. Temporal**: Procedures executed in same phase of execution (e.g., initialisation). **7. Utility**: Related utilities grouped when stronger cohesion NA.

Coupling: **1. Content**: One component modifies another's internal data. **2. Common/Global**: Modules share global vars. **3. Control**: One module dictates another's behaviour via flags. **4. Data**: Modules share data via parameters. **5. External**: Dependencies on external sys (e.g., OS, shared libraries). **6. Temporal**: Actions occur together due to timing, not logic. **7. Inclusion/Import**: Module includes/imports other modules (e.g., package or library).

Modular Monoliths: Nested models (Eg,. separation by technical logic + separation by biz logic) aim to reduce tight coupling via APIs for communication between domains. Each domain has hidden implementation, encapsulated & accessed only through public-facing API. Modular boundaries improve maintainability & scalability within monolithic application.

Model-Code Gap: Models & source code often fail to align perfectly. Architecture models include abstract concepts like components, design decisions, & constraints that are not directly represented in source code. Source code represents concrete, machine-executable implementation.

Architecture Diagrams: visual representation of sys. Highlight big picture for dev teams & stakeholders. Act as shared vision to guide implementation. Aid in technical discussions about feature implementations. Serve as map for navigating source code. Facilitate onboarding for new team members. **Diagram Hierarchy: 1. System Context** - High-level overview showing sys users & dependencies. **2. Containers** - Represents logical deployment units (e.g., apps, databases, file sys). **3. Components** - Shows interactions & responsibilities within each container. **4. Code** - Details actual implementation.

Best Practices: Ensure diagrams are titled & labeled for clarity; Include legend to explain meaning of arrows, lines, & shapes; Avoid using ambiguous terms like "business logic" w/o explanation; Ensure consistency in directionality & relationships; Provide enough detail to be informative but avoid overwhelming reader.

Drawing Diagrams: Verify sufficient information, ensure architectural drivers (FR, constraints, scenarios) are clear. Decompose sys iteratively 1) Choose element to refine 2) Identify drivers & concepts 3) Instantiate architectural elements & allocate responsibilities. Verify, refine, & documenting results. **Parallelism:** Large-scale sys require parallel design activities, which must be merged into consistent architecture.

Issues: **Unexplained Notations:** Use consistent colour coding, shapes, & symbols w legend. **Ambiguity:** Clearly define elements & their relationships. **Omission of Technology Choices:** Explicitly include tools, frameworks, & options. **Mixing Abstraction Levels:** Avoid combining high-level & low-level details in same diagram. **Overcomplication/Oversimplification:** Strike balance between clarity & detail.

4.2 Software Architecture Design: Architecture Styles

Architecture styles can be segmented into 2 groups - code division/sys deployment

Code Division: **1. Technical Partitioning**: Focuses on separation of concerns. May align w team's expertise: Eg., Frontend devs, Backend devs, Database administrators. **2. Domain Partitioning**: Aligned w domain. Logical components map to problem: Eg., Customer, Payment, Shipping. **3. Visual Partitioning**: 1) Presentation which focuses on UI. 2) Services which contains app logic. 3) Persistence which manages storage & retrieval of data.

System Deployment: **1. Monolithic**: Deploys all logical components as single unit. App runs as one process. **2. Distributed**: Consists of independent logical components. Logical components run as individual processes. Communication happens over network.

Code Deployment	Code Partitioning	Examples
Monolith	Technical	Layered
	Domain	Modular Monolith
Distributed	Technical	Event-driven
	Domain	Microservices

4.2 Software Architecture Design: Layered

Organised as layers of components. Supports independent dev & evolution of different sys parts, depending on how app code is packaged. Comprises one or more layers for software under dev, w each layer having distinct & specific responsibility. **Examples**: 2-tiered/3-tiered/n-tiered architecture.

Typical Design: Each layer communicates w layer directly above. Layers communicate through interfaces. Highest level layers can communicate w 1 or more layers below. Presentation -> Business -> Data.

1-tier Architecture: Presentation, Application, & Resource are merged into one layer **1. Advantages**: Performance optimisation, no context switching overheads. **2. Disadvantages**: Difficult to modify, impractical in modern sys.

2-tier Architecture: Historically emerged w PC. Client communicates directly w server. Server contains app layer & resource management layer. Separates presentation layer which resides in client. Client has ability to further process info provided by server. Clients can be thin (limited functionality)/fat (rich functionality) **1. Advantages**: Performance optimisation by grouping app & resource layers, portable across platforms. **2. Disadvantages:** Limited scalability, increased maintenance complexity w distributed clients.

3-tier Architecture: Historically emerged with increase in network bandwidth provided by LANs. Separates Presentation, App, & Resource Management Layers. Adds App Layer between Presentation & Resource Management layers as Middleware. Middleware between clients & server integrates between different information services. Resource management layer consists of all servers being integrated. **1. Advantages**: Scalability, modularity. **2.**

5.1 Microservices Architecture: Overview

A single application as suite of small services. Each microservice offers well-defined business capability. Each microservice is developed & deployed independently. Communication through well-defined mechanisms like HTTP resource APIs. **Characteristics**: Highly cohesive & loosely coupled; Independent dev & deployment- Services share no implementation details & rely on communication. Owned by small, autonomous teams. **Boundaries**: aligned with business capabilities.

Netflix Microservices Case Study: Over 1,000 microservices. **Examples** Creating main menu list of movies; Determining subscription status for relevant content; Recommending videos based on watch history; Billing credit cards for subscription renewals; Monitoring & migrating users to optimal Content Delivery Appliances (CDA); Storing & transcoding videos for various devices; Adding copyright markers for DRM.

Identifying Microservices: **1. Domain-Driven Design (DDD)**: Complex sys are collection of sub-domains. **Domain**: The problem space. **Sub-Domain**: Specific focus areas, e.g., Sales, Warehouse, Finance. **Bounded Context**: Encapsulation of processes & models. **Aggregates**: Units of transactional consistency. **2. Event Storming**: Focus on discovering domain events. Commands cause events; events may trigger compensating actions. Define microservice boundaries by identifying consistent contexts. **Example**: Product added to cart, Shipping details confirmed, Order delivered.

5.2 Microservices Architecture: Data Patterns

Database-Per-Service Pattern: **Description** - Each service owns & embeds its data, ensuring loose coupling. Services have independent access & control over their respective databases. **Benefits** - **1. Loose Coupling:** Independent scaling of services & databases. **2. Technology Flexibility**: Each service can choose its own database type (e.g., relational, NoSQL) based on requirements. **3. Resilience**: Faults in one database don't affect others. **Challenges** - Managing hundreds of database clusters is expensive & complex. Data consistency across services must be ensured via communication mechanisms.

Data Delegate Pattern: **Description** - A dedicated service (delegate) manages shared data. Other services interact only with delegate service, rather than directly accessing shared data. **Benefits** - **1. Centralised Data Management**: Simplifies control & governance over shared data. **2. Consistency**: Prevents multiple services from modifying same data directly. **3. Loose Coupling**: Reduces dependencies among services. **Challenges** - Introduces additional latency due to delegate acting as an intermediary. Can become bottleneck if not properly scaled.

Data Lake Pattern: **Description**- A centralised, queryable repository aggregates data from multiple services. Data streaming tools like Kafka, RabbitMQ, IBM MQ, or Apache Pulsar are used to stream relevant data into lake. **Benefits** - **1. Unified View**: Provides consolidated view of all service data for analytics & reporting. **2. Optimised Querying**: Aggregated & indexed data improves query performance. **3. Scalability**: Can handle large volumes of data for batch processing or analytics. **Challenges** - Requires reliable data streaming infra. Data governance & security can become complex.

Sagas Pattern: **Description** - Manages distributed transactions through series of steps & compensating actions (undo operations) in case of failure. **Benefits**- **1. Decentralised Coordination**: Eliminates need for centralised transaction manager. **2. Flexibility**: Handles long-running transactions effectively . **3. Resilience**: Enables partial rollbacks, reducing failure impact. **Example** - Airline Reservation: 1) Book seat. 2) Process payment with loyalty points. 3) Send notification. 4) Compensate (e.g., refund) if payment fails.

Event Sourcing: **Description** - State changes are stored as events rather than persisting current state. Projections are used to derive current state from event log. **Benefits** - **1. Complete History**: Provides full audit trail of state changes. **2. Flexibility**: Enables replaying events to rebuild state in case of failure. **3. Scalability**: Works well with distributed sys as events are immutable. **Examples**: **Chess**: Moves reconstruct board state. **Accounting Journals**: Balances derived from transaction logs.

CQRS (Command Query Responsibility Segregation): **Description:** - Separates read & write paths. Commands trigger events, & materialised views are created for querying. **Benefits** - **1. Independent Scaling**: Read & write models can scale independently. **2. Optimised Query Performance**: Precomputed materialised views improve efficiency. **3. Flexibility**: Different database types can be used for reads & writes.

5.3 Microservices Architecture: Deployment Patterns

Service Instance Per Host: **Description** - Each service instance is deployed in isolation, either on VM or in container. **1. Service Instance Per VM**: Netflix packages services as EC2 AMIs using Aminator. **2. Service Instance Per Container**: Containers include required libraries & applications managed by tools like Kubernetes. **Benefits** - **1. Isolation**: Each service has dedicated resources, improving fault tolerance. **2. Flexibility**: Allows independent deployment & scaling. **3. Portability**: Containers can run across various env.

Immutable Infra: ** Description** - Components are immutable; modifications require recreation of infra. **Benefits** - **1. Predictability**: Immutable components reduce configuration drift & errors. **2. Reproducibility**: Ensures consistent behaviour across env. **3. Simplified Updates**: Changes are applied by creating new instances, avoiding live modifications.

Infra as Code (IaC): **Description** - Infra is defined & managed using machine-readable files (e.g., Terraform scripts). **Benefits** - **1. Version Control:** Infra changes can be tracked & managed like application code. **2. Automation**: Enables consistent & repeatable provisioning of resources. **3. Scalability**: Simplifies management of large-scale sys.

Service Collaboration: **Description** - Patterns for coordinating service interactions. **1. Orchestration** - A central controller manages workflows across services **2. Choreography**: - Services interact independently, relying on events & triggers **Benefits** - **1. Orchestration**: Provides centralised control & visibility of workflows. **2. Choreography**: Enables decentralised interactions, reducing dependencies.

Service Communication: **Description** - Patterns for inter-process communication. **1. Request-Response** (Synchronous): TripManagement requests passenger info from Passenger Management. **2. Event-Driven** (Asynchronous): Order events trigger shipping processes. **Benefits** - **1. Request-Response**: Ensures immediate responses for critical queries **2. Event-Driven**: Decouples services & supports real-time data propagation.

Service Discovery: **Description** - Patterns to locate & manage service instances. **1. Client-Side Discovery**: Clients query service registry to find available instances. **2. Server-Side Discovery**: Load balancer queries registry & routes requests to appropriate instances. **3. Service Registry**: Zookeeper, Consul, Kubernetes. **Benefits**- **1. Dynamic Scalability**: Automatically adapts to changes in service instances. **2. Improved Fault Tolerance**: Supports redundancy through registries.

API Gateway: **Description** - A single entry point for all requests into sys. **Responsibilities**- Routing, authentication, monitoring, caching. **Benefits**- **1. Simplified Client Interaction**: Hides internal complexities of microservices. **2. Enhanced Security** - Centralises authentication & access control. **3. Load Balancing**: Distributes traffic efficiently across services.

6.1 Events Driven Architecture: Events

Events are ways for software component to let rest of sys know that something important has just happened. Through events, software components pass information to other components. **Initiating Event**:

8. Asynchronous Communication

Communication can be Synchronous/Asynchronous. Can have single/multiple Receivers. Can be Persistent/Transient. Async is "Fire-&-forget communication"; When software component broadcasts information, it does not wait for response or care if recipient is available. **Examples** - Persistent Asynchronous: Email; Persistent Synchronous: Messaging; Transient Asynchronous: UDP; Transient Synchronous: RPC.

Async vs Sync: **1. Async**: Communication includes lag between message sent & received/responded. Caller continues execution without waiting. Supports independent sender & receiver functioning. One-to-many communication. **Example**: AMQP **2. Sync**: Caller sends message & waits for receiver to respond. Involves request-response patterns. **Example**: HTTP/HTTPS **Advantages** - Better responsiveness: Faster response time for requests; Better availability: Components do not depend on availability of other components. **Trade-offs** - More complex error handling compared to synchronous communication.

HTTP/HTTPS Case Study: **1. Sync Request-Reply**: Client (e.g., web browser) sends resource request to server. Server sends back response or an error message. Resources can be - Simple files (e.g., HTML); Dynamically generated responses. **2. Async Request-Reply**: **A.** Client sends request. API responds w 202 (Accepted) & status link. Client polls status endpoint using HTTP GET requests. Pending: Returns HTTP 202. Complete: Returns HTTP 302 (redirects to resource). **B.** Asynchronous API endpoint accepts client requests & queues them. Backend worker processes queue. Status endpoint monitors 1) Request completion 2) Response provision or resource redirection 3) Pending status includes HTTP 202 & ETA.

Events vs Messages: **Similarities**: Use asynchronous communication; Decouple services; Carry information between services. **Differences**: **Event**: Represents fact or state change. Represents something that has occurred. Broadcasted using Pub-Sub patterns (e.g., Kafka Topics). **Message**: Represents request (e.g., command or query). Represents request for action or data. Delivered via point-to-point interactions (e.g., AWS SQS).

Tenant Manager Case Study: **Req** - Tenant Manager Service ensures all domain services delete or anonymise tenant-specific data. Notification Service sends deletion confirmation email. **Design 1** - Tenant Manager publishes `TenantDeleted` event; Domain services consume & react to event; Notification Service sends email after deletion. **Revised Design** - Tenant Manager sends direct messages to services; Waits for acknowledgments from domain services; Sends message to Notification Service after confirmations.

Topics vs Queues: **1. Topic** - Supports one-to-many communication; Allows non-sequential processing with unique offsets. **2. Queue** - Supports single-receiver processing. Enforces FIFO order.

Protocols & Brokers: Protocol is message formatting & broker is message medium **1. Single Receiver Setup** - Point-to-point communication; Ensures exactly one consumer & single message processing. **2. Multiple Receiver Setup**: Pub/Sub mechanism allows broadcasting to multiple subscribers.

Kafka Case Study: **Description** - Open-source platform, originally developed by LinkedIn. Donated to Apache. Supports horizontal scaling via multiple brokers. **Structure** - Events categorised into topics & partitions; Events with same key routed to same partition. **Benefits** - Enables non-downtime scaling; Each broker communicates to maintain cluster.

AMQP – Advanced Message Queuing Protocol Case Study: **Features** - Peer-to-peer communication. Asynchronous messaging protocol. Messages routed via exchanges to queues. Support asynchronous persistent communication. Intermediate storage for message while sender/receiver are inactive. Communicate by inserting messages in queues. Sender is only guaranteed that message will be eventually inserted in recipient's queue. No guarantees on when\if message will be read **Exchange Types**: **1. Direct**: Routes based on exact key match. **2. Fanout**: Broadcasts to all bound queues. **3. Topic**: Wildcard match between routing key & binding pattern. **Example**: RabbitMQ

Persistent vs Transient Communication: **1. Persistent** - Messages stored at intermediate hops until processed. Guarantees eventual delivery. **2. Transient** - Temporary buffering; discarded if undeliverable. Requires active sender & receiver.

Persistence Combinations: **1. Persistent Asynchronous** - Non-blocking sender; Guarantees eventual delivery. **2. Persistent Synchronous** - Blocking sender until receipt acknowledgment. **3. Transient Asynchronous** - Non-blocking sender; messages discarded if receiver inactive. **4. Transient Synchronous** - Blocking sender; acknowledgment based on delivery or response.

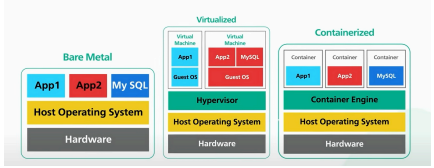
9 Messaging Patterns

Enables Async Communication. **Process** - Components/Services send packets of data (messages). Messages are payloads like strings, byte arrays, records, or objects. **Enterprise Integration:** Enabled through messaging technology. Use of technology to tightly connect everything in an enterprise: Applications, data, clouds, APIs, processes, devices. Seamless sharing of information between sys. MOM (Message-Oriented Middleware) providing messaging capabilities.

Enterprise Integration Patterns (EIPs): Collection of technology-independent solution to integration problems. Provides common language for Devs & App architects. Vendor-independent. **Examples**: Message Channel; Point-to-Point Channel; Pub-Sub Channel; Message; Message Endpoint

Message Construction: **Header:** Structural information (e.g., type, origin, destination, size). **Payload:** Actual information. **Properties:** Optional metadata for filtering. **Types** - TexT: Java String (e.g., XML content); Map: Name-value pairs; Bytes: Stream of uninterpreted bytes; Stream: Stream of primitive values; Object: Serialisable Java object. Empty: Contains only header fields & properties.

Message Channels: Connect senders & receivers. One-directional. Two-way communication requires Request & Reply channels. Correlation Identifier Used to match replies to requests. **Request/Reply Process:** Requestor assigns unique Request ID. sends message to jms/RequestQueue. Replier stores Request ID as Correlation ID in reply. Replier replies to jms/ReplyQueue address in request message. Requestor processes reply using Correlation ID. **Request-Reply chaining**: When request causes reply, & reply is in turn another request that causes another reply, & so on. Useful for message traceability **Special Purpose Channels**: **1. Invalid Message Channel**: Handles erroneous messages. **2. Dead Letter Channel (DLQ)**: Manages undelivered messages. Messages are redriven back to source/another queue.



Deploying Mechanisms: **1. Bare Metal**: **Pros**: Complete Control; Physical Isolation. **Considerations**: From code to executable - catering to target platforms; customised build & linking; availability of libraries & dependencies; Potentially wasted hardware resources; Cost; Scalability; Reduced dev productivity. **2. Virtual Machines**: **Pros**: Improved resource utilisation; Reduced cost; Flexible; Scalable **Considerations**: Full OS running inside VM; Susceptible to side-channel attacks; "Noisy neighbour" problem **3. Containers**: **Pros**: Lighter than VM - include only OS processes & dependencies where necessary; better utilisation of hardware capacity; rapid deployment. Write once run everywhere - provision containers to run apps; ideal for CI/CD, Agile/DevOps practices. Granular & controllable - deployment can be whole sys or elements within sys; monitoring available; rollback/patch/redeploy easily. Easy integration of internet & related advances - build cloud-native apps. Include runtime w code - caters to heterogeneous platforms; can achieve interoperability & portability. Supports dependency & change management - improved maintainability & portability. Environment Management - applies to both dev & deployment (Eg. Dockerfile provides env spec.). Reproducible - guaranteed to be identical on any sys that can run containers. Isolation & Security - avoid conflicting dependencies; can provide some sand-boxing for code execution. Quick to launch; Supports DevOPs best practices; Can be used w orchestrator like Kubernetes - Orchestration automates provisioning, deployment, networking, scaling, availability, & lifecycle management of containers; Directs how & where containers run. **Considerations**: Not suitable for all apps;Not suitable for performance-critical apps. **Container vs Orchestrator**: **Containers** provide platform for building & distributing services; Not suited for running complex apps on its own; Often need multiple containers, each doing specific tasks/providing specific services. **Orchestrator** integrate & coordinate many parts; Scale up/down deployment based on demand; Fault tolerance for app; Communication among containers. Orchestrator works w containers to realise production (E.g., Container: Docker, Orchestrator: Kubernetes) **4. Server-less Functions**: Cloud provider manages physical servers. Dynamically allocate resources on behalf of devs (or users) in production. Works based on Function-As-A-Service (FAAS) - event-driven execution model (run when needed); App logic deployed in containers; Containers managed by platform/service provider; Stateless apps; Ephemeral – short execution times. Cloud-native dev model; Devs/enterprises don't manage servers. Economics: Metered on demand ⇒ no cost when idle (Eg: AWS Lambda). **Process**: 1) Devs package code in containers 2) Application deployed via containers 3) Once deployed, apps respond to demand – automatically scale up/down as needed; **Use Cases**: Batch processing of image & video; Scheduled data compression

2.3 Software Dev Process: Process Models

Goal-directed & Evolving Process requiring Transformation & Creative Activity. **Projects involve people, product, technology, etc.**: Reqs, design, spec, code, test cases, ..., Phases, activities, milestones, ... ; Organisation of team, communication channels, ... ; **Reduces risk of failure**: ~20% of large projects fail; Large number of projects delayed; **Framework for project planning & execution**: Who is doing what, when & how; **Divide software dev work into phases to improve**: Design; Product management; Project management; **Examples**: Waterfall, Spiral, Rapid Prototyping, eXtreme Programming, Rational Unified Process (RUP), Test driven dev, Agile (Scrum, Crystal), etc.

If reqs are well-understood, fixed, & effort predictable: Waterfall model; Highly structured approach; Good for stable reqs & familiar domain & solution; Variation: Have feedback loops. **For fuzzy & evolving reqs**: Iterative & incremental dev; Develop sys iteratively & incrementally; Learning from earlier cycles; Types: depth-first & breadth-first

2.4 Software Dev Process: Best Practices

Agile : Methodology emphasising iterative dev & cross-functional collaboration. Individuals & interactions over processes & tools. Working software over comprehensive documentation (fastidious paperwork, not dev docs). Customer collaboration over contract negotiation. Responding to change over following plan. Focus on quick response to changes. Eg. SCRUM. Work done in sprints, where subset of product backlog is cleared. Often daily 15 min SCRUM meeting.

CI/CD: **Continuous Integration**: Dev practice requiring devs to integrate code into shared repository. Each check-in is verified by an automated build, allowing teams to detect problems early. **Continuous delivery**: above + ensuring every good build is potentially ready for production release. Can be released into production at any given time. **Continuous deployment**: above + automating release of good build to production env. Released automatically into production **Benefits**: 1) Low-risk releases 2) Faster time to market & early feedback 3) Higher quality 4) Lower cost 5) Code changes to an app are released automatically into production env. **Use Case**: Ability to get changes of all types - including new features, configuration changes, bug fixes, & experiments - into production, or into hands of users, safely, quickly & sustainably **Example**: Blue-Green Deployments

DevOps: Blends software dev & operations staff & tools. A set of software dev practices that combine software dev (Dev) & operations (Ops). Intends to reduce time between committing change & change release while ensuring high quality. **Ops**: 1) Check in code 2) Pull code changes for build 3) Run tests 4) Store artifacts & build repository 5) Deploy & release 6) Configure Env 7) Update databases 8) Update apps 9) Push to users 10) Application & Network Performance Monitoring. **Benefits**: **Speed of Delivery**: Enables dev & op teams to deliver faster for customers. CI/CD automate software release process from build to deploy can quicker release new features & fix bugs. **Reliability**: CI/CD practices in testing ensure change is functional & safe. Monitoring & logging help stay informed of performance in real-time. Contribute to reliable delivery & positive experience of end-users. **Scale**: Operating & managing infra & dev processes w automation & consistency improve scalability. IaC help to manage dev, testing, & production env in repeatable & more efficient manner. **Improved collaboration**: Dev & op teams collaborate closely, share responsibilities, & workflows. Reduces inefficiencies & saves time (e.g. reduced handover periods between dev & ops)

Trends: Increasing focus on improving software engineer efficiency – **(Generative) AI** in software dev; **Automation**: Know what needs to be done. Do it efficiently, reliably every time! Repeated tasks. E.g., workflow automation (CI/CD, infra deployment, ...); **Pattern Detection**: Tasks adhere to rules. Exploit rule structure. E.g., identify query patterns. Sift through gigantic codebases to detect vulnerabilities; **Collaboration**: Code sequence prediction & generation. Brainstorm code solutions. Rapid prototyping E.g., Copilot, ChatGPT

External: Observed when software is executing. Impacts UX. Develops user's perception of software quality. **1.1 Availability**: Measure of planned uptime where sys is fully operational. Increased redundancy to meet availability reqs. Hot-back & failover mechanisms increase complexity. $\text{Availability} = \frac{\text{Uptime}}{\text{Uptime} + \text{Downtime}}$ **1.2 Installability**: Ease of installing sys for end-user. **1.3 Integrity**: Prevent information loss & preserve data correctness. **1.4 Interoperability**: System readiness for exchanging data & services w other software & hardware. **1.5 Performance**: System responsiveness & UX **1.6 Reliability**: Probability of software executing w/o failure for specific period of time. **1.7 Robustness**: Degree of sys performance when faced w invalid inputs, defects, & attacks. **1.8 Safety**: Prevents injury or damage to people or property. **1.9 Security**: Authorisation, authentication, confidentiality. Protect data from unauthorised access, allow only authorised use, prevent data tampering. **1.10 Usability**: User-friendliness & ease of use. Measured by, 1) Average time for task completion 2) Number of errors made by users 3) Waiting times 4) Ease of learning & use (e.g., tooltips, autocompletion).

Internal: Not directly observed when software is executing. Perceived by devs & maintainers. Encompass design aspects that may impact external attributes. **2.1 Efficiency**: How well sys utilises hardware, network, etc. **2.2 Modifiability**: How easily designs & code can be understood, changed, & extended. **2.3 Portability**: Effort needed to migrate software from one env to another. **2.4 Reusability**: Effort required to convert software component for use in other apps. **2.5 Scalability**: Ability to grow to accommodate more users, servers, locations, etc., w/o compromising performance or correctness. **2.6 Verifiability**: How well software (components) can be evaluated to demonstrate that it functions as expected.

Security Reqs in SRS : Often overlooked if not explicitly defined in SRS. Features not explicitly stated (e.g., intrusion detection) have high likelihood of exclusion during design. Specifying security req in SRS ensures acceptance tests include them, improving security assurance. Business owners aware of trade-offs (e.g., usability vs. security risks) should help in specifying.

Examples: **1. Authentication**: Strategies for password recovery, account lockout, session expiry, & re-authentication. **2. Auditing & Logging**: Activity logging specifics, detail levels, & access permissions. **3. Intrusion Monitoring**: Define suspicious activities, potential fraud, & response mechanisms.

Performance Reqs: Influences **1. Architecture**: May require replicated databases/caching **2. Hardware**: Latency needs may dictate specific configurations. **3. Deployment**: Decisions between monolithic/microservice architectures

Types of Scaling: **1. Vertical Scaling**: Adding capacity to existing machines. Advantages: Easier to maintain. Disadvantages: Single point of failure. **2. Horizontal Scaling**: Adding more machines (nodes). Advantages: Improved resilience & fault tolerance. Disadvantages: Increased cost & complexity.

3.5 SRS: Management

Process: 1) Collect Initial Reqs 2) Analyse Reqs 3) Define & Record 4) Prioritise Reqs 5) Agree on & Approve Reqs 6) Trace Reqs to Work Items 7) Query Stakeholders Post-Implementation 8) Utilise Test Management 9) Assess Impact of Changes 10) Revise Reqs 11) Document Changes **Tools**: Word; Spreadsheet; GitHub; JIRA; JAMA; RequisitePro

Prioritisation: Quality attributes often conflict, requiring prioritisation based on app reqs. Balance FR & NFR. Address safety, security, performance, & scalability trade-offs.

Traceability: **Elements**: UIN, type, high-level description, priority. **Links to**: Use case references. Design doc, code modules, test cases. **Example**: Can NFRs trace to code?

Validation vs. Verification: **1. Validation**: Ensures reqs align w biz objectives. **2. Verification**: Ensures reqs correctly written (e.g., completeness, correctness, feasibility). **Techniques**: **A. Informal**: Peer desk-check, pass-around, walkthrough. **B. Formal**: Inspection w checklists & formal processes.

Documentation: **1. Textual**: Vision & Scope, Use Case, SRS, Product backlog. **2. Visual**: Structured Models - Data flow diagrams, entity-relationship diagrams. Object-oriented Models - State-transition diagrams, dialog maps. **3. Recollect**: Feature List, User Story

AI Use: **1. Reqs Elicitation**: Persona spec as domain experts. Context-based qn generation using LLMs. **2. Analysis**: Automated refinement, sub-task identification, tagging. **3. Predictive Analytics**: Identify & prioritise successful reqs. **4. Automated Tagging & Categorisation**: Labelling & binning based on req type, priority, & other characteristics. Improves traceability of reqs. Enhances ability to link test cases, code modules, & design documents to specific reqs. Eg., Automating tracing of testing processes to reqs.**5. NLP in Reqs Analysis**: Analysing natural language reqs to identify ambiguities, Conflicting reqs, Missing elements. Ensures reqs are clear, consistent, & complete. **6. Predictive Analytics for Reqs**: Predicts which reqs likely to be implemented successfully. Helps in identifying high-priority reqs. Assists in focusing resources on most critical & feasible reqs. **Considerations**: Be cautious of non-deterministic outputs. Address potential non-explainable chains of thought in AI predictions to ensure transparency.

4.1 Software Architecture Design: Overview

The overall organisation of sys. A high-level design blueprint to guides dev process. **Bass, Clements & Kazman Definition**: Structure/Structures of program/computing sys, comprising Software components, externally visible properties of components & relationships among components.

Characteristics: **1. Represents Structure**: Depicts organisation of data & program components necessary to build software. Focuses on both structural & behavioural aspects of sys. **2. High-Level Design**: Provides overarching view of sys, avoid low-level implementation details. Establish foundation for detailed sys design. **3. System Organisation**: Defines overall layout & interaction of different components within sys.

Parts: **1. Component**: Models app-specific function; **2. Connector**: Models interactions between components for transfer of control &/or data; **3. Configuration**: Define topology/structure of sys.

Reference Architectures: Common architectural framework used across multiple apps. Provides standard structure that can be reused in different but similar contexts. Explains high-level structures of apps within domain. Facilitates consistency, best practices & efficiency across projects. **Examples**: **1. Web Applications**: Common frameworks like MVC

Architectural Patterns: Reusable solution to recurring architectural problem. Forms the basis for detailed architectural design. Address specific app problems within given context. Provides structured approach to manage limitations & constraints.

Control Flow: Reflect computation order. Describes how focus of control moves throughout execution. Data may accompany control.

Data Flow: Reflect data availability, transformation & latency. Explains how data moves through collection of computations. As data moves, control is activated.

Call & Return: Control moves from one component to another & back. **1. Hierarchical**: Shows control flow w master-control & layered subprograms. **2. Non-Hierarchical**: Illustrates interaction between manager objects & procedural calls.

Message & Event: Components communicate via event notifications, message passing, RPC, or other protocols. Can be async/sync. Can involve point-to-point communication. **1. Message**: Data sent to specific address. In message-driven sys, each component has unique address. Components await messages & react to them. **2. Event**: Data emitted by component for anyone

Disadvantages: Expensive communication, challenges w internet-based integrations.

n-tier Architecture: Extends 3-tier by adding more layers (e.g., distributed sys or cloud services). Increases flexibility but adds complexity.

4.3 Software Architecture Design: Pipe & Filter

Data enters sys (from data source) & flows through filters one at a time until assigned to some final destination (data sink). A series of transformations on successive input data occurs. Data source, data sink & filters have set of inputs (read) & set of outputs (produce). **Example**: Incoming data → Decrypt (Filter) → Authenticate (Filter) → De-Duplicate (Filter) → Output cleaned data.

Typical Design: Divide app's task into several self-contained data process steps & connect steps to data processing pipeline via intermediate data buffers. Data should flow in **streams**: Good for processing images, audio, video, or batch data. Good for limited user interaction, like batch processing sys.

Filters: Transforms input streams; Computes incrementally, output begins before input is consumed; Independent, sharing no state with other filters; Apply operations; Expect input in particular format; Produce output in defined format; Independent of other components data

Connectors (Pipes): Each pipe transmits outputs of one filter to inputs of another.

Class Responsibility Collaborators (CRC) for Pipe & Filter: **1. Filter**: Responsibility - Gets input data; Performs function on its input data; Supplies output data. Collaborator: Pipe. **2. Pipe**: Responsibility - Transfers data; Buffers data; Sync active neighbours Collaborators: Data Source, Data Sink, Filter. **3. Data Source**: Responsibility - Delivers input to processing pipeline Collaborator: Pipe. **4. Data Sink**: Responsibility - Consumes output Collaborator: Pipe.

Azure Functions Case Study: **Domain**: Image Processing. **Pipe**: Azure Storage Queue. **Filter**: Azure Functions for 1) Content moderation 2) Resizing 3) Reorientation 4) Watermarking 5) CDN publication **Example Workflow**: Unprocessed images → Azure Storage Queue → Azure Function (Content Moderation) → Azure Storage Queue → Azure Function (Resize) → Azure Storage Queue → Final Processed Images.

Processing Invoices Example: Each filter incrementally processes data & passes it to next filter via pipe. Control is distributed s.t. each filter can run when it has necessary data. Data sharing is strictly limited to what is transmitted on pipes. **Workflow**: Read issued invoices → Identify payments → Find payments due → Issue payment reminder.

4.4 Software Architecture Design: MVC

Support user's mental model of relevant info. Enable user to inspect & edit info. Utilises an **observer pattern** for sync between Model & View.

Components: **1. View** - Handles UI elements like buttons, text boxes, & widgets; Observes Model changes & updates UI; May overlap w Controller in tightly coupled sys; Variants include input processing via View. **2. Controller** - Acts as intermediary, coordinating between View & Mode;. Updates Model based on user actions; Triggers change in View.**3. Model** - Encapsulates biz logic & data persistence; Maintains app state & notifies observers of changes. **Examples**: GUI-based sys for enhancing maintainability.

MVC Variants: Flux, MVA (Model-View-Adapter), MVP (Model-View-Presenter), MVP/PM → MVVM (Model-View-ViewModel), WebMVC → SPA (Single Page Application)

Benefits: **1. Separation of Concerns**: Output/UI presentation independent of input handling. Application state separate from user interaction **2. Extensibility**: Adding new View/Controller pair/functionality simplified **3. Restricted Communication**: Reduces complexity & side effects. **4. Testability**: Components are easy to mock for testing **6. Support**: Many frameworks (e.g., Angular, React) provide built-in MVC solutions.

Web MVC Case Study: Server hosts Model. Client interacts with server via HTTP requests. **Controller**: Handles HTTP requests, selects Model, prepares View; maps requests to specific handlers; delegates actions (retrieving data/rendering templates). Often split into 1) Page Controller that handles specific page logic 2) Front Controller that manages HTTP requests.**View**: Renders HTTP response (e.g., HTML, JSON); May use templates to render model contents **Model**: Implements biz logic & manages persistence. Can use database directly (e.g., MySQL) or via ORM. Needs to manage concurrent modifications **Examples**: **Traditional Web Apps** - Require full-page reloads for each interaction (e.g., GET/POST requests) **Modern Web Apps** - Use AJAX to fetch data w/o refreshing page; JSON responses processed dynamically by client.

SPA Case Study: A JS program downloaded & continuously running in browser. Allows queries to be sent & data retrieved without refreshing page. Provides fluid UX by avoiding full-page reloads. Frameworks have build step to generate static bundles of HTML, CSS, & JS. Hosted via "View Controller" (index.html). Use JS logic to make HTTP API requests to "API Controllers" that respond w JSON data.**Advantages** - Offers smooth & responsive UX; Saves bandwidth by only transferring data instead of full page reloads; Reduces perceived latency for end users. **Disadvantages** - Development complexity is higher than traditional applications; Requires JS to function; Vulnerable to Cross-Site Scripting (XSS) attacks.

4.5 Software Architecture Design: REST

Defines constraints for transferring, accessing, & manipulating textual data representations (hypermedia) across networks in stateless manner. Provides rule set for creating web services but is not an architecture by itself. Enable uniform interoperability between different applications on internet. Utilises HTTP for data access & manipulation via methods like: **GET:** Retrieve data. **PUT:** Update existing resources. **POST**: Create new resources. **DELETE:** Remove resources. **Advantages:** Sys less tightly coupled, enabling scalability, usability, accessibility & mash-up ability. Stateless interactions improve sys reliability & scalability. **Disadvantages**: Statelessness may reduce performance due to repetitive data transfer. Standardised URI usage may decrease efficiency compared to app-specific formats.

Constraints: **1. Client-Server Architecture:** Separation of concerns between client (presentation) & server (data storage). Benefits: Portability of UI; Scalability of server components. Independent evolution to support Internet-scale sys. **2. Statelessness:** Each interaction is self-contained (contains all information within query params, headers, or URI); No client state is maintained on server. Benefits: Frees server resources quickly; Resilient to partial failures; Easier monitoring & debugging (focus on single-request data). **3. Cacheable:** Server responses specify cache-ability & duration. Benefits: Improves network efficiency by reducing repetitive interactions; Enhances user-perceived performance. Drawbacks: Potential to serve stale data. **4. Layered System:** Organise app as network of layers. Each layer only interacts with its immediate neighbour. Benefits: Improved sys complexity management; Intermediary layers (e.g., caching, load balancing) enhance performance & availability. **5. Uniform Interface:** Ensures consistent way of interacting w resources. Resource-based - Stable & unique resource identifiers (URIs). Self-describing messages - Include all necessary processing information. Hypermedia-driven app state - Use links to expose resources & state transitions. Benefits: Decouples implementation from services provided. Suitable for large-grained hypermedia data transfer. **6. Code-on-Demand (Optional):** Allows executable code (e.g., JS) to be downloaded by clients for added functionality. Benefits: Simplifies client's initial implementation; Provides extensibility

Originates from an end user & typically kicks off business process. **Derived Event**: Internal events generated in response to initiating event. Typically, software component responds to an initiating event & broadcasts what it did to rest of sys, within scope of initiating event. Events usually contain data, represented in key/value format. **Key**: For identification, routing, & aggregation operations on events with same key. **Value**: The complete details of event. **Example**: An online order, which includes all order information.

Events Types: **1. Unkeyed Events** - Describe singular statement of fact. Example: Key: N/A, Value: ISBN: 372719, Timestamp: 1538913600. **2. Entity Events** - Represent unique thing, keyed on unique ID of that thing. Example: Key: ISBN: 372719, Value: Author: Adam Bellemare. **3. Keyed Events**: Contain key but do not represent an entity. Used for partitioning event stream to ensure data locality within single partition. Example: Key: ISBN: 372719, Value: UID: A537FE.

6.2 Event Driven Architecture: Overview

Definition: Applications designed based on exchange of events.

Components: **1. Event Producers** - Publish data to streams or queues. **2. Event Brokers** - Receive, store, & route data for consumption; May act as an "event bus" for routing information **2. Event Consumers**- Listen for & consume event data. **Advantages** - High performance: Asynchronous processing; Scalability: Decoupling enables independent scaling of services; Fault tolerance: If one service fails, others continue to operate; Evolvability: Easy to add new functionality by creating derived events. **Challenges** - Complexity: Asynchronous communication increases sys complexity; Testing: High effort required for testing independent, decoupled services.

EDA v.s. Microservices: **1. EDA** - Relies on asynchronous communication; Responds to things that have happened (event processing) **2. Microservices** - Built on synchronous communication (occasionally asynchronous); Handles requests or commands for actions that need to happen (request processing).

EDA + Microservices: **Definition** - A hybrid architecture combining microservices & EDA principles. **Workflow** 1) Producer microservices publish events to streams 2) Consumer microservices process events from input streams 3) Event data serves as both storage & communication mechanism.

6.3 Event Driven Architecture: Event Broker

An Event Broker receives events, stores them in partitions/queues. & Enables consumption based on topics. **Key Features** - 1) Immutability: Events cannot be modified after publication; 2) Replayability: Consumers can re-read any event for state recovery or debugging; 3) Strict Ordering: Events in stream partition are delivered in same order as published; 4) Infinite Retention: Event streams can retain data indefinitely. **Examples**: RabbitMQ; Kafka; Azure Event Hubs; AWS EventBridge **Frameworks**: Spring Cloud Stream; Axon Framework **Monitoring Tools**: Prometheus; Elastic Stack (ELK); New Relic.

Partition Example - Kafka: Producers distribute events across partitions; Consumers read from partitions; Scalability achieved by adding machines & rebalancing.

7.1 Scalability: Overview

"Scalability is property of sys to handle growing amount of work by adding resources to sys." Increase capacity in an app-specific dimension by managing resources like CPU, memory & servers **Strategy 1**: Replicate software processing resources to increase throughput. **Example**: Increase number of requests sys can process in time period; Manage larger volumes of data. **Strategy 2**: Optimise available resources **Example**: Use more efficient algorithms; Add database indexes for faster queries; Rewrite servers in faster programming languages; Real-world analogy: Allocate more traffic lanes for high-demand directions during peak hours (morning/evening).

Supermarket Chain Case Study: Opening new stores. Increasing number of self-checkout kiosks in every store. **Software Requirements**: Handle increased volume from item scanning without decreased response time; Process & store larger data volumes from increased sales; Manage inventory, accounting, planning, & more; Evolve ordering predictions to anticipate sales & stock needs; Generate real-time (daily/hourly) sales summaries from each store, region, & country; Compare trends & identify unusual events (e.g., unexpected weather conditions, large event crowds); Help affected stores respond quickly.

Scale Cube: **1. X-axis:** Horizontal scaling (identical application copies) **2. Y-axis:** Functional division (different functions/services) **3. Z-axis:** Data distribution.

Increasing Responsiveness Use caching. Acknowledge requests without immediate database persistence. Send data to queue for asynchronous database writing.

7.2 Scalability: App Scaling

Extends stateless, load-balanced, cached architecture. Services call dependencies multiple times & are also replicated & load-balanced. Providing replicas for web clients & for mobile clients each of which can be scaled independently based on load experienced. Two replicated services. Each load balanced & employs caching to provide high performance & availability. Both utilise core service that provides database access. **Example:** Amazon calling 100+ services per user request.

Swim Lanes Architecture: **Description** - Isolate groups of services within boundaries to prevent failure propagation; Each swim lane for tranche of customers. **Applications:** Region-specific requirements, faster responses. **Example:** Netflix architecture.

7.3 Scalability: Service Scaling

Monoliths grow in complexity with additional features. Suffices if request loads stay relatively low. If request loads grow: Requests take longer to process. Single server becomes overloaded & bottlenecked.

Scale Up: Upgrade server hardware. **Example** Upgrade from t3.xlarge (4 CPUs, 16GB memory) to t3.2xlarge (8 CPUs, 32GB memory).

Scale Out: Replicate service & distribute requests among replicas. **Reqs** - **1. Load Balancer:** Distribute user requests & relay responses. User requests are routed to replicas via load balancer. Load balancer relays responses back to client. **2. Session Store:** Maintain user session data accessible across replicas. Manages unique user sessions for applications. Load balancer must allow servers to share requests evenly. (Eg., Shopping cart data must be stored for any replica to access.) **Challenges:** Adding service instances increases processing capacity, but database response time may still limit sys. **Solutions:** Query database less frequently using caching; Scale out database using distributed databases **Example**: If N replicas handle R requests, each server processes R/N requests.

Caching: Stores commonly accessed database results in memory for faster retrieval. Processing logic should check cache before querying database. Cached data must be refreshed or invalidated appropriately. **Example:** Store weather forecast data until it expires.

7.4 Scalability: Database Scaling

2 main ways to solve this. Scale up with more powerful data stores. Scale out with distributed databases.

Read Replicas: **Configuration** - Primary node handles writes; Secondary nodes serve reads. **Replication** - Asynchronous changes replicated to secondary **Example** Geographically distributed replicas for global clients.

Partitioning: Similar to sharding. **1. Horizontal Partitioning** - Rows divided across partitions based on specific criteria (e.g., hash function on primary key/row value). **2. Vertical Partitioning:** Columns divided into partitions (e.g., static vs. dynamic data)

Datatype Channels: Handles specific data types. **Example**: RabbitMQ Direct exchange uses routing keys for targeted delivery.

Message Routing: Routers process & forward messages Simple Routers route messages from one inbound channel to one or more outbound channels. Composed Routers combine multiple simple routers to create more complex message flows. **1. Content-Based Router:** Routes based on message content. Has to have knowledge of all possible recipients & their capabilities **2. Context-Based Router:** Handles failover/load balancing. **3. Message Filter:** Eliminates unwanted messages. Only single output channel. If message content matches criteria specified by Message Filter, message is routed to output channel, otherwise message is discarded **Pub-Sub Channels w Filter**: Sends messages to all subscribers.

Content-Based Router vs PubSub + Filter: **Content-Based Router**: Exactly one consumer receives each message; Central control & maintenance - predictive routing; Router needs to know about participants. Router may need to be updated if participants are added or removed; Often used for biz transactions, e.g. orders. Generally more efficient with queue-based channels. **Pub-Sub + Filter**: More than one consumer can consume message; Distributed control & maintenance - reactive filtering; No knowledge of participants required; Adding or removing participants is easy; Often used for even notifications or informational messages; Generally more efficient with pub/sub channels.

Message Transformation: **Translator:** Converts between formats (e.g., EDI to XML). **Canonical Data Model:** Provides common format for all sys. **Scatter-Gather Pattern Process:** Broadcasts single message, gathers responses, & aggregates them. **Example:** Mulesoft ESB.

Message Endpoints: Interface between application & messaging sys. Can be used to send messages or receive them, but one instance does not do both. An endpoint is channel-specific, so single application would use multiple endpoints to interface with multiple channels. In JMS, two main endpoint types are MessageProducer, for sending messages, & MessageConsumer, for receiving messages. **Consumers**: **1. Polling**: Controls message consumption rate. Proactively reads messages once it is ready to consume them **2. Event-Driven**: Reacts to incoming messages.

10 Object-Interaction Patterns

A design pattern is solution to recurring problem in context. **Key Elements**: **1. Context** - Situation in which pattern applies; Should be recurring **2. Problem** - Goal to achieve in context; Includes any applicable constraints **3. Solution** - General design to address problem; Ensures goal achievement while satisfying constraints. **Benefits**: A toolkit of tested solutions for common software design problems. Shared vocabulary for efficient communication among team members. **Creational**: Object creation (e.g., Factory). **Structural**: Relationships between objects (e.g., Facade). **Behavioural**: Object interactions (e.g., Observer).

Shapes with Colours Case Study: **Context** - A geometric shape class has subclasses for Circle & Square; Additional requirement: Red & Blue colours for shapes **Problem** - Adding new shapes (e.g., triangle) or new colours (e.g., green) exponentially increases subclasses. Example: Adding triangle creates 2 more subclasses (one per colour); Adding new colour creates 3 more subclasses (one per shape). **Solution** - Identify varying dimensions (shapes & colours); Use object composition instead of inheritance: Separate one dimension into its own class hierarchy. Reference other dimension through objects of new hierarchy.

Bridge Pattern: Decouple abstraction from implementation. Enables extensibility by creating separate hierarchies for abstraction & implementation.

Proxy Pattern for BookSearch: Introduces an intermediary (Proxy) between client & base object. Clients reference Proxy, which holds reference to base object & Implements same interface as base object.

BookSearch Case Study: A `BookSearch` class with method `getBook(String ISBN)` . Increased usage demands enhanced performance. Solution needs to: Implement caching (check cache before searching). Keep `BookSearch` cohesive. Respect SRP. Allow third-party cache libraries. Use Proxy Pattern to solve.

Adapter Pattern: Allow incompatible interfaces to work together. **Implementation**: **Service**: The useful (legacy/3rd-party) class. **Adapter**: Implements client interface. Translates client calls into service-compatible format. **Examples**: Shape compatibility adjustments. Square peg into round hole analogy.

Facade Pattern: Provides unified interface to simplify sub-sys usage. Clients interact with facade, which handles sub-sys interactions transparently. **Examples**: API Gateway as facade at architecture level; JDBC in Java.

Observer Pattern: One-to-many dependency for change notification. **Key Components**: **Register/Attach**: Observers subscribe to subjects. **Notify**: Subject notifies observers of changes. **Event**: Triggers in subjects. **Update**: Observers update their state. **Models**: **1. Pull Model**: Observers retrieve detailed information. **2. Push Model**: Subject pushes specific information. **Examples**: MVC pattern. Event-driven architectures (e.g., Android, iOS).

Mediator Pattern: Encapsulates object interaction to promote loose coupling. Components interact via mediator, not directly. Mediator maintains references to components & orchestrates communication. **Examples**: Spring MVC's Dispatcher Servlet. DOM event management in web pages.

Data Transfer Object (DTO): Encapsulates data for efficient transmission between nodes. Reduce remote calls by bundling data. No business logic. Contains accessors & serialisation methods. **Application**: Used in architectures where network overhead needs optimisation.