

1 Software Applications, Deployment & Development Processes

1.1 Cloud Computing

- Software infrastructure hosted on an external data center with services delivered over the internet
- Different models**
 - On-site:** User manages applications, data, runtime, middleware, OS, virtualisation, servers, storage, networking
 - IaaS:** User manages applications, data, runtime, middleware, OS | service provider manages the rest
 - PaaS:** User manages applications, data | service provider manages the rest
 - SaaS:** Service provider manages everything
- Cloud native is the software approach of building, deploying and managing modern applications in cloud computing environments
- 1.2 Deployment (Software Delivery)**
 - Deployment comprises activities that make the software available for use after development (process between software acquisition and execution)
 - Deployment Issues:** Integration of the internet and related advances (Portability), Large-scale content delivery (Availability, Performance), Heterogeneous platforms (Interoperability), Dependency and change management (Maintainability), Coordination and communication among components (Performance), Security

- Deployment Mechanisms:**
 - Bar^e metal:** (+): Complete control, physical isolation | (-): Wasted hardware resources, cost, scalability issues
 - Virtual machine:** (+): Improved resource utilization, flexible, scalable | (-): Vulnerable to side-channel attacks, noisy neighbor problem
 - Container:** (+): Lighter than VM, write once run anywhere, granular control | (-): Not suitable for all apps, not suitable for performance-critical applications
- Container VS Orchestrator VS Serverless:**
 - Container:** Provide the platform for building & distributing services
 - Orchestrator:** Separate software that integrate & coordinate many parts, scale up/down deployment, provide fault tolerance, provide communication among containers
 - Serverless:** Cloud provider dynamically manages the allocation and provisioning of servers (used for small, stateless, event-driven workloads, e.g., processing an image upload, API endpoints)

1.3 Software Development Process:

- Use waterfall model when requirements are well-understood, fixed, and effort predictable | Use iterative development for fuzzy and evolving requirements
- CI/CD Pipeline:**
 - Continuous Integration: Development practice that requires developers to integrate code into shared repo several times a day → Each check-in is then verified via automated build
 - Continuous Delivery: Ensuring that every good build is potentially ready for production release (Manual deployment to production)
 - Continuous Deployment: Automating release of a good build to production environment (Auto deployment to production)
- DevOps:** Blends software development & operations staff and tools → Reduce time between committing change to system and the change being placed into production while ensuring high quality

2 Specifying Software Requirements

2.1 Requirements

- Definition:** Capability needed by a user, Capability that must be met or possessed by a system, Documented representation of a condition or capability, Specification what should be implemented
- User centric Requirement:** Eg. As a user, I can upload a 20 mb image file into the system so that I can retain the original image without loss of quality
- Product centric Requirement:** Eg. The system will support a range of graphic file formats up to 20 mb in size
- Requirement Development Phases:** Elicitation → Analysis → Specification → Validation
- Outcomes of Requirements Development Process:** (1): Software Requirements Specification (SRS), (2): Rights, responsibilities & agreements
- SRS VS Product Backlog:** Product Backlog (Repo of work to be done, facilitates prioritization of work & planning), SRS (in-depth description of software product to be developed, direct/indirect requirements of system, only tells what work is to be done)
- Validation:** Whether you have written the right requirements
- Verification:** Whether you have written the requirements right

2.2 Types of Requirements

- Business:** Describe why organization is implementing the system
- User:** Describe goals or tasks user must be able to perform with the product
- System:** Describes connections between your system and outside world
- Functional:** Specifies something the system should do
- Non-functional/Quality:** Describes something not directly related to system functionality, but how well the system works
- Constraints:** States a limitation on design or implementation choices
- Data**

NOTE: Business Req to be in Vision and Scope document, User Req to be in User Requirements document, FRs, NFRs, System Req, Constraints to be in SRS

2.3 Software Quality Attributes:

- External:** Observed when software is executing, impacts UX, develops user's perception of software quality (Eg. Availability, Performance, Robustness, Safety, Security, Reliability, Integrity, Deployability, Compatibility, Installability, Usability, Interoperability)
 - Security: Specifying security features at SRS ensures that acceptance tests include testing for security | About privacy, authentication, integrity
 - Safety: About whether a system can harm someone or something
 - Performance: Responsiveness of system, impacts UX, includes response time, throughput, data capacity, dynamic capacity, predictability in real-time systems, latency, behavior in degraded modes or overloaded conditions
 - Availability: Planned uptime of system ($\frac{Uptime}{Uptime+Downtime}$)
 - Usability: Measures the effort required to prepare input, operate, and decipher output of software
- Internal:** Not directly observed when software is executing, perceived by developers/maintainers, encompasses aspects of design that may impact external attributes (Eg. Efficiency, Scalability, Verifiability, Portability, Maintainability, Testability, Modifiability, Reusability)
 - Scalability: Ability of application to accommodate growth in application usage (*Vertical scaling*: increasing capacity of system by adding capability to machines used which is easier to maintain, but causes single point of failure | *Horizontal Scaling*: increasing capacity of system by adding additional machines which increases fault tolerance, but adds costs and complexity)

3 High Level Design - Software Architecture

3.1 Software Architecture

- Definition:** Structure of system, which comprise software components, externally visible properties of those components, and the relationships among them
- Consists of:** (1): Component (element that models an application-specific function, responsibility, requirement, task, process), (2): Configuration (Topology/Structure), (3): **Connector** (Element that models interactions among components for purpose of transfer of control/data)

3.2 Common Definitions

- Control Flow:** Reasoning is on computation order, how the focus of control moves throughout the execution
- Data Flow:** Reasoning is on data availability, transformation, latency, how data moves through collection of computations
- Call and Return:** Control moves from 1 component to another and back, can be hierarchical/non-hierarchical
- Message:** Data sent to a specific address (each component has a unique address other components can send messages to)
- Event:** Data emitted from a component for anyone listening to consume (message sent to publishing infrastructure where consumers may later retrieve, is immutable, ordered in sequence of creation)

3.3 Decomposition & Packaging

- Slicing:** (1): Horizontal Slicing: designing by layers, (2): Vertical Slicing: designing by feature
- Principle of Modularity:** Modularization results in shorter development time, better flexibility, better comprehensibility | Decomposing big chunk into smaller chunks with well-defined APIs
- Types of Cohesion:** Functional, Layer, Communicational, Sequential, Procedural, Temporal, Utility
- Types of Coupling:** Content, Global variables, Control, Data, External, Temporal, Inclusion/import

3.4 Architectural Styles

- How code is divided:**
 - Technical Partitioning:** Focus on separation of concerns (Eg. Presentation, Services, Persistence)
 - Domain Partitioning:** Aligned with the domain (Eg. Customer, Shipping, Payment)
- How is it deployed:**
 - Monolithic:** Deploy all logical components that make up the application as 1 unit, application runs as 1 process
 - Distributed:** Application consists of independent logical components, logical components run as individual processes, communicate over network
- Types:**
 - Layered (Technical & Monolith):**
 - Lower level layers provides functionality for higher level layers
 - Open layer architecture:** 1 layer can talk to another layer, which can be layers away
 - Close layer architecture:** 1 layer can only talk to a neighboring layer
 - More layers result in scalability by allowing each layer to run in different server, but communication becomes expensive
 - Less layers result in performance optimization, no context-switching overheads, but difficult to modify, impractical
 - Modular Monolith (Domain & Monolith):** Separation by domain (business) concern, reduces coupling (Eg. Order domain, recipe domain, where each domain consists of presentation, logic & persistence layers)
 - Event-driven (Technical & Distributed):**
 - Microservices (Domain & Distributed):**
 - Pipe & Filter:**
 - Data enters system and flows through components one at a time until data is assigned to some final destination (Data Sink)
 - Components consist of *Filters*, *Data Source*, *Data Sink*, where each component can read and produce
 - Filter* transforms input streams, computes incrementally → output begins before input is consumed, is independent, shares no state with other filters

- Pipe* transmits output of 1 filter to input of another filter
- Purpose:** Divides the app's task into several self-contained data process steps & connect these steps to data processing pipeline via intermediate data buffers → Data flows in streams (good for image, audio, video, or batch data processing with limited user interaction)
- 6. Model-View-Controller**
 - View (widgets in UI, buttons, text boxes), Controller (coordinates btw. Model and View), Model (business logic)
 - Benefits:** Separation of concerns, Facilitates extensibility, Restricted communication reduces complexity & side effects, Better testability (easy to mock components), Frameworks provide MVC solution
 - Web MVC:** Controller (handles user HTTP requests, select model, prepare view), View (renders HTTP response), Model (business logic & persistence)
 - Single Page Applications (SPA):** Send query and retrieve data in background without refreshing webpage

3.5 REST Architecture

(Not an architecture by itself)

- Definition:** Defines constraints for transferring, accessing, and manipulating textual data representations in a stateless manner across a network of systems → Provide uniform interoperability between different applications on the internet (HTTP to request access and use data)
- Constraints:**
 - Client-Server:** REST apps should have client-server architecture for separation of concerns → Improve portability of UI & scalability of server components
 - Stateless:** No client state (session) maintained on server → Server is bound by no. of concurrent requests & not the no. of clients interacting → Improve scalability, reliability, monitoring
 - Cache-able:** Response from server should include if data is cache-able or not → Client returns data from its cache in response to subsequent requests → Improves network efficiency, but client can potentially receive stale data
 - Layered System:** App must be organized as a layered system → Improved overall system complexity by restricting complexity to individual layers, intermediary servers may improve system availability & performance, provide data transformation & filtering
 - Uniform Interface:** Uniform way of interacting with a given server irrespective of device/application, exploits HTTP/HTTPS requests & responses
 - Resource Identifier:** Stable, global, unique method to identify resources
 - Resource Representation:** Components perform operations on resource representations
 - Code-on-demand:** Allow client functionality to be extended by downloading executable code, simplifies client from having to pre-implement all functionality, allows extensibility
- Advantages of REST:** Systems are less tightly coupled, provides scalability, usability, accessibility
- Disadvantages of REST:** Being stateless decrease network performance by increasing repetitive data sent in series of queries | Using URI degrade efficiency since info is transferred in standardized form rather than one which is specific to app's needs

4 Microservices Architecture

4.1 Microservices Architectural Style

- A single application as a suite of small services
- Each microservice offers a well-defined business capability (Features in microservice should be highly related)
- Each microservice is defined to be developed & deployed independently (Services do not need to share any of their code or implementation with other services → minimal dependency on each other, services are owned by different teams)
- Microservices communicate with each other through well-defined mechanisms (Synchronous: RESTful APIs, gRPC, GraphQL, Asynchronous: Event-based publish & subscribe)

4.2 Domain Driven Design

- Complex system is fundamentally a collection of multiple domain models (sub-domains)
- Domain:** Problem space that a business occupies and provides solution to
- Sub-domain:** Component of main domain, belongs to problem space
- Bounded Context:** High cohesive boundary relevant to the sub-domain → Belongs to solution space
- Types of Collaborations between different Bounded Contexts (BC):**
 - Shared Kernel:** 2 contexts are developed independently, but they end up overlapping some subset of each other's domain
 - Upstream-Downstream:** 2 contexts are in provider (upstream) - consumer (downstream) relationship
 - Supplier-Customer Relationship:** Supplier is the BC that provides a service, functionality or data to another BC, while Customer is the BC that consumes the functionality/data (Eg. Order Context supplies order data, Recommendation Context fetches the order data)
 - Conformist Relationship:** 1 BC (the conformist) fully adopts the model of another BC without trying to impose its own requirements or interpretations (Eg. Payment service must align entirely with the bank's API and data structures, regardless of whether it fits the payment service's internal domain model or not)
- Aggregate:** A cluster of related objects that we treat as a single unit for purpose of data changes (Has *transactional boundary*, which means any changes to aggregate will either all succeed or none will succeed | Has *consistency boundary*, which means all processes or objects external to aggregate are only allowed to read aggregate's state & its state can only be modified by executing corresponding methods of aggregate's public interface)

- Aggregate Root:** Parent entity of the aggregate → designated as aggregate's public interface

4.3 Event Storming

- Command** causes **Events** (Eg. User/external system issues a command)
- We treat **Aggregate** as a unit for the purpose data changes → Eg. Order Processing Aggregate consists of Confirm order Command & Order confirmed Event
- Policy:** WHEN event THEN command (basically links 2 aggregates together, Eg. When payment card is submitted event happens, then execute confirm order command)
- A BC solves 1 problem & can contain more than 1 aggregate
- 2 different aggregates can be created to solve 1 problem delimited by 1 BC

