Container vs Orchestrator
-Containers provide the platform for building and distributing services
-Orchestrator works w/ containers for production
Serverless - leaner than containers
-Developers package the code in containers
(on demand)
-Cloud provider manages physical servers
-Dynamic allocation of resources
-Works based on Func-as-a-service
Factors Affecting Software Development
-Requirements
-Process (Resources, Time)
-Criticality, Consequences
-People (Competence), Technology
CI/CD Pipeline
-CI: Build, Unit tests, Deploy, Acceptance Tests
-CDelivery: Extra deploy to production (manual)
-CDeployment: Extra deploy to production (auto)
-Benefits: Low-risk release, early feedback, faster market, higher quality, lower cost
DevOps
-Combine development and operations.
-Reduce the time between change a sys and change in normal performance to ensure high quality
-Benefits: Speed of Delivery, Reliability, Scale, Improved collaboration
CH2: Requirement
-Capability needed by a user to solve problems
-Capability must be met or possessed by a sys
-Documented Representation of condition/capability
-Phases: Elicitation, Analysis, Specification, Validation
-Sources: Doc, interviews, surveys, event-response tables, prototyping, observation
-Outcome: SRS, req under change control, <Rights, responsibilities, and agreements>
-**SRS components**: Interfaces, func capabilities, performance levels, data structures/elements, safety, reliability, security/privacy, quality, constraints, and limitations
-**SRS**: tells what work is to be done vs Product backlog: Repo of the work to be done
-**Quality of strong SRS**: Accurate, Complete, Modifiable, Ranked, Testable, Traceable, Unambiguous, Verifiable, Valid
-**Requirements**: Business, User, System, Quality,
FR, NFR, Constraints, Data
-Business req: why the organization is implementing the system
Software Development Process
-Reduce risk of failure, Framework for project planning and execution, Divide software development work into phases to improve.
-Software Quality Attri: Availability Performance Efficiency Scalability Robustness Safety Security Reliability Integrity Verifiability Deployability Compatibility Installability Portability Maintainability Usability Testability Modifiability Reusability Interoperability
-**Security Requirements in SRS**: Business owners. Typical Security Requirements: The authentication requirements. Auditing and logging requirements. Intrusion Monitoring requirements.
-**Safety vs Security**: Safety, it is about whether a system can harm someone or something. Security, is about privacy, authentication, and integrity.
-**Performance:** Encompass the responsiveness of the system. Low performance => unhappy user.
-Impacts safety of the system.
-Requirements affect the choice of architecture, design and deployment.
-Affect choice of hardware and type of network necessary.
-**Scalability**: Hardware (eg) increased incoming data would mean adding disk capacity. Software scaling (eg) to accommodate an increased number of transactions.
-**Vertical scaling** refers to increasing the capacity of a system by adding capability to the machines
-**Horizontal scaling** refers to increasing the capacity of a system by adding additional machines
-Microservices are small, focused and independently deployable => good for horizontal scaling
-**Availability:** Measures the planned uptime of the system. Impact cost of deploying the software and complexity of the software design.
-**Usability:** Measures the effort required to prepare input, operate, and decipher the output of the software. Ease of learning and ease of use.
Requirement Traceability
-Each requirement has a type, a unique identifier, a brief high level description, priority and other information.
- Req could be traced to use case ref, design document ref, code module ref, test case ref and others.
-**Validation:** Whether u have written the right req; do they trace back to business objectives?
-**Verification**: Whether u have written the req right; req have the desirable properties, eg., completeness, correctness, feasibility, prioritization, unambiguity, etc.
-How is it done? Informal: Peer desk-check, pass-around, walkthrough. Formal: Inspection, formal process, checklist.
-**Documenting req:** Textual: Vision & scope doc, use case doc, SRS. Visual: Structured analysis models, diagrams and analysis models. Recollect: Feature list, user story.
Software Architecture
-Represents the structure of data and program components that are required to build a software
-Reference architecture & architectural patterns: Common architectural framework -> Numerous app, same framework. Explains high-level structures of similar applications. Lead to architectural patterns -> Architectural solution forms the basis for architectural design, addresses applications-specific problem within a specific context, adheres to a set of limitations and constraints.
-**Control flow:** Reasoning is on the computation order. How the focus of control moves throughout the execution data may accompany control.
-Data flow: Reasoning is on the data availability, transformation, latency. How data moves through a collection of computations. As data moves, control is activated.
-Call and Return: Control moves from one component to another and back. Can be hierarchical or non-hierarchical.
-Message and Event: A message is some data sent to a specific address. An event is some data emitted from a component for anyone listening to consume.
-**How to break architecture up?**
Divide and conquer, keep abstraction as high as possible, increase cohesion, reduce coupling, reusability, flexibility.
-**Decomposition, Componentizing & Packaging**: Horizontal slicing – designing by layers. Vertical slicing - designing by feature.
-**Principle of Modularity**: Modularization => shorter development time, better flexibility & better comprehensibility. Decomposing a big-chunk into smaller chunks with APIs. Manage the complexity of a problem by breaking them down to smaller manageable modules.
-Layered Architecture: software is organized as layers of components. Supports independent development and evolution of diff system parts. Comprises one or more layers for the software under development with each layer having a distinct and specific responsibility.
-**Pipe and Filter:** Data enters the sys and flows through the **Components** one at a time until the data is assigned to some final dest (**data sink**)
-Components: Filters, Data Source, Data Sink
-Filter: Transforms the input streams, computes incrementally so output begins before the input is consumed, independent, share no state with other filters
-Connectors: **Pipes** transmits outputs of one filter to another input.
-Divide the application's task into several self-contained data process steps and connect these steps via intermediate data buffers. Dataflows in streams. Good for limited user interaction, like batch processing systems.
-**Model View Controller (MVC):** Goal: Support user's mental model of relevant info, Enable user to inspect & edit this info.
-View & Controller roles may played by the same object when tightly coupled.
-**Benefits:** Separation of Concerns-results in modularity, Facilitates extensibility, Restricted communication reduces complexity and side effects, better testability, frameworks provide MVC solution.
-**Web MVC**: Two communicating entities in Web app: Sever (hold model), Client (interact w/ server)
=> Controller: Responsible for handling user HTTP req, select model, prepare view. Additional responsibilities: Map requests to specific handlers. Page controller, Front controller(Middleware?).
=> View: renders the HTTP response, May use a template to render contents of model
=> Model: Business logic + persistence
-**Single page app:** Adv: Fluid, save bandwidth, reduce latency. Disadv: JS, harder to develop, vulnerable to XSS attk.
-Architecture Diagrams: Big pic of what is built, map that Software Dev us to navigate.
-Software sys, highest lvl of abstraction.
-Container is a context or boundary inside
-Component grps related functionality encapsulated behind an API.
Rest Architectural Style
-Defines contrstaints for transf, access, manupulating text data representation in a stateless manner across a system.
-Intention: Provide uniform interoperability, between different applications on the internet
-**Rest Archi Constraints:**
=>**Client-server**: Rest apps should have client-server architecture. Reason: separation of concerns -> Improve portability of user interface, scalability of server components.
=>**Stateless**: Interaction btwn client & server is self contained. No client state maintained on server. Server bound by number of concurrent req, not number of clients. Reason: To improve scalability, reliability, monitoring
=>**Cacheable:** Response from server include if data is cacheable or not. Client returns data from its cache in response to subsequent equivalent req. Reason: To improve network efficiency. (-ve) Client potentially receive stale data.
=>**Layered sys:** App must be organized as layered sys. Improved overall sys complexity by restricting complexity to individual layers. Intermediary servers may improve sys availability & performance. Provide data transformation & filtering.
=>**Uniform Interface:** Generality to the component interface. Efficient for large-grained hypermedia data transfer. Exploits HTTP/HTTPS requests and responses. Anything can be a **Resource**. **Resource** is a conceptual mapping to a set of entities. Identified using stable, global & unique resource identifiers. Components perform operations on resource representations. REST needs self-describing messages for resource interactions. Components get complete understanding of resources or relevant states thru inspecting representations. Hypermedia driving app state: representations that are exchanged should be linked.
=>**Code on Demand:** (Optional) Allow client functionality to extend by downloading exec code. Simplifies the client from having to pre-implement all func. Allows extensibility.
=>**Adv & disadv REST:** Sys are less tightly coupled. Provides scalability, usability, accessibility and mashup ability. Stateless may decrease network performance by increasing the repetitive data sent in series of queries. URI may degrade efficiency, since info is transf in a standardized form (not specific to app needs)
Microservice
-A set of software app designed for limited scope that work with each other to form a bigger soln. Each is minimal but well-defined capabilities for creating a modularized overall architecture. Microsvc is an independent capability which separate func into collection of independent svc. Benefits: easier deployment, testing and maintenance.
-**What is 'independent':** Svc do not need share code or implementation. Communication via APIs. Each svc developed, deployed, operated and scaled w/o affecting the func of other svc.
-**Microsvc Characteristics**:
=>**Organised around business capabilities**
=>**Loosely coupled**: Reduce implementation coupling, reduce domain coupling
=>**Owened by small teams:** Small team led to loosely coupled microsvc
=>**Independently Deployable:** Each microsvc has its own specific deployment, resource, scaling and monitoring req. Each service instance must be provided with appropriate CPU, mem and I/O resources. Run multiple instances of different svc on a host.
->**Pattern**: **Svc instance per host** run each svc instance in isolation on its own host. **Svc instance per VM** each svc instance is a VM launched using that VM image. **Svc instance**

per container each svc instance runs in its own container. A container image is a filesys image consisting of the app and lib required to run the svc.
-**Right size for microsvc?**
=>**Domain model:** Critical & fundamental or foundational concept behind business.
=>**Domain Driven Design(DDD)** suggests that model be kept isolated from technical complexities.
-**Ubiquitous Lang:** Shared lang between domain experts & developers. Find "in-context" and "out-context". Find scnarios in the context.
-**Sub-domains**: Specific responsibilities and reflects some of the business organisation's struct.
Problem space. SOC
-**Bounded context**: Include input, output, events, req, processes and data models. **Solution space. SOC & SRP & Cohesive**. Define business sub-domainds
-Aggregates: define business entity models.
-**Identify microsvc:** Apply SRP. Decompose by verb or use case, by nouns/resources/by defining a service that is responsible for all operaations on entities/resources. Consider factors such as team size, tech, scalability req, availability req and security req.
–Initial Goal – identify/ reduce number of services to work with based on business capabilities or sub-domains
–Target services that encompass entire a Bounded Context
–breaking these services into smaller services / split them e.g. around Aggregate boundaries
–A microservice is defined in a 'bounded' sense meaning its function is narrowly defined.
–It is essential to organize microservices in terms of high-level domain vocabulary (who they serve) and low-level domain vocabulary (what they do).
-**Database per svc pattern:** Each svc has its own database schema. Use a type of database that is best suited to its needs.
Service communication
-**Inter process comm:** Request/sync response: client makes a req to a service and waits for a response. Notification: client sends a req to a service but no reply is expected or sent. Request/async response: client sends a req to a svc, which replies async.
-Service Discovery:
=>**Client-side Discovery Pattern:** Client queries svc registry. Client determines network locations of available svc instances.
=>**Server-side Discovery Pattern:** Client makes a req to a svc via load balancer. Load balancer queries svc registry and routes each req to an available svc instance.
=>**Service Registry Pattern:** Database of svc, their instances and their locations. Both pattern above.
Event
-Describe some action/fact, notifications of some change in state/data, recordings of incidents.
-Unkeyed: No key val. Entity: Track the ID. Keyed: Has key.
Event-Driven Architecture (EDA)
-Decoupled applications can asynchronously communicate by issuing and consuming events via an event broker. Loose coupling. Event producers, brokers, consumers.
-**Adv:** Can decouple sys (loose coupling) that can be independently scaled & updated, handle high vol of data with low latency, support realtime processing & analytics, more scalable and resilient to failures. Real-time data.
-**Event-Driven Microsvc**: Connected as interconnected graph. Producers: microsvc producing info. Consumers: svc rely on data given.
-Benefits: Granularity, scalability, tech flexibility, business req flexibility, loosely coupling, continuous delivery support, high testability
-**Event Sourcing:** events stored in an append-only log. Current state derived by applying historical events to initial state.
Command-Query-Responsibility Segregation (CQRS)
-Separate write and read path. State of DB recovered from log.
-**Adv:** One write model can push data into many read models. Read model can be in any database.
-Pattern rely on separation of commands from queries. (Use same/diff DB)
- SoC, SRP, Interface Segregation Principle
Remote-Procedure call Synch
-Inter-process communication that allows program to make procedure execute in another address space
-**Asynchronity in Req-reply:** Decouple backend processing from reqter, backend needs async but reqter needs clear response.
Async message passing:
-Loosely coupled sys
-**Single Receiver:** Msg q like a buffer. Receive and put msg on q.
-**AMQP:** Peer to peer protocol. Broker either deliver msg to subs to the q or subs fetch/pull msg from q.
-**RabbitMQ**: Direct: Msg route to q whose binding key matches key of msg. Fanout: Routes msg to all q. Topic: Wildcard match btwn routing key and pattern.
-**Multi receiver:** Msg published to a topic. Topic = broadcasting station. Use single centralized sys that manage pub/sub.
-**Kafka:** Unit = msg. Category = topics -> Broke down into partitions. Msg produced with key. Append msg to partition. Consumer sub to one or more topics and read in order. Single consumer fail, remain member of grp rebalance partitions to take over for missing member. Zookeeper: Manage brokers, communicate with each broker and get its health status.
-**Persistent comm:** msg stored at each immediate hop along the way until next node is ready to take msg
-**Transient comm:** msg are buffered only for small periods of time. Cannot delivered or host down, will be discarded.
-**Combi**: Persistent Async, Persistent sync, Transient Async (not guaranteed), Transient sync(Receipt, delivery, response)
Msg passing
-**Async:** Msg q sys support async persistent comm. Comm by inserting msg in q. No guarantee on when or if msg will be read. (Kafka, RabbitMQ, ActiveMQ)
-**Java Msg Svc**: msg standard allows app components to create, send, receive and read msg. Loosely coupled, reliable and async.
-**Amazon SQS**: Store copies of msges on multiple servers for redundancy & high availability. Lifecycle: 1min – 14 days. 4days.
-**Enterprise integration patterns**: collections of tech-independent soln to common integration prob. EIPs are vendor-independent.
=> Msg channel. Point 2 point channel. Pubsub channel. Msg. Msg Endpoint.
-**Message:** Encapsulate method req & data struct. (Header, payload)
-**Command msg:** Specify Func or method on the receiver
-**Doc msg:** Enable sender transmit one of its data struct to receiver.
-**Event msg:** Notify the receiver of a change in sender.
-**Msg channel (Msg q):** Connect collaborating senders and receivers using a msg channel.
-**Req/Reply Channels:** Requestor: Sends a req msg and wait for a reply msg. Replier: Receives req msg and responds with reply msg. Channel transmits msges in 1 dir. 2 way msges need 2 way channel.
-**ReqQ:** Q the requester uses to send req msg.
-**ReplyQ:** Q the replier uses to send the reply msg.
-**InvalidMsges:** Q that the requestor and replier move a msg to when msg unable to interpret.
-**Return addr:** Tell replier where to send reply to. Correlation ID specify which req this reply is for.
-**Desc of Corr ID:** Requestor, replier, req, reply, req ID, corr ID.
-**Req-reply chaining**: Useful if app retrace the path of msges from last reply to original req.
-**P2P Channel:**
=> One way & req response: Pubsub used when multi parties are interested in some msg. **Beneficial:** where it's important to comm with multi svc that do work in parallel but all responses need to be aggregated afterward.
-**Invalid Msg Channel:** Msg that makes no sense
-**Dead letter channel**: Msg where msg sys cannot deliver
-**Data Type Channel:** Sender sends a data item such that receiver know how to process it. (Choose appropriate channel to send msg).
Msg Router
-Consume msg from one msg channel and reinsert them into diff msg channels depending on cond.
-**Simple Routers:** Route msg from one inbound channel to one or more outbound channels.
-**Composed Routers:** Combine multi simple routers to create more complex msg flows.
-**Content-Based Router:** examines the msg content and routes the msg onto diff channel based on data in msg. Knowledge of all possible recipients and capabilities. Change in recipient list -> change in content-based router.
-**Msg filter:** Only single output channel. Otherwise, discarded.
-**Context-based Router**: Decide msg dest based on specific contexts. Perform load balancing, test or failover func.
-**SNS topics**: Broadcast, subset, filter policy
Message splitter:
-Single msg -> multi msg
Message Aggregator:
-Multi msg -> Single msg
-Identify correlated msges and publish a single, aggregated msg to output channel.
Message Scatter-Gather:
-routes/broadcasts a single msg to no of participants concurrently & reassemble/aggregates the replies into a single msg.
-Ideal for req responses from multiple parties, aggregating and processing the data.
-equiv to **Broadcast + aggregate**
Message Translator:
-Msg translator btwn client and svc converts msg to another format.
-Resolves differences in msg formats w/o changing app or having app know other data format
-**Canonical Data Model:** provide additional lvl of indirection btwn app individual data formats.
Message Endpoints:
-interface btwn an app and msg sys
-send/receive msg but one instance does not do both. Multi endpoints to interface w/ multi channels.
-**Consumer endpoints:** Polling: control when consumes each msg. Event-Driven: event that triggers the receiver into action
Modularity:
-Manage complexity. Separates func of a program into independent modules.
-**Modular Monolith**: composed of loosely coupled, highly cohesive modules. **Benefits:** Independent modules – easier to develop, test and maintain and reusable in other applications.
-**Common principles:** High Cohesion, Loose coupling
-**Shouldn't be module:** Too small, cant be described, depending on code size.
-Design smell: If component has lots of responsibilities than other classes, make it hard to reuse, maintain and test.
-Program to Interface (P2I): Decoupling. Or use abstract class.
-**Favor composition over inheritance:** "has-a relationship preferred over is-a relationship"
What is a Design Pattern
-A soln to a prob in a context.
-**Context**: situation in which the pattern applies
-**Problem:** goal to be achieved in the context including constraints.
-**Soln:** Is pattern. General design.
-**Patterns consist:** Name, classification, intent, participants, implementation code.
-**GoF patterns: Creational Patterns:** handle issues with object creation. Instantiate single object or groups of related objects. **Structural Patterns:** provide a structure to the relationship between objects. Relationships btwn classes or objects to form larger structures. **Behavioral Patterns:** help define how objects interact with each other to deliver a task. Define manners of comm btwn classes and objects.