## Software Development Process

**Centralized vs Decentralized Computing** If latency matters -> Computation at edge(device). If computational power matters -> Computation at cloud servers(centralized).
**Containers(eg Docker)** provide platform for building and distributing services e.g. Microservices. Need multiple containers to run a complex application with each providing specific service.
**Orchestrator(eg Kubernetes)** integrates and coordinates the containers. Able to scale up/down deployment based on demand, provide fault tolerance and provide communication among containers.
**Serverless(eg AWS Lambda)** are cloud-native development models, app responds to demand and automatically scales up/down when needed. Managed by cloud provider.
**Agile Process** for iterative development of application and cross-functional collaboration.
**Scrum** is an example of Agile framework. Work is done in sprints, where subset of product backlog is cleared. Often includes a daily 15min meeting.

### Software Delivery

**DevOps** is a set of software development practices that combine software development and operations. Intended to reduce the time between committing change to a system and the change being placed into normal production, while ensuring high quality.
**Continuous Integration(CI):** Auto build, unit tests, deploy to staging, acceptance tests.
**Continuous Delivery(CDel):** CI + **manual** deployment to production
**Continuous Deployment(CDep):** CI + **auto** deployment to production
**DevOps benefits:**

1. Speed of delivery - quick release new features and fix bugs
2. Reliability - Change is functional
3. Scalability of app
4. Improved Collaboration

## Specifying Software Requirements

**Requirement** is a capability needed by a user, or must be met by a system, ie specification of what should be implemented.
**Software Requirement Specification(SRS)** has 9 topics: Interfaces, Functional Capabilities, Performance Levels, Data Structures/Elements, Safety, Reliability, Security/Privacy, Quality, Constraints and Limitations
**Product Backlog** Repository of work to be done, facilitates planning and prioritisation of work
**Types of Requirements:**
**Business requirements** Why organisation is implementing system
**User requirements** Goals or tasks user must be able to perform with product
**System requirements**
**Quality requirements**
**Functional requirements(FR)** Behaviour product will exhibit.
**Non-Functional requirements(NFR)/Constraints** How well system works eg response time < 5s
**Data requirements**

### Quality Attributes:

**External** Observed when software executing, impacts user's experience of using software and develops user's perception of software quality.
Includes: Availability, Performance, Robustness, Safety, Security, Reliability, Integrity, Deployability, Compatibility, Installability, Usability, Interoperability
**Internal** Not directly observed, perceived by devs and maintainers, encompasses aspect of design that may not impact external attributes
Includes: Efficiency, Scalability, Verifiability, Portability, Maintainability, Testability, Modifiability, Reusability.

### Requirement Validation & Verification

**Validation** Written the right requirements, trace back to business objectives
**Verification** Written the requirements right, requirements have the desirable properties. Checked informally by passing reqs around or formally by formal inspection.

## Software Architecture

**Building Blocks of Architecture:**

- Configuration - Topology or structure
- Component - Element that models an application specific function
- Connector - Element that models interactions among components for transfer of control and/or data

**Definitions:**
**Control Flow** Reasoning based on computation order Data Flow Reasoning based on data availability, transformation, latency
**Call and Return** Control moves from one component to another and back
**Message** data sent to specific address
**Event** data emitted from a component (Listener to pick up)
**Horizontal slicing** Designing architecture by layers
**Vertical slicing** Designing architecture by feature
**Principle of Modularity** Allocate different functions to software modules and specify APIs for interaction(eg Microservices)
**Types of Cohesion:**

- **Functional** - one computation, no side effects
- **Layer** - related services kept tgt, strict hierarchy, higher lvl svcs can access only lower lvl svcs. **Accessing svc may result in side effects**
- **Communicational** - Operate same data kept tgt
- **Sequential** - Work in sequence for some computation kept tgt, **output from one is input to next**
- **Procedural** - Called one after another kept tgt
- **Temporal** - Used in same general phase of exec kept tgt (eg initialisation)
- **Utility** - Related utilities kept tgt

**Types of Coupling:**

- **Content** - Component modifies internal data of another
- **Common/global** - Using global vars; all modules using global var are coupled
- **Control** - A directly controls B by passing info on what B should do(e.g using a "flag" to tell B what to do)
- **Data** - One module sharing data with another module(eg passing params)
- **External** - Dependency exists on elements outside scope; eg OS, shared libraries or hardware
- **Temporal** - Two actions bundled tgt bcs occur at same time
- **Inclusion/import** - Importing of packages to use

**Types of Architecture:**

- **Layered Architecture** - Strict hierarchy present, each layer have a distinct and specific responsibility
- **Pipe and Filter** - Divide app into several self-contained data process steps and connect to a data processing pipeline via intermediate data buffers. **Good for limited user interaction like batch processing systems**.
- **Model view controller(MVC)** - View for UI, Controller to coordinate btwn Model and View, Model for business logic. Benefits: Separation of Concerns(SoC) -> modularity, facilitates extensibility, reduce complexity and side effects, better testability.
- **Web-MVC** - Server: Model, Client: Perform requests. Controller: Handle user http req, select model, prepare view. View: Render http response. Model: Business logic + **persistence**
- **SinglePageApp(SPA)** - JS program on browser. Send and retrieve w/o refreshing webpage

## Rest API

**REpresentational State Transfer(REST)** Intention: provide uniform interoperability betwn diff apps on internet. Exploit HTTP reqs to access and use data.
**Constraints:**

- **Client-Server** Client requests for resources and server stores resources. Satisfies SoC.
- **Stateless** Server does not keep track of client state. Full info contained as part of query params. Allows for scalability, reliability and monitoring.
- **Cacheable** Server caches data retrieved for efficiency of subseq reqs. Downside: Stale data retrieved
- **Layered system** Hierarchical structure, except each layer cannot see beyond immediate neighbour
- **Uniform interface** Interoperability feature. Generality to component interface, eg HTTP reqs. Use unique resource identifiers to obtain resource.
- **Code-on-demand** Allow downloading of executable code eg Javascript to simplify client from pre-implementing all functionality.

## Microservices

**Microservices** are set of software apps designed for limited scope that work with each other to form a bigger soln. each has **well-defined capabilities for modularity purposes**. Services are independent of one another. Benefits: Easier deployment,testing,maintenance.
**Characteristics:**

- Organised arnd business capabilities(Problem domain)
- Loosely coupled
- Owned by small team
- Indepedently deployable (Service instance per host(SIPhost) or SIPcontainer)

**Domain Driven Design(DDD):**
**Ubiquitous language** Shared language btwn domain experts and devs. Find scenarios in context, mix of biz and technical jargon.
**Subdomains** Partitions of problem space
**Bounded Context** Partitions of solution space, are technical solutions to problem. Mostly relevant to the subdomain attached.
**Aggregate** Cluster of related objects(eg Ticket,Message,Attachment). Has transactional and consistency boundary.

- Changes to aggregate either all succeed or all fail - transactional
- Modified only through public interface, otherwise read only - consistency

**Aggregate Root** Parent of cluster, designated as the agg's public interface
**Key Feature of Microservice** Single Responsibility Principle(SRP). Each microservice only responsible for all operations on entities of a given type.
**Identifying Microservices:**

- Use sub-domain to define services
- Microservice are **Bounded Contexts**
- Use aggregates - May exhibit many characteristics of a microservice(Aggregates have high functional cohesion and are loosely coupled)

**Microservice Concepts:**
**Database per service pattern** Each service has its own database schema
**Service communication** IPC either synchronous or async: Request/sync response, Notification(one-way request), Request/async response.
**API gateway** Single point of entry to system, encapsulates internal architecture
**Orchestration vs Choreography** Orchestration is when one single entity guides and drives processes**(sync)**. Choreography is when it is event driven and responsibility is on receiver instead**(async)**.

By: Khoo Jing Hong, Derrick

**Service Discovery**

- **Client-side discovery:** Client queries a service registry
- **Server-side discovery:** Client requests via load balancer instead and load balancer queries service registry
- **Service registry pattern:** Service instances are registered with service registry on startup, deregistered on shutdown.

## Event Driven Architecture

**Events** are means of async communications between services, typically in key-value format. Are immutable in nature and in temporal order.

**Types of events:**

- **Unkeyed** Only value present
- **Entity** Ensures unique ID(key) present
- **Keyed** Same key can be used to map to diff values

**Typical structure** Source -> Event generator(producer) -> Event bus(broker) -> Event listener(consumer)

**Key components** Event producers, Event brokers, Event consumers

**Event Driven Architecture(EDA):** is when services are notifications based. Each service has no knowledge that another service exists.

**Event Driven Communication** is asynchronous, works with real-time data. Best suited for cases where components need to react to changes in state/real time data processing is required.

**Benefits of Event Driven Microservices:**

- **Granularity** Easily rewritten if biz requirements change
- **Scalability** Individual services can be scaled up/down as required
- **Technological flexibility** Each service can use its own language or technology
- **Biz requirement flexibility** Low cross-team dependency
- ****Loose coupling** ED microservices rely on domain data and not on specific implementation APIs
- **Continuous Delivery support** Easy to ship small and modular microservices
- **High testability** Few dependencies, easy to mock out testing endpoints to ensure coverage

**Event Sourcing:**

**Storing events** Events are stored in an **event log** in the order they are created in. We are able to rederive the current state of system by rewinding the log and replaying events in order.

**Key point to note** Event log is **append-only**. No overwriting events in event log.

**Command-Query-Responsbility Segregation(CQRS):** Separation of write path from read path. Writes go into **Kafka** on command side(rather than updating DB directly). Supports materialising view(table that contains predefined queries) for performance optimisation for reads. **Utilises underlying pattern: Separation of commands from queries.**

**Commands** are operations that change application state and return no data.

**Queries** are operations that return data but dont change application state.

**Kafka** is an in-memory data structure that can hold events or are general-purpose distributed data stores capable of handling event sourcing.

## Communication Patterns

**Sync vs Async communication:**

- Sync: Caller waits on receiver reply
- Async: Caller continues execution and does not wait on reply (eg Advanced Message Queue Protocol(AMQP) - RabbitMQ)

**Synchronous Request-Reply:**

- Remote-Procedure call (Code not in same address space, remote execution) - commonly on another computer on a shared network
- Sequence: Client calls **client stub**. **Stub** packs parameters into a message**(marshalling)** and syscall to send msg. Client OS -> Server OS. Server OS passes incoming packets to **server stub**. **Server stub** unpacks msg**(unmarshalling)**, then calls procedure.
- Another example is **HTTP req/response**.

**Async Request-Reply:** Although client does not wait, client still expects response eventually.

**Async Message Passing:** Usage of **Message Queue (Single Receiver)** or **Pub-sub pattern(Multi Receiver)**

- AMQP: Peer-to-peer protocol
  - Message published to exchanges(mailboxes), exchanges then distribute message copies to queues. Then, broker(queue) either delivers to subscribers or consumers fetch from queues on demand.
  - **Binding** Used to bind a queue to an exchange
  - **Routing key** Message attribute used by exchange to decide how to route msg to queues(depending on exchange type)
  - **Exchange types:**
    - Direct - Binding key == routing key
    - Fanout - Send to all queues bound to it
    - Topic - Wildcard matching between routing key and routing pattern specified in binding
- Pub-Sub pattern (Message not specifically directed)
  - Message published to a topic(broadcast station), consumers can subscribe to topic to receive message.
  - Usage of brokers - central point where messages are published
    - e.g **Kafka**. **Zookeeper** is used to manage all the brokers in Kafka.
- **Partitions** A topic can be broken down into a number of partitions. Idea here:**Consumer Group**(multiple consumers),ea partition consumed by 1 cons, msgs written to part w given key

**Persistent vs Transient communication:**

- Persistent: Async case - sender guaranteed that message will eventually be inserted in recipient's queue. **No guarantee on when though.**
- Transient: Message buffered only for small periods of time. If msg cannot be delivered or next host is down, msg is discarded. Sync case - Receipt-based(ack when received), Delivery-based(ack when delivered, longer than receipt), Response-based(ack when processed).

## Messaging Patterns

**Message Construction.** Message contains:

- Header: Specify type of info - command(tell receiver what to do), document(pass data but does not tell receiver what to do), event(notification)
- Payload: Actual information

**Message Channel(Message Queue).** Channel connects the collaborating senders and receivers using a message channel that allows them to exchange messages. Channel transmits messages in one direction. Two-way messages needs two-way channel.

**Request/Reply channel:** Requestor-Replier. Needs two queues RequestQueue() and ReplyQueue()

**Return Address:** Request contains a return address to tell the replier where to send reply to

**Correlation ID:** Message ID of received message. Specifies which request the reply is for.

**Point to Point(P2P):** Request processed by single consumer(single receiver)

**Publish-subscribe channel:** Request broadcasted to all interested parties(multi receivers). Pub-sub can be extended to request-response msging, where requests use pub-sub channel, and response is aggregated and on a P2P channel.

**Invalid Message Channel:** Receiver end, queue to handle cases where msg cannot be interpreted

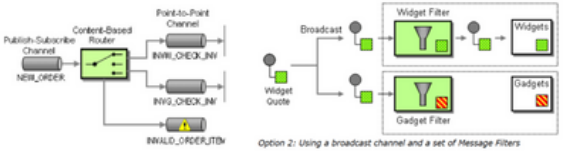
**Dead Letter Channel:** Broker end, queue to handle cases where message cannot be delivered(eg receiver down)

**Data Type Channel:** Separate channels for each data type (e.g XML, byte-array etc)

**Message Routing.** Message routers consume messages from 1 msg channel and reinsert them into diff msg channels, depending on a set of conditions.

**Simple routers** Route msgs from 1 inbound channel to one or more outbound channels

**Complex routers** Combine multiple simple routers to create more complex msg flows.



Option 2: Using a broadcast channel and a set of Message Filters

**Content-Based Router** Examines msg content and route msg onto diff channel based on data contained in msg. **Needs to have knowledge of all possible recipients and capabilities.**

- **Message Filter** Special kind of content based router, used to send messages to intended audience in pub-sub channel when broadcasted. If msg content matches criteria specified by msg filter, msg is routed to output channel, otherwise it is discarded.

**Context-Based Router** Decides msg destination based on specific contexts, eg load balancing.

**Attribute based filtering(subscriber filter)** Instead of publisher being responsible for routing, can require publisher to set **message attributes** and each subscriber to set a subscription attribute(sub filter policy).

**Message Splitter** Used to split single msg to multiple msges.


**Message Aggregator** Used to combine multiple msgs into a single msg.

**Message Scatter-Gather** Basically broadcast + aggregate. Routes/broadcasts a single msg to a number of participants concurrently and aggregates replies into a single msg.

**Sample seq for activity diagram**

Fork = pub-sub channel
Join = P2P channel
Diamond = Aggregator, followed by content-based routing

**Message Transformation.**

**Message Translator** Used to convert msg from one data format to another (eg XML to byte-array)


**Canonical Data Model** We can reduce number of translators needed for large apps. Idea is to first convert to a standard syntax(canonical format or common format), then second system converts from canonical format to its own data format.

**Message endpoint** Interface betwn app and messaging system. Is channel specific.


Can only be used to send messages or ceive, cannot do both

Requires multiple endpoints to interface w multiple channels.

**Polling consumer** Proactively reads msgs **when it is ready to consume them**

**Event-Driven consumer** reactively processes msg on arrival.

## Principle-Pattern Entwinement

General design principles --
SoC, SRP, High Cohesion, Loose Coupling, Abstraction, Encapsulation, Information Hiding, Interface Segregation principle(ISP), Dependency Inversion, Open-close principle, Design for reuse

Principles of microservice design – Design with ubiquitous language, loose coupling, domain specific requirements/around business capabilities, small team structures, containerised deployments, scaling components at different pace, decentarlized development and deployment

Principles in Event driven and messaging systems – asynchronous communication, decoupling(or loose coupling)

Architecture: Layers, Pipe&Filter, MVC & its flavors, Microservices

A design pattern is a reusable solution to a recurring problem in software design.

Microservices : DDD patterns, Deployment patterns, Service Discovery patterns, Service communication and collaborations patterns (eg API Gateway, Orchestration, choreography)

Event-Driven & Messaging : Event sourcing, CQRS, Event-Broker, Pub-Sub, (EIP)Messaging patterns

**Creational patterns** Provide ways to instantiate single object or groups of related objs

**Structural patterns** Provide manner to define **relationship betwn classes** or objects to form larger structures

**Behavioural patterns** Define manner of **communication between classes** and objects

**Design patterns** provide low lvl soln to problem.