CS3219
AY24/25 SEM 1
github/SelwynAng

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
  1. **On-site:** User manages applications, data, runtime, middleware, OS, virtualisation, servers, storage, networking
  2. **IaaS:** User manages applications, data, runtime, middleware, OS | service provider manages the rest
  3. **PaaS:** User manages applications, data | service provider manages the rest
  4. **SasS:** 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:**
  1. Bare metal: (+): Complete control, physical isolation | (-): Wasted hardware resources, cost, scalability issues
  2. Virtual machine: (+): Improved resource utilization, flexible, scalable | (-): Vulnerable to side-channel attacks, noisy neighbor problem
  3. 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 $\longrightarrow$ 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 $\longrightarrow$ 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 $\longrightarrow$ Analysis $\longrightarrow$ Specification $\longrightarrow$ 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

1. **Business:** Describe why organization is implementing the system
2. **User:** Describe goals or tasks user must be able to perform with the product
3. **System:** Describes connections between your system and outside world
4. **Functional:** Specifies something the system should do
5. **Non-functional/Quality:** Describes something not directly related to system functionality, but how well the system works
6. **Constraints:** States a limitation on design or implementation choices
7. **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)*
  1. Security: Specifying security features at SRS ensures that acceptance tests include testing for security | About privacy, authentication, integrity
  2. Safety: About whether a system can harm someone or something
  3. 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
  4. Availability: Planned uptime of system ($\frac{Uptime}{Uptime+Downtime}$)
  5. 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)*
  1. 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

1. **Control Flow:** Reasoning is on computation order, how the focus of control moves throughout the execution
2. **Data Flow:** Reasoning is on data availability, transformation, latency, how data moves through collection of computations
3. **Call and Return:** Control moves from 1 component to another and back, can be hierarchical/non-hierarchical
4. **Message:** Data sent to a specific address (each component has a unique address other components can send messages to)
5. **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:**
  1. Technical Partitioning: Focus on separation of concerns (Eg. Presentation, Services, Persistence)
  2. Domain Partitioning: Aligned with the domain (Eg. Customer, Shipping, Payment)
- **How is it deployed:**
  1. Monolithic: Deploy all logical components that make up the application as 1 unit, application runs as 1 process
  2. Distributed: Application consists of independent logical components, logical components run as individual processes, communicate over network
- **Types:**
  1. 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
  2. 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)
  3. Event-driven (Technical & Distributed):
  4. Microservices (Domain & Distributed):
  5. 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 $\longrightarrow$ 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 $\longrightarrow$ Data flows in streams (good for image, audio, video, or batch data processing with limited user interaction)
     - We can use files as pipes that capture output of one process and feed into another process
  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 HTTP requests, select model, prepare view), View (renders HTTP response), Model (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 $\longrightarrow$ Provide uniform interoperability between different applications on the internet (HTTP to request access and use data)
- **Constraints:**
  1. Client-Server: REST apps should have client-server architecture for separation of concerns $\longrightarrow$ Improve portability of UI & scalability of server components
  2. Stateless: No client state (session) maintained on server $\longrightarrow$ Server is bound by no. of concurrent requests & not the no. of clients interacting $\longrightarrow$ Improve scalability, reliability, monitoring
  3. Cache-able: Response from server should include if data is cache-able or not $\longrightarrow$ Client returns data from its cache in response to subsequent requests $\longrightarrow$ Improves network efficiency, but client can potentially receive stale data
  4. Layered System: App must be organized as a layered system $\longrightarrow$ Improved overall system complexity by restricting complexity to individual layers, intermediary servers may improve system availability & performance, provide data transformation & filtering
  5. Uniform Interface: Uniform way of interacting with a given server irrespective of device/application, exploits HTTP/HTTPS requests & responses
     - *Resource Identifier:* Global, unique method to identify resources
     - *Resource Representation:* Components perform operations on resource representations
  6. 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

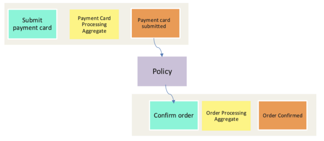
1. A single application as a suite of small services
2. Each microservice offers a well-defined business capability (Features in microservice should be highly related)
3. Each microservice is defined to be developed & deployed independently (Services do not need to share any of their code or implementation with other services $\longrightarrow$ minimal dependency on each other, services are owned by different teams)
4. 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 $\longrightarrow$ Belongs to solution space $\longrightarrow$ Not the same as Microservice (BC is a concept, while MS is an implementation $\longrightarrow$ Ideally, a MS should align with a BC & it is possible for a BC to map to multiple MSes & a MS to implement multiple BCs)
- **Types of Collaborations between different Bounded Contexts (BC):**
  1. Shared Kernel: 2 contexts are developed independently, but they end up overlapping some subset of each other's domain
  2. 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 $\longrightarrow$ 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 $\longrightarrow$ 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



5 Data Patterns in Microservice Architecture

5.1 Database-server-per-service Pattern

- Each service has its own database server
- (+): Loose coupling, allow scaling services at database level, easier to replace underlying database technology to something appropriate with each service
- (-): Hundreds of microservices $\longrightarrow$ Different kinds of database $\longrightarrow$ Explosion of no. of DB clusters $\longrightarrow$ Expensive, unmanageable
- However, Data independence $\neq$ Each microservice to own cluster of DB $\longrightarrow$ Microservices do not modify the same data by having:
  1. Private-tables-per-service: Each service owns a set of tables that must only be accessed by that service
  2. Schema-per-service: Each service has a database schema that is private to that service

5.2 Data Delegate Pattern

- Multiple services require data from a single table $\longrightarrow$ Not loosely coupled
- Can be solved via Data Delegate Pattern (hide shared data behind a delegate service) $\longrightarrow$ Authoritative service for all things related to the other services which require common data $\longrightarrow$ Stop accessing database directly from other services

5.3 Data Lake Pattern

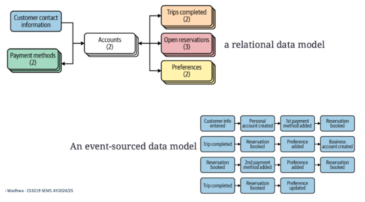
- Data lake is a read-only, query-able data sink (shared space containing copy of data from all concerned microservices) $\longrightarrow$ Owner microservices just stream relevant data into data lake

5.4 Sagas Pattern

- Every step of a transaction defines a compensating action $\longrightarrow$ Compensating action executes if we need to roll back the transaction due to later failure $\longrightarrow$ Compensating action is registered on a routing slip & passed along to next step $\longrightarrow$ If 1 of the later steps fails, it kicks off execution of all compensating actions on routing slip, "undo" modifications & bring system to reasonably compensated state
- Sagas $\neq$ ACID transactions $\longrightarrow$ Sagas does not promise that when a distributed transaction is rolled back, system will necessarily get back to initial state (Eg. Initial state has 0 notifications, but if we cancel reservation, final state has 2 notifications)
- Sequence of events does matter & should be constructed carefully $\longrightarrow$ Better to move steps that are harder to compensate for towards end of transaction (Eg. Move notification to end of process, would save from having to send a lot of correction messages)

5.5 Event Sourcing

- **Definition:** Is about storing facts and any time you have state change $\longrightarrow$ For each modification of state, log (store) the event representing the result of the action (Instead of focusing on persisting the state of the app, you should persist the stream of events which got it into its current state)
- **Event:** Source of truth, **System State:** Series of consecutive events
- **Event Sourcing VS Relational Modelling:**
  - Relational Modelling: Store a state of something (Eg. Current price of economy seat on a flight)
  - Event Sourcing: Store facts, incremental changes of the data $\longrightarrow$ Current state of system is a derivative, a value that is calculated from the events



- **Event:**
  - Has 3 parts (unique id, event type, data)
  - Events acts as data that describes some action/fact, Events are notifications of some change in state/data, Events are recordings of what happened
  - Projection function takes current state and a new event to calculate new state $\longrightarrow$ Take rolling snapshots (saving projections) to speed up calculation of new state from the last snapshot
- **Event Store:** Database storage that can reliably store a sequence of data entries (Store new events, assign correct sequence, notify event subscribers)

5.6 Command Query Responsibility Segregation (CQRS)

- The way we query & the way we store data do not have to be the same $\longrightarrow$ Separate write path from read path (Should something go wrong, internal state of database can be recovered from the log & writes and reads can be optimized independently)
- **Commands:** Operations that change application state & return no data (have side effects within application)
- **Queries:** Operations that return data but do not change application state
- Model used for commands will differ from model used for queries (Data can be stored in different tables/databases for command and query models)

6 More Patterns in Microservices Architecture

6.1 Service Instance per Host Pattern

- Run each service instance in isolation on its own host
- **Service Instance per VM:** Package each service as a VM image $\longrightarrow$ Each service instance is a VM
- **Service Instance per Container:** Each service instance runs in its own container $\longrightarrow$ Container image is a file system image consisting of the applications & libraries required to run the service

6.2 Immutable Infrastructure & Infrastructure as Code

- **Immutable Infrastructure:** Cannot be changed after it is created (updating immutable object requires it to be destroyed and replaced with a new one) $\longrightarrow$ Contain behavior and structures that are easier to predict & reproduce because they do not change
- **Infrastructure as Code:** All infrastructure must be represented as a set of machine-readable files/code $\longrightarrow$ Config files to specify cloud resources needed for deployment $\longrightarrow$ Benefits include speed (no humans needed for deployment), consistency (avoid human errors), accountability (traceable via code)

6.3 Service Collaboration & Communication

- **Service Collaboration:**
  1. Orchestration: Rely on central brain to guide and drive process (Eg. Service A sends instructions to other services directly)
  2. Choreography: Inform each part of system of its job, let them work out the details (Eg. Service A publishes an event that other services subscribe to)
- **Service Communication:**
  1. Request/Sync Response: Client makes a request to service & waits for a response
  2. Notification: Client sends a request to service but no reply is expected or sent (one-way request)
  3. Request/Async Response: Client sends a request to service, which replies asynchronously

6.4 Event Driven VS Request-Response Communication

- **Event-Driven:** Async pattern, Real-time data, Loose coupling between components in the system (No service knows of existence of any other service, and no service owns the entire workflow), Event (something happened), Best for cases where components need to react to changes in state/real-time data processing is needed (Eg. IoT systems, real-time analytics)
- **Request-Response:** Sync pattern, Introduces latency (if requester waits for response), Tighter coupling since requester needs to know details of responder, Request (do this), Best for cases where client needs specific info (Eg. web APIs, database queries)
- Real-time data: Data that is published as it is generated, with no known end of data being published $\longrightarrow$ Travelling in streams/queues that are built to handle large volumes of data

6.5 API Gateway

- A server that is the single point of entry into the system
- Encapsulates internal system architecture & provides an API that is tailored to each client
- Have other responsibilities such as authentication, monitoring, load balancing, caching, etc
- **Backends for Frontends:** A dedicated backend service is created for each frontend interface (e.g., web, mobile, or other client applications) $\longrightarrow$ Allows each frontend to interact with its own backend service, which is tailored to its specific needs

6.6 Service Discovery

- **Service Discovery:** API Gateway needs to know location (IP address and port) of each microservice with which it communicates
  1. Client-side Discovery Pattern: Client queries a service registry & determines network locations of available service instances $\longrightarrow$ then uses a load-balancing algorithm to select 1 of the available service instances & makes a request
  2. Server-side Discovery Pattern: Client makes a request to service via load balancer $\longrightarrow$ Load balancer queries service registry & routes each request to an available service instance
  3. Service Registry Pattern: Service instances are registered with service registry (database of services, their instances & their locations) on startup & deregistered on shutdown $\longrightarrow$ Client of the service and/or routers query the service registry to find available instances of a service

7 Event Driven Architecture

7.1 Events

- **Definition of Event:** Way for a software component to let rest of system know that something important has just happened (allow passing of info between components)
- **Initiating Event:** Originates from an end user & kicks off a business process
- **Derived Event:** Internal events generated in response to initiating event
- **Structure of Event:**
  - Events are typically represented in key/value format (Key: For ID, routing and aggregation operations on events | Value: Complete details of event)
  - Unkeyed: Describes a singular statement of fact (Eg. typically used for system-wide notifications or triggers that are consumed by subscribers interested in general behaviors rather than specific entities) | Entity: Unique thing and is keyed on the unique ID of that thing | Keyed: Contains a key but does not represent entity

7.2 Asynchronous Communication

- **Definition:** When software component broadcasts info to other components, it does not wait for response, nor does it care if other components are available or not (Fire-and-forget)
- **Diagram notation:** Dotted lines for async, Solid lines for sync
- **Pros (+):** Better responsiveness (less time to get response for a request), Better availability (other components' availability don't matter)
- **Cons (-):** More complex/fragmented error-handling than sync comms

7.3 Event Driven Architecture (EDA)

- **Definition:** Way of designing & building apps that are based on exchange of events (relies on async comms when sending & receiving events)
- **Components in EDA:** (1): Event Producers (Publishers): Publish data into streams/queues | Event Brokers: Implements an event broker/event bus | Event Consumers (Subscribers): Listen for & consume event data
- **EDA VS Microservices (MS) Architecture:**
  1. Communication & Performance: EDA relies on async comms (fast), MS communicates via sync using REST and occasionally async comms (sync comms creates latency)
  2. Bounded Contexts & Data Ownership: Nice-to-have for EDA, Foundational for MS
  3. Event VS Request: EDA is built on event processing (responding to something that has happened), MS is built on request processing (responding to something that needs to happen)

7.4 Event Driven Microservices

- **Definition:** Microservices communicate by producing & consuming events (Each event stream has 1 and only 1 producing MS $\longrightarrow$ This MS is the owner of each event produced to that stream)
- **Event Data:** Serves as data storage & communication mechanism between services
- **Event Broker:**
  - Receives events, holds/stores events, and provides events for consumption
  - Event brokers use immutable, append-only event log $\longrightarrow$ Preserves state of event ordering, where events are given auto incrementing index ID
  - Consumer service "seeks" to index ID of last message it read (offset), then scans sequentially, and reads messages in order while recording its new index ID in the partition
  - Traits of Event Broker: Immutability (Event can be altered only by publishing new event with updated data), Ordering (data is served to consumers in same order it was published), Indexing (events are assigned indexes), Partitioning (event streams are partitioned into sub-streams), Infinite retention (Events can be retained for infinite time), Replayability (Any consumer can read whatever data it requires)
  - Event Partitions: Producers write events over many partitions, each partition is an append-only event log, load-balanced consumers share the partitions between them
  - Scalability: Reading & writing can be done in parallel with events in partitions $\longrightarrow$ Scaling is straightforward (with Kafka, hitting scalability wall is impossible)

8 Scalability

8.1 Goal of Scalability

- Increase capacity in some app-specific dimension by:
  1. Increase # of requests that a system can process in a given time period
  2. Increase amount of data to process & manage
  3. Increase resources (CPU, memory, servers)
- **2 strategies to scale:**
  1. Replicate software processing resources (eg. sever) to provide more capacity to handle requests & thus increase throughput
  2. Optimize available resources by using more efficient algorithms, adding extra indexes in databases to speed up queries, rewrite server in faster programming language

8.2 Downside of Monolithic Architecture

- Monoliths grow in complexity as app has more features $\longrightarrow$ If request loads stay relatively low, app architecture can suffice
- If request loads grow $\longrightarrow$ Requests take longer to process (Single server will be overloaded, leading to bottleneck)

8.3 Scaling for Service

1. **Scale Up Service:** Upgrade server (More CPU, more memory)
2. **Scale Out Service:** Replicate service (run multiple copies on multiple server nodes $\longrightarrow$ N replicas, R requests result in each server node processing R/N requests), requires Load Balancer & Session Store
   - Load Balancer: All user requests are sent to LB, which chooses a service replica target to process the request | LB relays responses from service back to client
   - Session Store: API implementations in services must retain no state associated with individual client's session for LB to be effective and servers to share requests evenly $\longrightarrow$ Data representing state of the client will be stored in the Session Store

NOTE: Simple Scale Out still has limits (Single database capability will still limit response time)

8.4 Scaling for Database

1. **Caching (query database infrequently):**
   - Caching stores recently retrieved & commonly accessed database results in memory, so they can be retrieved quickly
   - For data that is *frequently read and changes rarely*, processing logic can be modified to first check a distributed cache $\longrightarrow$ If data not in cache, code must query database & load results into cache as well as return it to caller $\longrightarrow$ Decide when to remove/invalidate cached results
2. **Scale Up Database:** Scale up to bigger, more powerful data store (However, database grows to exceed processing capability of a single node, Low latency database accesses are required to service clients spread around globe)
3. **Scale Out Database using Read Replicas:** Configure ≥ 1 nodes as read replicas of main database (Writes are only possible to Primary node, All changes are asynchronously replicated to Secondary nodes, Secondary nodes are used as read replicas & are physically located in different regions to support global clients)
4. **Scale Out Database by Partitioning Data:** Distribute database over multiple independent disk partitions & database engines
   - Horizontal Partitioning (split by row): Splits logical table into multiple physical partitions (Individual rows allocated a partition based on some value in the row or hash function on primary key)
   - Vertical Partitioning (split by column): Partitions table by columns, where a row is split into ≥ 1 parts (Eg. Partition a row between static, read-only data & dynamic data)

8.5 Scaling processing with multiple tiers

1. A service can call ≥ 1 dependent services, which in turn are replicated & load-balanced
2. Provide service replicas for web clients & mobile clients, each of which are scaled independently based on individual load $\longrightarrow$ Each service replica can be load-balanced & use caching, while utilizing a core service that provides database access *(similar to BFF pattern)*
3. **Swim Lanes/Pod Architecture/Fault-Isolation Architecture:** Placing a group of services within a boundary such that any failure within that boundary is contained within the boundary & the failure does not propagate or affect services outside of the said boundary $\longrightarrow$ To meet expectations of region-specific requirements, have faster response time from customer-specific perspective

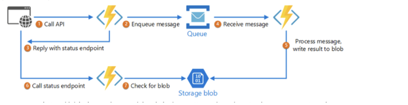
8.6 Scaling in 3D

1. **X-axis:** Horizontal scaling (run multiple identical copies of app behind LB)
2. **Y-axis:** Splits app into multiple, different functions or services
3. **Z-axis:** Distributing data storage

9 Asynchronous Communication

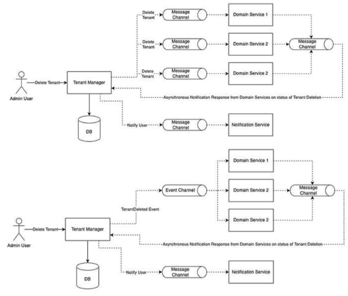
9.1 Request-Reply Pattern

- **Synchronous:** Client sends a request for a resource to a server $\longrightarrow$ Server sends back a response corresponding to the resource or a response with error message
- **Asynchronous:** Decouple backend processing from requester, where backend processing needs to be asynchronous, but requester still needs a clear response
  1. Async API endpoint accepts work from client and puts work in a queue for processing $\longrightarrow$ Sends a HTTP response including location header to a status endpoint
  2. Backend function picks up operation queue, does the work, writes result to a shared storage location
  3. Status endpoint checks whether request was completed $\longrightarrow$ If request is completed, it returns a response, or redirects client to resource URL $\longrightarrow$ If request is still pending, returns 202 accepted with self-referencing location & ETA for completed resource



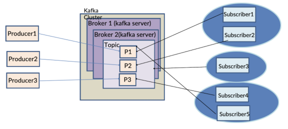
9.2 Event VS Message

- **Similarities:** Both utilize async communication, facilitate decoupling of services, carry some info from 1 to other services to consume
- **Differences:**
  - Event: Carries an info which is a fact, a state change, something which has happened in the past | Events are broadcasted for other services to consume & typically use Pub-Sub pattern | Publisher owns the event payload & underlying channel (Topic) on which events gets published
  - Message: Carries a request (command/query) to be executed by another service | Messages are point-to-point interactions between 2 services | Receiver owns the message payload & underlying channel (Queue) on which messages are published
- **Topic VS Queue:**
  - Topic: Enables one-to-many communication, reaching multiple subscribers simultaneously | Allows for non-sequential processing
  - Queue: Single receiver, ensuring each message reaches one receiver | follows FIFO principle
- **Design Examples**

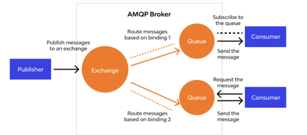


9.3 Protocols & Brokers

- **Pub-Sub Broker (Multiple Receiver/Topic/Event-based)**
  - Eg. Kafka $\longrightarrow$ Each Kafka Broker is responsible for delivering the event to right consumer | Can add as many brokers we need without any downtime (enables horizontal scaling)
  - Unit of data within Kafka is an Event $\longrightarrow$ Events are categorized into Topics $\longrightarrow$ Topics are broken down into partitions $\longrightarrow$ All events produced with given key get written to same partition



- **AMQP (Advanced Message Queuing Protocol) (Can both configured for both topic & queue)**
  - Messages are published to exchanges $\longrightarrow$ Exchanges then distribute message copies to queues $\longrightarrow$ Broker either deliver messages to consumers subscribed to queues OR Consumers pull messages from queues on demand
  - Protocol can be used with any programming language (Can be used by any client that supports AMQP in any language)

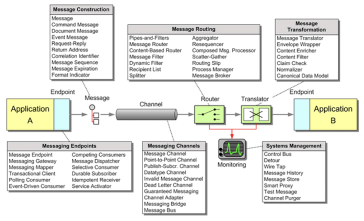


- Exchange Types:
  NOTE: Binding is a link set up to bind a queue to an exchange | Routing key is message attribute the exchange looks at when deciding how to route the message to queues
  1. *Direct:* Message is routed to queues whose binding keys exactly matches the routing key of the message (1-to-1)
  2. *Fanout:* Fanout exchange routes messages to all queues bound to it (1-to-Many)
  3. *Topic:* Topic exchange does a wildcard match between routing key & routing pattern specified in the binding (1-to-Many)

9.4 Persistent VS Transient Communication

- **Persistent Communication:** Messages are stored at each intermediate hop along the way until next node is ready to take delivery of message
- **Transient Communication:** Messages are buffered only for small periods of time $\longrightarrow$ If message cannot be delivered or next host is down, message is discarded

10 Messaging Patterns



10.1 Message Construction

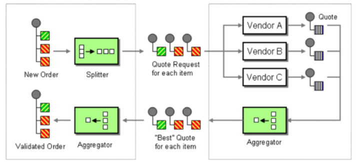
- Message includes a header specifying type of info being transmitted, its origin, destination, size & other structural info
- Message includes a payload containing actual info
- Message includes optional properties that are used for message selection & filtering

10.2 Message Channel

- **One-way Property:** A channel transmits messages in 1 direction, 2-way message needs a 2-way channel (Eg. Request-and-reply each needs its own channel)
- **Return Address:** Request contains a return address to tell replier where to send reply to
- **Correlation ID:** Reply should contain Correlation ID that specifies which request this reply is for (Message ID of Request == Correlation ID of Reply, Correlation ID of initial Request will be null)
- **Special Purpose Channels:** Invalid Message Channel (Handles erroneous messages) | Dead Letter Channel Handles message that could not be delivered | Datatype Channel (For specific type of data, all messages on a given channel contain same data type)
- **Pub-sub Channels:** Used when multiple parties are interested in certain messages (Any message published to a topic is received by all topic subscribers) $\longrightarrow$ Pub-sub request-response patterns are useful when it is important to communicate with multiple services that do parallel work, but their responses need to be aggregated afterwards *(NOTE: Pub-sub channel for request messages, P2P channel for response messages)*

10.3 Message Routing

- **Message Routers:** Consume messages from 1 message channel & reinsert them into different message channels, depending on a set of conditions
- **Simple Routers:** Route messages from 1 inbound channel to ≥ 1 outbound channel
- **Composed Routers:** Combine multiple simple routers to create more complex message flows
- **Context-based Router:** Decides the message's destination based on specific contexts (used for load balancing, test or failover functionality)
- **Content-based Router:** Examines message content & routes message onto a different channel based on message data | has to have knowledge of all possible recipients & their capabilities | Changes in recipient list leads to change in content-based router
  - Message Filter: Special kind of Content-based router that eliminates undesired messages from a channel based on a set of criteria | has only 1 output channel (for accepted message)
  - Comparison against Pub-Sub Channel with Message Filter: Content-Based Router has exactly 1 consumer receiving each message, Central control and maintenance, Needs to know about participants, Often used for business transactions like orders, Generally more efficient with queue-based channels | Pub-Sub Channel with Message Filter can have ≥ 1 consumer to consume a message, Distributed control and maintenance, No knowledge of participants required, Used for event notifications/info messages, Generally more efficient with Pub-Sub Channels
- **Message Splitter:** A single message split into multiple messages $\longrightarrow$ Each message can then be routed using a Content-Based Router
- **Message Aggregator:** Multiple messages aggregated into a single message $\longrightarrow$ Aggregator then publishes message to output channel for processing
- **Message Scatter-Gather:** Routes/broadcasts single message to multiple participants concurrently & reassembles/aggregates replies into single message $\longrightarrow$ Ideal for requesting responses from multiple parties & then aggregating and processing that data



10.4 Message Transformation

- **Message Translator:** Converts messages from one format into another $\longrightarrow$ resolves differences in message formats without changing the apps or having apps know about each other's data format, but if large number of apps communicate with each other, 1 Message Translator may be needed btw. each pair of communicating apps
- **Canonical Data Model:** Provides additional level of indirection btw. app's individual data formats $\longrightarrow$ System A sends data to System B by first translating its data into canonical format, System B then receives data from System A and translates canonical format into its own data format

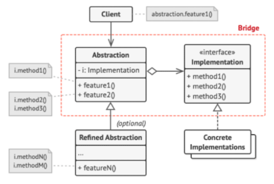
10.5 Message Endpoint

- **Definition:** Message Endpoint is interface btw. an app and a messaging system $\longrightarrow$ Used to send messages or receive them, but 1 instance does not do both $\longrightarrow$ Endpoint is channel-specific (Single app will use multiple endpoints to interface with multiple channels)
- **Main Endpoint Types:** MessageProducer & MessageConsumer
- **Consumer Endpoints:**
  1. Polling Consumer: Proactively reads messages once it is ready to consume them (Wants to control when it consumes each message)
  2. Event-Driven Consumer: Reactively process a message on its arrival (Message delivery is an event that triggers receiver into action)

11 Object Interaction Design Patterns

11.1 Design Pattern

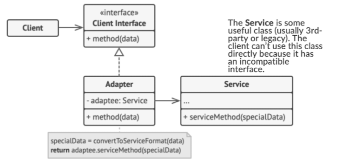
- **Definition:** Design Pattern is a solution *(General design that can be applied to problems in this context)* to a problem *(Goal to be achieved in the context)* in a context *(Recurring situation in which pattern applies)*
- **Bridge Pattern:** Used to decouple an abstraction from its implementation via composition (not inheritance), so they can evolve independently $\longrightarrow$ Useful when extending a class's functionality in multiple ways without creating a proliferation of subclasses
  - Normally consists of (1): Abstraction, (2): Refined Abstraction, (3): Implementation, (4): Concrete Implementation, where Abstraction contains Implementation



- **Proxy Pattern:** Use an intermediary object (Proxy) between a base object and its clients $\longrightarrow$ Clients no longer have direct reference to base object, but instead interact with proxy $\longrightarrow$ Proxy has reference to base object & implements same interface as base object (Proxy contains base object)

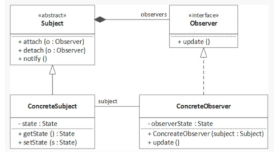
11.2 GoF Patterns

- **Pattern Categories**
  1. Creational: Help designers handle object creation issues (Eg. Factory)
  2. Structural: Provide a structure to relationship btw. objects (Eg. Facade, Bridge, Proxy, Adapter)
  3. Behavioral: Define how objects interact with each other to deliver a task (Eg. Observer, Mediator)
- **Adapter Pattern:** Useful to reconcile differences btw. 2 incompatible interfaces, allowing them to work together seamlessly (Adapter is a class that implements the client interface, while wrapping the service object) (+): Single Responsibility Principle (Separate interface/data conversion code from primary business logic) | Open/Closed Principle (Can introduce new types of adapters without breaking existing client code)
  (-): Increased code complexity due to new interfaces & classes

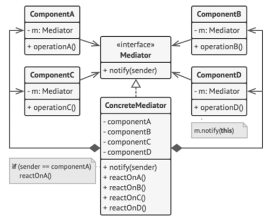


Client
«interface»
Client Interface
+ method(data)
The Service is some useful class (usually 3rd-party or legacy). The client can't use this class directly because it has an incompatible interface.
Adapter
- adaptee: Service
+ method(data)
Service
...
+ serviceMethod(specialData)
specialData = convertToServiceFormat(data)
return adaptee.serviceMethod(specialData)

- **Facade Pattern:** Provide a unified interface to a set of interfaces in a subsystem $\longrightarrow$ Clients interact with the facade instead of individual subsystem (Eg. API Gateway)
  (+): Isolate Code from complexity of subsystem | (-): Facade can become god object coupled to all classes
  NOTE: Difference btw. Facade & Adapter is that Facade focuses more on simplifying & unifying access, while Adapter focuses more on improving compatibility (Eg. RCS Communicator component is more of an Adapter)
- **Observer Pattern:**
  - Defines a one-to-many dependency btw. objects so that when Observable changes state, all Observers are notified and updated
  - Observers register themselves to Observable because they want to be notified when there is a change
  - (+): Open/Closed Principle | Can establish relations btw. objects at runtime | (-): Observers are notified in random order
  - Push Model: Observable changes state $\longrightarrow$ Notifies all observers $\longrightarrow$ Observer calls back to Observable to retrieve more detailed info
  - Pull Model: Observable changes state $\longrightarrow$ Subject pushes a snapshot of its state to all observers



- **Mediator Pattern:** Define an object that encapsulates how a set of objects interact $\longrightarrow$ Promotes loose coupling by keeping objects from referring to each other explicitly & letting us vary their interaction independently $\longrightarrow$ If an object is updated with new interaction rules or a new object is added, only mediator object needs to be updated
  (+): Single Responsibility Principle (extract communication into single place) | Open/Closed Principle | Reduce coupling & reuse components | (-): Mediator can become god object



- **Data Transfer Object (DTO) Pattern:** A group of values in an ad-hoc structure just for purpose of passing data around (No business logic, only contain storage, accessors, serialization/parsing methods) $\longrightarrow$ Reduces roundtrips to server by batching up multiple parameters in a single call