**L1/2 - Software Development Process**

The application of a systematic, disciplined, quantifiable approach to the **development**, **operation**, and **maintenance** of software.

- Hardware Advancement and data availability and proliferation of devices forced the need for better software engineering.
- Increasing demand from digital natives encouraged the growth of SWE

**Types of Software Applications**

Various ways to categorize software applications

- **Based on Computation & Response:**
  - **Real-time systems** – RTOS; **Concurrent** - Multi-threading and responsive programs; **Distributed** - Distributed applications for scalable computing
- **Nature of code & data**
  - Open-source software (e.g. GNU/Linux, Apache, etc.)
  - Open-content systems (e.g. Wikipedia)
- **Deployment mode:** Embedded, Desktop, Edge Systems, Cloud-Native Systems

**Edge Computing**

- Complements cloud intelligence for balanced centralized computing and localized decision-making
  - Cloud-based applications have network latency issues that require computation to be located at the edge.
  - Some computations are not suitable to run on the edge.
    - E.g. some security systems may require edge computation
- **Challenges**: Limited computation power, battery life, and heat dissipation

**Cloud Computing Applications**

- Hosted on external data centers, delivered over the internet
- Models include IAAS, PAAS, SAAS (Examples: AWS, Heroku, Google Sheet)
- Cloud-enabled - legacy applications modified for cloud operation

**Cloud-Native Applications**

The software approach of building, deploying, and managing modern applications in cloud computing environments

- **Characteristics:** Immutable infra, $\mu$services-based, API-driven, service mesh
- Utilizes containers, dynamically managed
- **Cloud native development:** Involves CI/CD, DevOps, Serverless
- **Application Stack:** Infrastructure layer, Provisioning layer, Runtime layer, Orchestration and management layer, application definition and development layer, observability and analysis tools

**Software Deployment**

Activities that make the software available for use after development

- Process between acquisition of software and execution of software
- Deployment decisions can affect the quality attributes (**see §requirements**)

**Deployment Challenges:**

- Driven by hardware advances, data demand, device proliferation
- Re-thinking of deployment strategies necessary

**Some Deployment Considerations:**

- Integration of the Internet, shift towards cloud-nativity, **portability**
- Managing large-scale content delivery (**availability and performance**),
- **Interoperability** on heterogeneous platforms (e.g. deployable on different OSes, ARM vs x86, etc.), etc.

**Dependency and Change Management (maintainability aspect):**

- Management of interdependencies, configurations, version changes, and how dependencies communicate with each other
  - (e.g. changes in Docker compose versions, façade pattern etc.)
- Coordination among components, ensuring **performance** and **security**

**Deployment Mechanisms**

- **Bare metal** (Apps on top of OS)
- **Virtual Machines** (Apps in OSes on top of hypervisor)
- **Containers** (apps in isolated environments on top of container engine)
  - Enables easy integration of the internet and related advances
  - Bundles runtime with code, reproducible on any OS
  - Supports dependency and change and environment management
- **Considerations**:
  - specific target platforms/environments **vs** write once run anywhere
  - maximizing hardware resources and reducing costs
  - Enabling scalability and portability

**Containers vs Orchestrator**

- Containers provide the platform for building and distributing services
- Orchestrator, a separate software that integrate and coordinate services, scale up and down based on demand, provide communication between containers

**Serverless** – execution of code managed entirely by cloud providers

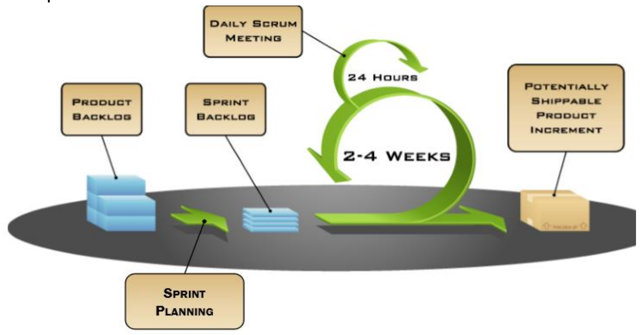
**Software Development Process/Models (SDLC)**

- Involves people, product, technology, phases, activities, milestones
- Aims to reduce project failure risks
- A framework for project planning and execution
- Divides software development work into phases to improve
  - Design, Product management, Project management
- **Typical SDLC Process:** Req. Analysis → Design → Implementation → Testing → Evolution → Req. Analysis
- **Factors affecting Software Development**
  - Requirements, Process (Resources, Time), Criticality and Consequences, People (Competence) and Technology
- **Many Software Process Models**: Waterfall, Spiral, Rapid Prototyping, eXtreme Programming, Rational Unified Process, Test driven development, Agile (Scrum, Crystal, etc.)

**Model Selection Criteria:**

- **Waterfall** for well-understood, fixed requirements
- **Iterative & incremental** for fuzzy and evolving requirements

**Scrum** - An agile framework for managing complex projects, typically software development.



- Iterative and incremental approach, dividing the project into small, manageable units called **Sprints, usually lasting 2-4 weeks.**
- **Key artifacts**: Product Backlog, Sprint Backlog, and product Increments.
- **Ceremonies**: Sprint Planning, Daily Stand-Ups, Reviews, Retrospectives.
- **Key roles**: Scrum Master, who facilitates the process, the Product Owner, who represents the stakeholders' interests and maintains the Product Backlog, and the Development Team, which handles the actual work.

These elements work together to promote transparency, inspection, and adaptation, with a strong focus on continuous improvement and delivering tangible, shippable products at the end of each Sprint.

**CI/CD**

**Continuous Integration:** Build → Unit Tests → Staging → Acceptance Tests...

- Development practice that requires developers to integrate code into a shared repository several times a day. Each check-in is verified by an automated build, allowing early problem detection

**Continuous Delivery:** ... → manual deployment to production

- About ensuring readiness of good builds for deployment

**Continuous Deployment: ...** → automated deployment to production

- About automating the release of a good build to production

**DevOps**

- Blends software development and operations staff and tools
- Reduce time between committing a change to a system and the change being pushed to production
- Encompasses CI/CD, monitoring and logging, communication, and collaboration tools, and building infrastructure as code.

**Recent Trends in Software Development Process** (seen in all aspects of SDLC)

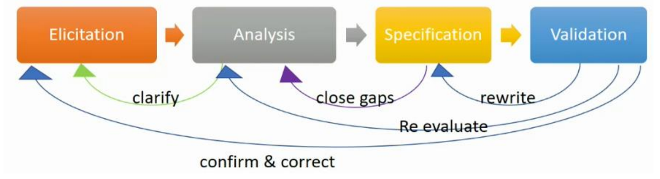
- **Workflow Automation**
- **Pattern Detection** – e.g vulnerability detections, detect committed keys
- **Collaboration** – GitHub Copilot, ChatGPT, documentation generation, AI explaining codebase

**L2/3 – Requirements**

- A condition/capability needed by a user to solve/achieve a problem/objective
- A condition or capability that must be met or possessed by a system
- A documented representation of a condition or capability

**Software Requirements Process**

- SRS is a non-linear process, can jump from any stage to any stage



**Elicitation**: Discovering requirements through methods such as interviews, workshops, document analysis, and prototyping.

**Analysis**: Analyzing, decomposing, deriving, understanding, and negotiating requirements, as well as identifying gaps.

**Specification**: Writing and illustrating reqs for comprehension, review, and use.

**Validation**: Confirming the correct set of requirements that will enable developers to build a solution.

**Sources of Requirements**

**Documents**: Descriptions of current or competing products, standards, regulations, and help desk problem reports.

**Event-Response Tables**: Identify external stimuli and describe system responses.

**Interviews**: Involving focus groups, use case workshops, and product champions as key customer representatives.

**Prototyping**: Useful for requirements design and implementation.

**Questionnaires and Marketing Surveys**: Often require a pilot run due to the difficulty in asking clear questions.

**Usage-Centric Requirements**: Observation of users performing their jobs, creating workflow diagrams, conducting "day in the life" studies, and analyzing the information users have when performing tasks.

**Outcome of Requirements Development Process**

**Software Requirements Specification (SRS)**:
- A set of precisely stated services and constraints that software must satisfy, providing a complete description of the software's behavior; Direct and indirect requirements of the system; Only tells what work is to be done

**Rights, Responsibilities and Agreements:**
- All major stakeholders are confident of development within a balanced schedule, cost, functionality, and quality.

**Requirements under Change control**
- Need to have an official body to manage changes to requirements

***Product Backlog (in contrast to SRS, not outcome of req. dev. process)***
- **Repository** of the work to be done; **Facilitates** prioritization of work and planning; **In some cases,** we can use SRS like a product backlog

**Types of Requirements**

**Business Reqs (A)**: High-level needs of the organization, explains why the organization is implementing the system and recorded in **vision and scope document**

**User Reqs (B)**: Goals or tasks the user must be able to perform with the product.
- Captured as **use cases, user stories, or scenarios**

**System Reqs (C):** Detailed specifications describing the functions of the software.

**Quality Reqs (C):** Standards that describe the desired quality attributes of the sw.

**Functional Requirements (C):**
- Specific behaviors or functions the software must exhibit.
- Defined by biz rules, existing sys reqs, and how it should interface externally

**Non-Functional Requirements (C):** Requirements that describe how well the software performs a function, rather than what the function is.
- Not directly related to the functionality of the system
- May describe how well the system works or quality attributes

**Constraints (C)**: Limitations/restrictions on system design/implementation

**Data Reqs (C)**: Specifications related to data handling, storage, and retrieval

**All requirements labeled (C) form the SRS document**

**Quality Attributes -** Non-exhaustive taxonomy of Quality Attributes:

| | | | | |
|---|---|---|---|---|
| **Availability** | **Performance** | Efficiency | Scalability | **Robustness** |
| **Safety** | **Security** | **Reliability** | **Integrity** | Verifiability |
| **Deployability** | **Compatibility** | **Installability** | Portability | Maintainability |
| **Usability** | Testability | Modifiability | Reusability | **Interoperability** |

- **External facing quality attributes (bolded)**
  - Observed when software is executing and impacts user's experience of using the software. Forms user's perception of software quality
- **Internal Quality Attributes (non-bolded above)**
  - Not directly observed when software is executing, but perceived by maintainers and developers
  - Encompasses aspects of design that **may impact external attributes**
- Different applications have different quality attributes. For example,
  - Embedded software may prioritize **robustness** and **reliability** while web applications may prioritize **interoperability**.

**Quality Attribute – Security**
- Traditionally considered in the design phase, one stage late in SDLC
- We should specify security features in the SRS to ensure they are included in acceptance tests, thereby improving the software's security assurance.
- **Safety vs Security**: (preventing harm to someone or something) vs (concerning privacy, authentication, and integrity).

**Quality Attribute – Performance**
- Encompassing system responsiveness and impacting user experience.
- **Various performance metrics**: response time, throughput, data capacity, dynamic capacity, real-time guarantees, latency, behavior in degraded modes etc
- Performance requirements affect quality attributes, architecture, design, deployment, hardware choice, and network type:
  - An overloaded real-time system unable to respond to critical alerts
  - Stringent query response time may demand replicated databases
  - Latency requirements may require deploying caches
  - Monolithic vs Microservice architecture

**Quality Attribute – Availability**
- Measures the planned uptime of the system $\frac{uptime}{uptime+downtime}$
- May impact deployment cost and software design complexity.
  - E.g. need to increase redundancy, graceful failover etc.

**Quality Attribute – Scalability**

The application's ability to accommodate growth, with both hardware and software consequences (such as rewriting to handle parallel & distributed computing). It includes horizontal scaling (adding machines) and vertical scaling (upgrading existing machines).

**Quality Attribute Trade-**offs (i.e. cannot optimize all quality attributes)
- Quality attributes may complement or counter another quality attribute

| | Availability | Efficiency | Installability | Integrity | Interoperability | Modifiability | Performance | Portability | Reliability | Reusability | Robustness | Safety | Scalability | Security | Usability | Verifiability |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Availability** | ▨ | | | | | | | | + | | + | | | | | |
| **Efficiency** | + | ▨ | | | – | – | + | – | | | – | | + | | – | |
| **Installability** | + | | ▨ | | | | | | + | | | | | + | | |
| **Integrity** | | | – | ▨ | – | | – | | | – | | + | | + | – | – |
| **Interoperability** | + | | – | – | ▨ | | – | + | + | | + | – | | – | | |
| **Modifiability** | + | | – | | | ▨ | – | | + | + | | | + | | | + |
| **Performance** | | + | | | – | – | ▨ | – | | | – | | – | | – | |
| **Portability** | | – | | | + | – | – | ▨ | | + | | | | – | – | + |
| **Reliability** | + | – | | + | | + | – | | ▨ | | + | + | | + | + | + |
| **Reusability** | | – | | – | + | + | – | + | | ▨ | | | | – | | + |
| **Robustness** | + | – | + | + | + | | – | | + | | ▨ | + | + | + | + | |
| **Safety** | | – | | + | + | | – | | | | + | ▨ | | + | – | – |
| **Scalability** | + | + | | + | | | + | + | + | | + | | ▨ | | | |
| **Security** | + | | | + | + | | – | – | + | | + | + | | ▨ | – | – |
| **Usability** | | – | + | | | | – | – | + | | + | + | | | ▨ | – |
| **Verifiability** | + | | + | + | | + | | | + | + | + | + | | + | + | ▨ |

**Requirement Validation & Verification**

Assessing if the requirements are correct and align with business objectives.
- **Validation**: Ensures the right reqs are written and trace back to biz objectives.
- **Verification**: Ensures reqs are written correctly, checking for properties like completeness, correctness, feasibility, prioritization, and unambiguity.

Methods for validation & verification include **informal approaches** (peer desk-check, pass-around, walkthrough) and **formal methods** (inspection with a formal process and checklist)

**L4 – Architecture / High Level Design**

Software Architecture design represents the structure of data and program components that are required to build a software.

**Building Blocks of Architecture**
- **Component**: An element that models an application-specific function, responsibility, requirement, task, or process.
- **Configuration**: Refers to the topology or structure.
- **Connector**: An element that models interactions among components for transferring control and/or data

**Reference Architectures & Architectural Patterns:** These provide a common architectural framework used by numerous applications. They explain high-level structures of similar applications and lead to architectural patterns. These patterns form the basis for architectural design and address application-specific problems within a specific context

**Architectural Design Decisions**

Many design decisions to consider, non-exhaustive. Some are listed below.
- **Type of Deployment**: Decisions about whether the architecture should support single-core/multi-core, single-processor/multi-processor, and standalone/distributed systems.
- **Decomposition of Structural Components**: How to break down system's structural components into sub-components, influencing maintainability, scalability, and flexibility

**Component Interaction in Architecture Diagrams**

The interaction of components is represented by these concepts
- **Control Flow –** depicting order and seq. of execution between components
- **Flow of Data** – depicting how data flows between components.
- **Call & Return** – mvment of control from one component to another and back
- **Message and Event –** communication via event notification or message passing, can be synchronous or asynchronous and is point-to-point.
  - **Message** is data sent to a specific address
  - **Event** is some data emitted from a component for any listeners to consume

**Decomposition, Modularity**
- **Decomposition, Componentizing & Packaging:** This involves **horizontal slicing** (designing by layers – e.g. model controller routes separation in QnService) and **vertical** slicing (designing by features)
- **Principle of Modularity**: Aims for shorter development times and enhanced flexibility and comprehensibility by breaking down large components into smaller units with **clear interfaces** (APIs), simplifying complexity management

**Types of Cohesion**

**Functional**: Facilities perform only one computation with no side effects.

**Layer**: Related svcs are grouped together in a strict hierarchy, where higher-level svcs can access only lower level svcs. Accessing a svc may result in side effects.

**Communicational**: Facilities operating on the same data are kept together.

**Sequential**: Procedures working in sequence to perform computation are grouped together, with the output from one becoming the input to the next.

**Procedural**: A set of procedures called one after another are kept together.

**Temporal**: Grouping procedures used in the same phase of execution, such as initialization or termination

**Utility**: Related utilities are grouped together when they cannot be classified under a stronger form of cohesion

**Types of Coupling**

**Content**: A component modifies the internal data of another component

**Common/Global**: Global variables; all modules using the global var. are coupled

**Control**: One module controls another by passing information on what the second module should do, e.g., using a "flag".

**External**: Dependency on elements outside the system's scope, like the OS, shared libraries, or hardware.

**Data**: Sharing data between modules, e.g., via passing parameters.

**Temporal**: Two actions are bundled together bcuz they happen at the same time.

**Inclusion/Import**: Involving including a file or importing a package

**Classical Architectural Styles**

**Pipe and Filter Architecture**: Data enters the system, flows through components (filters), undergoing transformations until it reaches the final destination (data sink). Filters work independently and do not share state with other filters

**Layered Architecture**: Organizes software into layers of components, supporting independent development and evolution of different system parts. Each layer has distinct and specific responsibilities. The layer below only provides services to layers on top through layer interfaces.

- **Pros of single layer** – performance and reduced context switch overheads
- **Pros of multi-layer** – modularity, portability, and enhanced scalability

**Model View Controller Architecture**

- Aims to support the user's mental model of information and enable inspection and editing of this information.
- It separates code into View (UI elements), Controller (coordinates between Model and View), and Model (business logic)
- **Benefits of MVC**: Offers SoC, resulting in modularity, extensibility, reduced complexity, better testability, and facilitates framework usage

**Web MVC + Single Page Applications**

**Web MVC**: Involves two entities - Server (holding the model) and Client (interacting with the server). The Controller handles user HTTP requests, selects the model, and prepares the view. The View renders the HTTP response, and the Model includes business logic and persistence.

- **Controller's Additional Responsibilities**: Mapping requests to handlers, managing the requests, selecting models, and preparing views.
- Can be split into **Page controller** (for handling requests to specific pages)
- or **Front controller** (for handling HTTP requests).

**SPA**: JS programs that run in the browser, sending queries & retrieving data w/o refreshing the webpage. They provide fluid experience, save bandwidth, and reduce perceived latency **but** are harder to develop, require JS & vuln to XSS

- **Frameworks in SPA**: Introduce extra build steps to create static bundles of HTML, JS, and CSS. SPA apps include logic for making HTTP API requests against resources served by API Controllers, usually responding with JSON

**L5 – Representational State Transfer (REST)**

**REST** defines constraints for transferring, accessing, and manipulating textual data representations of hypermedia in a stateless manner across a network of systems. It sets rules for creating web services but is not an architecture itself.

**Aim**: to provide uniform interoperability between different internet applications, exploit native HTTP capabilities like GET, PUT, POST, and DELETE

**Hypertext, Multimedia, Hypermedia**:

- **Hypertext**: Text with hyperlinks to other texts accessible immediately.
- **Multimedia**: Integration of multiple media forms (text, audio, video, images, graphics) with computers.
- **Hypermedia**: A nonlinear medium of information that includes graphics, audio, video, text, and hyperlinks

**REST Architecture Constraints**

**Client-server**: REST applications should adopt a client-server architecture for separation of concerns, enhancing UI portability and server scalability and enable independent evolution for internet-scale requirements.

**Stateless**: Each client-server interaction is self-contained with full info in the query, headers, or URI, improving scalability, reliability and monitorability. Each server is bounded by concurrent requests and not the number of interacting clients.

**Cacheable**: Server responses should indicate whether data is cacheable, enhancing network efficiency and user-perceived performance.

**Layered System**: REST apps should be organized as closed-layered systems, where each layer only interacts with its immediate neighbors, improving system complexity and potential performance (e.g., through load-balancing and caching).

**Uniform Interface**: Ensures a consistent method of interaction with servers, regardless of device or app type, using standard HTTP/S requests and responses

- **Generality of the component interface/Program to an Interface**
  - Implementations are decoupled from the services they provide
- Anything can be a resource (e.g., documents, images, users), and resources are identified using **stable, global, unique resource identifiers**.
- Operations on resources are performed through their representations, consisting of data and metadata.
  - Client has representation of resource and contains enough information to modify the resource on the server
- Messages should be self-describing and include sufficient information for processing, facilitating resource discovery and access through hypermedia links

**Code-On-Demand:** This optional constraint allows client functionality to be extended by downloading executable code (e.g., Applets, JavaScript), simplifying the client and enhancing extensibility

**L6 – Microservices**

**Microservices**: A **set of software applications** with a **l**imited scope that work together to form a larger solution. They have **minimal** but **well-defined capabilities**, for the sake of creating a **modularized** architecture.

- **Independent Capability**: Microservices separate functionality into a collection of independent services, **simplifying deployment, testing, and maintenance**.
- **Independence**: Services don't share code or implementation. Communication occurs via **well-defined APIs**, and each service can be developed, deployed, operated, and scaled independently **without affecting other services.**

**Microservices Characteristics**

**Organized Around Biz Capabilities**: Services are structured around specific biz functions, such as various departments in a retail company (e.g., warehouse, finance).

**Loosely Coupled**: Services interact through a well-defined boundary, such as via a communication protocol, promoting a reduced coupling in implementation, making it more maintainable and testable

- Also makes it possible to reduces domain coupling between microservices

**Owned by Small Teams**: Reflects Conway's Law, suggesting that the architecture of a system mirrors the organization's communication structure. Small, cross-functional teams are typically responsible for specific microservices.

**Independently Deployable**: Each microservice has distinct deployment, **scaling**, and monitoring requirements and can run multiple instances on a host

- **Deployment Pattern**: Each service instance runs in isolation on its own host, either as a VM or container.

**Identifying Microservices**

- **Size Consideration**: Microservices should be neither too big nor too small, with a well-designed domain model aiding in their identification.
- **Domain Model**: Represents a view of the problem domain and should be isolated from technical complexities. Rather, technical complexities are built around the domain model. Helps in reasoning about microservices
  - **Domain**: The problem space that a biz occupies and provides sol<u>n</u>s to. Includes rules, processes, ideas, biz-specific terminology, and anything related to the problem space. It exists regardless of the existence of the biz.

**Domain-Driven Design (DDD) Concepts**

**Ubiquitous Language**: A shared language between domain experts and developers, ensuring clear communication and understanding.

- It is the overlap between business jargon and technical jargon

**Sub-domains (SD)**: Component of the main domain. Belongs to the problem space. Focuses on a specific set of responsibilities and reflects some of the biz's organizational structure.

- **Core SD** (key differentiator for the biz and most valuable part of application)
- **Supporting** (related but not differentiators, can be inhouse or outsourced),
- **Generic** (not specific to business, often outsourced or off-the-shelf software).

**Bounded Contexts**: Highly **cohesive**, explicit boundaries relevant to a subdomain. Each bounded context is typically owned by a single team.

- E.g., UserService and QnService are two contexts with clear boundaries
- Comes with its own ubiquitous language

**Sub-domains vs bounded contexts**

- **Subdomains** – logical separations of the domain; problem space
- **Bounded Contexts –** technical solutions; solution space
- Both Subdomains and Bounded Contexts can overlap with each other
  - For example, UserService provides technical solutions for the subdomains of user management and authentication
  - The subdomain of user profile management was handled by UserService and QnHistoryService

**Aggregates**: Clusters of related objects treated as a single unit for data changes, with a transactional and consistency boundary

- **Transactional**: Changes to the aggregate will either all succeed, or none will.
- **Consistency**: External processes or objects can only read the aggregate's state, which can only be modified through the aggregate's public interface
- **Aggregate Root:** An aggregate is structured hierarchically with entities, and the parent entity is known as the Aggregate Root. This root entity serves as the aggregate's public interface through which all interactions with the aggregate's internal entities occur
- **Aggregates vs Bounded Contexts:** Both represent collections of things, but fundamentally differ. Aggregates define business entity models, while bounded contexts define business sub-domains

**Microservices and Database, Communication, and Discovery**

**Database per Service Pattern**: Each service has its own database schema, often resulting in data duplication and can use a database best suited to its needs

**Service Communication**: Can be synchronous or asynchronous, such as **request-sync response**, **notification** (aka 1-way request), **req-async response**

**API Gateway**: A server that is the single-entry point into the system, encapsulating the internal architecture and providing tailored APIs. Might have other responsibilities such as authentication, monitoring, load balancing, caching etc.

**Orchestration vs Choreography:** Orchestration relies on a central brain to guide and drive processes, while choreography informs each part of the system work and let it work out the details.

**Service Discovery:** Enables services in a microservices architecture to locate and communicate with each other.

- **Client-side Discovery**: Registry-aware clients locate service instances and makes a request to an instance based on a load-balancing algorithm
- **Server-side Discovery**: A central server (like an API Gateway) directs client requests to appropriate service instances.
- **Service Registry**: A database where services register their instances and locations; used for querying available services.
- **API Gateway Role**: Acts as a single-entry point, utilizing Service Discovery to route requests to the correct microservices. Handles additional tasks like authentication and load balancing

**L7 – Event Driven Architecture (EDA)**

A software architectural design pattern where decoupled applications can asynchronously communicate by issuing and consuming events via a broker.

- The broker facilitates **loose coupling** of applications, so applications don't need to know the origin or destination of the data.
- Key components: event producers, brokers (aka event bus) and consumers

**Events**: Occurrences within a business communication system, serving as both **data** and a **means for asynchronous communication** between services.

**Event Structure**: Typically in key/value format, where the key is for identification and routing, and the value provides complete event details.

**Types of Events**: **Unkeyed** (simple facts), **Entity** (keyed on the uniqueID of an entity; describes the state of that entity at a given point in time), **Keyed** (event w key but unrelated to any entity, for partitioning and aggregation purposes)
**Producers** (aka publishers): entities that generate or emit events
**Consumers** (aka subscribers): entities that listen for and react to events
**Decoupling of Producers and Consumers**: Emphasizes independence of producers and consumers in an EDA, allowing for scalable and flexible system design.
**Advantages of EDAs** (over traditional request-response architectures):
- Decoupling of systems for independent scaling and updates.
- Handle high data volumes with low latency.
- Support real-time processing and analytics (streaming processors)
- Enhanced scalability and resilience.

**Real-time data**: Published as it's generated in streams or queues.
**Event-Driven and Microservices**
Communication: Microservices produce and consume events.
Benefits: Granularity, scalability, technological flexibility, business requirement flexibility, **loosely coupling**, continuous delivery support, high testability.
**Event Sourcing -** A method of persisting application state as an ordered sequence of events, with the event log being the primary source of truth
**Immutable Event Log**: Events are stored immutably in the order they were created, providing a comprehensive audit of system activities.
**State Derivation/Recreation**: The current state of the system can be derived by replaying these events in sequence from a certain initial state or snapshot.
**Data Storage Pattern**: An append-only log is used to store events.
**Data Storage**: Event sourcing can utilize simple in-memory structures or distributed data stores like Kafka and Cassandra
**Command-Query-Responsibility Segregation (CQRS)**
**SoC**: CQRS divides the write path (commands) from the read path (queries). This separation allows for independent recovery and optimization of writes & reads.
**SRP**: Each component either only returns data (query) or alters data (command)
**Interface Segregation**: Clearly defined interfaces for commands and queries, ensuring straightforward and predictable interactions for clients
**Write Path**: Writes are directed into systems like Kafka rather than updating database tables directly.
**Event Stream Transformation**: Event stream can be transformed to suit queries, often using tools like KSQL, and materialized as precomputed queries or views.
**Scalability**: Underlying events are decoupled from views, allowing command and query to be scaled independently.
**L8 – Communication Patterns**
Components communicate either **synchronously or asynchronously**, have **single or multiple receivers** and can be either **persistent or transient**. (3-axes)
**Synchronous vs. Asynchronous Communication**
- Synchronous involves the sender blocking & waiting for a response (HTTP/S)
- Async does not block for receipt/response; can send multiple req (e.g., AMQP)

**Remote Procedure Call (RPC):** Sync Req-Reply/Call-Return pattern
- RPC is synchronous, mimicking a serial thread of execution.
- Allows a program to execute a procedure in another address space, as in distributed applications. (e.g. JAVA RMI, JSON-RPC, XML-RPC, gRPC)

Involves marshalling parameters into a message by the client stub and sending it to the server. The server stub unpacks (unmarshalling) and processes the request, replying in reverse order.
**HTTP/HTTPS Communication:** Sync Req-Reply pattern
A client (usually a web browser) requests a resource, and the server responds with the resource or an error message.

**Asynchronous Request-Reply Pattern:** e.g, polling the status from Judge0 server
- Decouples time spent on backend processing from the requester.
- Uses HTTP 202 and 302 for managing the req and delivery of the response.
- API endpoint accepts work and puts it in a queue for processing.
- Status endpoint manages the completion status and communicates with client

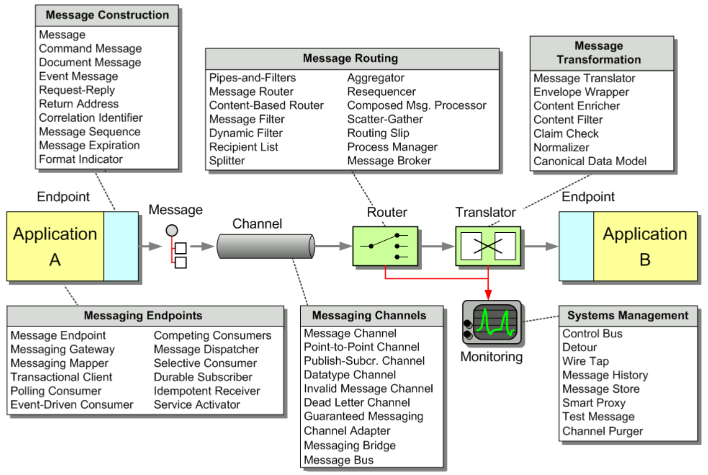
**Asynchronous Messaging – Single Receiver**
Fundamental in loosely coupled systems for scalability & independent operation
Involves a sender, message channel (queue), and receiver. A queue acts as a buffer for messages and ensures each message is consumed by **one receiver**
**AMQP – Advanced Message Queuing Protocol:**
Open protocol enabling client apps to communicate with messaging brokers.
Messages published to exchanges are then copied and distributed to queues.
**Exchange Types in RabbitMQ:**
- **Direct**: Routes messages based on a matching routing key.
- **Fanout**: Routes messages to all bound queues.
- **Topic**: Uses wildcard matching between routing keys and binding patterns.

**Asynchronous Message Passing with Multiple Receivers (Pub/Sub Pattern):**
Messages can be pub-ed to a topic, akin to broadcasting. Subscribers subs to topics of interest. **Key characteristic of pub/sub pattern** is that published msgs are **not directed to any specific subscriber.** A broker facilitates msg distribution.
**Kafka in Pub/Sub:** Open-source pub/sub broker cluster manager by LinkedIn
- Manages a cluster of brokers, each broker contains its own set of topics for its own domain, each topic is further partitioned by key for further scalability.
- Consumer groups have multiple consumers that collectively sub to a topic, each consuming msgs from a selected partition for load-balancing.
- Kafka maintains a log of events and consumers individually keep track of an offset to know which part of the data stream has been consumed.

**Asynchronous Communication Summary:**
- Single receiver involves point-to-point communication.
- Multiple receivers use a pub/sub mechanism.
- Message queuing systems support asynchronous persistent communication.

**Persistent vs. Transient Communication (Slide 34):**
- **Persistent**: involves storing messages at each hop, guaranteed delivery
- **Transient**: buffers messages for short periods, discarding if undeliverable.

**L9 – Messaging Patterns / Enterprise Integration Patterns**



**Message Construction – Types and components of a message**
**Command Message** – specifies a function/method to invoke on receiver (CQRS?)

**Document Message** – contains information/content of a data structure
**Event Message** – A notification. Has timestamp, content not really relevant
**Return Address** – For the receiver to know which reply channel to reply to.
**Correlation ID –** So we know which reply corresponds to which request.
**Message Channels** – A channel between sender and receiver to communicate.
**Point-to-point ch**: Request is processed by a single consumer
**Pub-sub ch:** Request is broadcast to all interested parties
**Datatype ch**: for sender to let receiver know how to handle a given data item
**Invalid Message ch**: for **receiver** to handle invalid message (e.g., stderr)
**Dead letter ch**: for **message broker** to handle undeliverable messages
**Message Routing:**
Message Routers **consume** messages from one channel and **reinsert** them into different message channels depending on a set of **conditions**.
**Simple Routers**: route message from an inbound ch to one or more outbound ch
**Composed Routers**: combining many simple routers to create complex msg flows
**Context-based**: routes based on context, such as failover, server-load, testing
**Content-based routing**: Performs message routing based on content of message.
**Message Filter**: A special kind of content-based router – removes undesired msgs
**Splitter**: Splits message into multiple messages, which can be routed accordingly
**Aggregator**: Special filter that receives a stream of messages, identifies correlated messages, and publishes a single aggregated message for further processing.
**Message Scatter-Gather:** Routes a single message to multiple participants concurrently and aggregates the replies into a single message. Ideal for requesting responses from multiple parties and processing that data
**Message Transformation**
**Translator**: Converts messages from one format to another
**Canonical Data Model**: Translate data to a shared format first, before translating to the receiver system's format – reduces the need for many P2P translators
**Message Endpoints** – interface between an application and message system
- Can be used to send messages or receive them, but not both.
- Endpoint is channel specific, so a single application would need multiple endpoints to interface with multiple channels.

**Polling consumer**: A application controls when it consumes a message; proactive
**Event-Driven consumer**: event delivery triggers receiver into action; reactive
**L10 – Principle-Pattern Entwinement**
**Modularity:** A universal principle used to manage complexity by dividing big systems into smaller, self-contained modules. Modules are deployable, manageable, cohesive, possibly reusable units of sw with well-defined interfaces &boundaries.
**Program to Interface:** Depend on interfaces to decouple from implementation
**Favor Composition over Inheritance**: Inheritance inherits undesirable features
**Encapsulate (& Separate) what varies:** If a component is bound to change frequently, then it's good practice to separate this part of code so that we can later extend or alter the part that varies without affect those that don't.
**Design Patterns:** A solution to a problem in a context.
- **Context** is the recurring situation in which the pattern applies
- **Problem** is the goal to be achieved in the context, inclusive of any constraints
- **Solution** is the pattern. A **general design** applicable to the problem in context

**Creational Patterns**: Provide ways to instantiate single obj or groups of objs
- Encapsulate/hide details about the actual creation

**Structural Patterns** provide ways to define relationships between classes or objs in order to form larger structures; how these components should be structured so that there is flexibility in interconnecting modules
**Behavioral Patterns** define manners of communication between classes and objs; Simplifies communication between objs and make it more understandable