1 Software Development Process

1.1 Types of Software Applications

1. **Embedded** - An embedded system is a microprocessor-based computer hardware system with software that is designed to perform a dedicated function, either as an independent system or as a part of a large system e.g. Cameras, MP3, Calculators
2. **Real-Time** - "real-time system" refers to any information processing system with hardware and software components that perform real-time application functions and can respond to events within predictable and specific time constraints e.g. video conferencing, online gaming
3. **Concurrent** - Concurrent systems are systems comprising a collection of independent components which may perform operations concurrently e.g. web servers that service many different connections
4. **Distributed** - computing environment in which various components are spread across multiple computers e.g. Telephone and cellular networks
5. **Open content systems** - Source code not released to public but content is publicly available e.g. Wikipedia, FaceBook, YouTube, Twitter, Flickr
6. **Open Source System** - Source code is openly available e.g. Linux, Apache

1.2 Variations

- **Requirements**: Known vs Emerge from crowd
- **Process**: Planned increment vs No stable state/version
- **Resources**: Dedicated and managed vs Adhoc resources
- **Contributors**: Controlled vs influenced

1.3 Edge Computing

- Edge computing is a distributed computing paradigm that **brings computation and data storage closer to the sources of data** e.g. IoT devices
- Balance the demands of centralized computing and localized decision-making

1.3.1 Decentralized vs Centralized Computing

- **Decentralized Computing** - Everyone has a computer, everyone is computing something
- **Centralized Computing** - Type of computing architecture where most/all of the processing is down on a central server e.g. SoC Compute Cluster, IBM Mainframe
- Cloud computing is a mixture of decentralized and centralized computing

1.3.2 Motivation for Edge Computing

- Cloud-based applications have a **significant amount of latency** as there is a time delay between request and response from server
- For many critical tasks that **need immediate computation** for e.g. Tesla's auto driving system, we cannot afford the latency and hence computation must be done at edge

1.3.3 Limitations of Edge Computing

- **Limited computation power** in edge computers
- Many of devices at edge are not connected to a power source ⟶ **issues of battery life and heat dissipation**
- Typical solution to these problems is to combine the power of cloud and edge computing together

1.4 Software Development

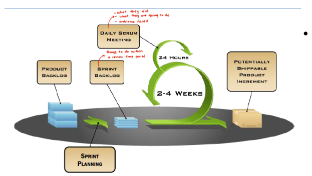
1.4.1 Factors affecting Software Development

- **Requirements** - known at the start? determines complexity of software
- **Process** - Waterfall? Agile? What resources and time is needed for the process?
- **Criticality** - Strict deadline? Experimental? What are the consequences of failure?
- **People/Resources** - What resources are available? What are the competence of the engineers? What technologies are available?

1.4.2 Process Models

Waterfall, Spiral, Rapid Programming, eXtreme Programming, Rational Unified Process (RUP), Test Driven Development, Agile (Scrum, Crystal etc.)

1.4.3 Agile Process



- Agile is a **methodology/ideology** that emphasizes iterative development and cross-functional collaboration
- Done using **SCRUM**.
  - Work is done in sprints, where a subset of the product backlog is cleared
  - Project Backlog is a piece of document noting down the FRs, along with their priority and the sprint to be completed in, typically **done after user stories**
  - Often have a 15 min scrum meeting at the start of the day to discuss what is done, what is in progress and what is to be done

1.5 Software Delivery

- **Deployment pipeline**: an automated implementation of application build, deploy, test, and release process
- Typical development pipeline: Dev ⟶ Test ⟶ User acceptance ⟶ Production

1.5.1 CI/CD Pipeline

- **Continuous Integration** - development practice that requires developers to integrate code into a shared repository several times a day. Each check-in is then verified by an automated build (typically in a staging env that is very similar to production env), allowing teams to detect problems early
- **Continuous Delivery** - ensuring that every good build is potentially ready for production release, manual deployment to production
- **Continuous Deployment** - automating the release of a good build to the production environment

1.5.2 DevOps

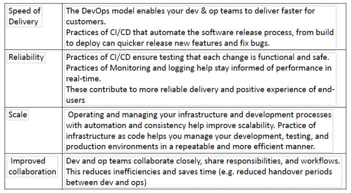
- DevOps is a set of **software development practices/process** that combine software development (Dev) and operations (Ops)
- intends to reduce the time between committing a change to a system and the change being placed into normal production while ensuring high quality
- It is the intersection of Dev, QA and Technology Ops

DevOps Pipeline

- Push code ⟶ Pull code and build ⟶ Test ⟶ Store artifacts and build repo ⟶ Deploy and release ⟶ Config Env ⟶ Update DB ⟶ Update apps ⟶ Push to users⟶ App and Network Performance monitoring

DevOps Benefits

- Uses CI/CD, Continuous Monitoring/Logging, Communication and collaboration and IaaS (e.g. GitHub Actions) to achieve the following benefits:



Speed of Delivery	The DevOps model enables your dev & op teams to deliver faster for customers. Practices of CI/CD that automate the software release process, from build to deploy can quicker release new features and fix bugs.
Reliability	Practices of CI/CD ensure testing that each change is functional and safe. Practices of Monitoring and logging help stay informed of performance in real-time. These contribute to more reliable delivery and positive experience of end-users
Scale	Operating and managing your infrastructure and development processes with automation and consistency help improve scalability. Practice of infrastructure as code helps you manage your development, testing, and production environments in a repeatable and more efficient manner.
Improved collaboration	Dev and op teams collaborate closely, share responsibilities, and workflows. This reduces inefficiencies and saves time (e.g. reduced handover periods between dev and ops)

2 Specifying Software Requirements

2.1 What is a Requirement

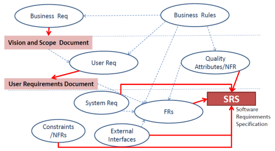
- condition or **capability needed by a user** to solve a problem or achieve an objective
- **capability that must be met or possessed by a system**
- **Specification** of what should be implemented
- Can be thought of as a contract of how a system should behave (functionalities)

2.1.1 Examples of Requirements

- **User stories**
  - User centric
  - Format: "As a {role} I want to {functionality} so that I {what is achieved}"
  - Could include constraints to bound the requirement even tighter (**NFRs**)
- **Non-user story format**
  - Product centric
  - Danger of giving users functionality that they did not request for, need to ensure that it is in sync with what users want
  - Typically have a better or more refined description
  - e.g. The system will support the following file formats: png, jpeg, bmp, tiff and gif
- Class Responsibilities Collaborator (CRC) Model
  - Follows OOP
  - Uses a collection of stand index cards divided into 3 sections
  - **Class**: represents collection of similar objects
  - **Responsibility**: something the class knows/does
  - **Collaborator**: another class that a class interacts with to fulfill its responsibilities
- Specifying Microservices Requirement
  - Typically in a table format with Name of MS, Description, Capabilities, Service API (Async, sync calls, commands, events published), NFRs
  - Should contain Consumer Tasks, Interface, Qualities, Logic/Rules, Data, Dependencies

2.1.2 Types of Requirements

- **Business Requirements**: Why the organization is implementing the system (recorded in vision and scope), e.g. reduce staff costs by 25%
- **User Requirements**: Goals the user must be able to perform with the product, e.g. check for a flight using the website
- **System Requirements**: Hardware or software issues, e.g. the invoice system must share data with the purchase order system. Affects functional requirements
- **Quality Attributes/NFR**: How well the system performs, e.g. mean time between failure ≥ 100 hours. A type of non-functional requirement (code could be involved)
- **Functional Requirements**: The behavior the product will exhibit, describes what developers must implement, e.g. passenger shall be able to print boarding passes.
- **Constraints**: Limitations on design and implementation choices, e.g. must be backwards compatible. A type of non-functional requirements
- **Data Requirements**: Describes data items or structures, e.g. product number is alphanumeric
- **Business Rules**: Policies/regulations/guidelines/standards. Constraints business, user and functional requirements e.g. staff gets 30% off
- **External Interfaces**: Describes connections between your system and outside world, e.g., must import files in CSV format

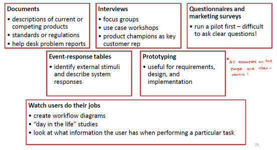


2.2 User (URS) vs Software (SRS) Requirement Specification

- User requirement describes the **end-user requirements** for a system
- Software requirements describes **what the system must do**
- SRS could include URS but URS will not include SRS
- SRS needs to be accurate, complete, consistent, testable, modifiable, ranked, traceable, verifiable and valid
- URS audience - End-users, SRS audience - Developers, specification of Acceptance Testing

2.3 Sources of Software Requirements

- Note that all e.g. below are user-centric



Documents
- descriptions of current or competing products
- standards or regulations
- help desk problem reports

Interviews
- focus groups
- use case workshops
- product champions as key customer rep

Questionnaires and marketing surveys
- run a pilot first – difficult to ask clear questions!

Event-response tables
- identify external stimuli and describe system responses

Prototyping
- useful for requirements, design, and implementation

*All examples on this page are user-centric!

Watch users do their jobs
- create workflow diagrams
- "day in the life" studies
- look at what information the user has when performing a particular task

2.4 Quality Attributes

2.4.1 External Quality Attributes

- **Availability**: Measure of the planned up time during which the system is fully operational
- **Installability**: How easy it is to install the system for the end-user
- **Integrity**: Preventing information loss and preserving data correctness
- **Interoperability**: How readily the system can exchange data and services with other software and hardware
- **Performance**: Responsiveness to user actions. May also compromise safety if e.g. a safety system responds poorly
- **Reliability**: Probability of software executing without failure for a specific period of time
- **Robustness**: Degree to which a system performs when faced with invalid inputs, defects and attacks
- **Safety**: Prevents injury or damage to people or property
- **Security**: Authorisation, authentication, confidentiality, etc
- **Usability**: User-friendliness and ease of use

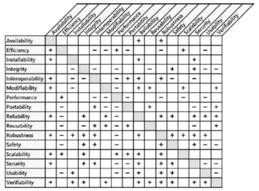
2.4.2 Internal Quality Attributes

- **Efficiency**: How well the system utilises the hardware, network etc
- **Modifiability**: How easily designs and code can be understood, changed and extended
- **Portability**: Effort needed to migrate the software from one environment to another
- **Reusability**: Effort required to convert a software component for use in other apps
- **Scalability**: Ability to grow to accommodate more users, servers, locations, etc. without compromising performance or correctness
- **Verifiability**: How well the software (components) can be evaluated to demonstrate that it functions as expected

2.4.3 Quality Attributes Requirements of Different Systems

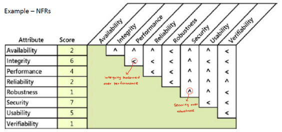
- Embedded software: performance, efficiency, reliability, robustness, safety, security
- Internet Application: availability, integrity, interoperability, performance, scalability, security, usability
- Desktop/Mobile software: performance, security usability

2.4.4 Quality Attribute Tradeoffs



	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	+		+	+		+			+	+	+	+		+	+	

Example – NFRs



Attribute	Score	Availability	Integrity	Performance	Reliability	Robustness	Security	Usability	Verifiability
Availability	2		^	^	^	<	^	^	<
Integrity	6			<	<	<	^	<	<
Performance	4				<	<	^	^	<
Reliability	2					<	^	^	^
Robustness	1						^	^	<
Security	7							<	<
Usability	5								<
Verifiability	1								

integrity preferred over performance

Security over robustness

2.4.5 Safety vs Security

- **Safety**: About whether a system can harm someone or something physically, financially, reputation etc. e.g. User should be able to see list of options with ingredients highlighted in red that are known to cause allergic reaction
- **Security**: about privacy, auth and integrity. e.g. Is data protected from disclosure, is system only accessible to authorized users.
  - Security breaches **typically involve some sort of financial harm**

2.4.6 Validation vs Verification

Validation is about whether you have the **right requirements** and if they **trace back to the business objectives**. Verification is whether you have written the requirements right i.e. complete, correct, feasible, priority unambiguous. Can be checked informally by passing the requirements around or formally through formal inspection

3 Software Architectures

Software Architecture design represents the structure and relationship of data and program components that are required to build a software. It helps to form some sort of **pattern** which allows certain type of application to reuse these architectures e.g. MVC. Building blocks of software architecture includes:

- **Component**: models an application-specific function
- **Connector**: models Interactions among components for the purpose of transfer of control and/or data
- **Configuration**: Topology or structure

3.1 Basic Concepts & Definitions

- Control Flow
  - Computation order
  - An arrow from 1 block in architecture diagram to another block typically means that there is some sort of sequence where the latter is performed after the formal block
  - How the focus of control moves throughout the execution. Data may accompany control. e.g. in a CI pipeline, control is passed when it moves from `Build` to `Test`
- Data Flow
  - data availability, transformation, latency
  - How data moves through a collection of computations
  - As data moves, control is activated ⟶ Data and Control flows are interlinked

3.2 Call and Return

1. **Hierarchical** (synchronous)
   - control moves from one component to another and back
   - consists of main program and procedures (possibly also sub-procedures)
   - "use" relation between components ⟶ arrow direction indicates which components calls what other component
   - **Single thread of control**
   - **Hierarchical decomposition**: correctness of a procedure depends on correctness of its sub-procedures
2. **Non-hierarchical**
   - Components communicate via Event notifications / Message passing / RPC and other protocols (could be async / sync, P2P or published)
   - No strict hierarchy between components

- **Pros**: Easy to modify, easy to extend by adding modules, easy to analyze control flow
- **Cons**: Hard to parallelize, hard to distribute, awkward to handle exceptions

3.2.1 Message vs Event Driven System

- **Message**: typically some data sent to a **specific address**
  - each component has a unique address other components can send messages to
  - will await message and respond to them
- **Event**: data emitted from a component for anyone listening to consume (**consumer must subscribe to event**)
  - Event typically emitted when state changes
  - **immutable** (cannot be changed or deleted)
  - **ordered in sequence of creation**

3.3 Common Architecture Style

- **Horizontal Slicing**: designing by **layers** e.g. Controller ⟶ Services ⟶ Repository/Model
- **Vertical Slicing**: designing by **feature** e.g. putting all things related to Orders together
- **Ports and Adapters**: keep domain-related code separate from technical details
- **Presentation-Domain-Data Layering**: Separate into 3 layers - Presentation (UI), domain logic (business logic), data access (DB). **Web application** typically uses this.
  - **Only can be applied in relatively small granularity**, once application grows need to apply more modularization

3.3.1 Pipe and Filter

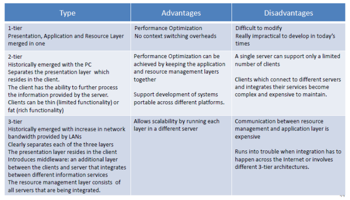
- Data oriented architecture where **control is distributed**
- Data enters the system and flows through components one at a time until data sink. Series of transformations on successive pieces of input data happens in the process
- **Components**: Filters, Data Source, Data Sink. Each component has a set of I/O
- **Filter**: transforms input streams, computes incrementally (runs when it has necessary data) ⟶ output begins before input is consumed, **independent** (share no state) ⟶ allows for pipe-lining and for filters to be added/removed/replaced easily
- **Connectors**: Pipes ⟶ transmits outputs of one filter to another, does not do anything to the data, buffers data if needed, are **first-class components**
- **Pros**: No complex component interactions, easy to reuse, parallelized, maintain and enhance
- **Cons**: Not good at interactive application, has pack and unpack cost (e.g. packets)

3.3.2 Implicit Invocation (Event Driven)

- Individual components announce data they will publish. Other components will subscribe to the events. When data appears, subscribers are invoked (implicit as caller does not directly call on method of the callee)
- Very commonly used in GUI, when form is filled, validator is implicitly called
- **Components**: Interfaces provide both a collection of procedures (as with abstract data types) and a set of events
- **Connectors**: procedure calls, bindings between event announcements and procedure calls
- **Pros**: No need to know name of subscribers, easy to parallelize, decouple control, real-time, can easily replace components
- **Cons**: Complex implementation, hard to test, subsystems don't know when or if events are available, sequence not determined, cannot assume other component will respond

3.3.3 Layered Architecture

- Independent development and evolution of different system parts. Each layer has a distinct and specific responsibility. **Mostly organized as horizontal layers** e.g. Presentation ⟶ Business ⟶ Data layer. Focuses on **technical domain**
- An **open-layer architecture** means that each layer can use the services of any layers below it while **close-layer architecture** means that each layer can only use the service of the layer directly below it



Type	Advantages	Disadvantages
1-tier Presentation, Application and Resource Layer merged in one	Performance Optimization No context switching overheads	Difficult to modify Really impractical to develop in today's times
2-tier Historically emerged with the PC Separates the presentation layer which resides in the client. The client has the ability to further process the information provided by the server. Clients can be thin (limited functionality) or fat (rich functionality)	Performance Optimization can be achieved by keeping the application and resource management layers together Support development of systems portable across different platforms.	A single server can support only a limited number of clients Clients which connect to different servers and integrates their services become complex and expensive to maintain.
3-tier Historically emerged with increase in network bandwidth provided by LANs Clearly separates each of the three layers The presentation layer resides in the client Introduces middleware: an additional layer between the clients and server that integrates between different information services The resource management layer consists of all servers that are being integrated.	Allows scalability by running each layer in a different server	Communication between resource management and application layer is expensive Runs into trouble when integration has to happen across the Internet or involves different 3-tier architectures.

3.3.4 Data abstraction and O-O organisation

- **Component**: ADT (Abstract Data Types) or Object that encapsulates data representations and their associated operations e.g. KWIC indexing
- **Connector**: Interactions that enable procedure(or method) invocations
- **Pros**: object hides its representation from its clients, allows designers to decompose problems into collections of interacting components
- **Cons**: object must know the identity of the other object (not that bad in practice), **side effects**: if A uses B and C also uses B, then C's effects on B can result in unexpected side effects to A, and vice versa (unexpected coupling)

3.3.5 Shared Repository

- Maintain all data in a **central repository** shared by all functional components of a data-driven app. Let the availability, quality, and state of that data **trigger** and **coordinate** the control flow of the application logic (e.g. Repo architecture of IDE)
- **Components**: central data structure that represents current state, collection of independent components that act on central data store
- **Connectors**:Interactions between the repository and the other components
- **Pros**: Independent clients, decouples data storage from manipulation, east to add clients, can share large amount of data
- **Cons**: Communication between clients are slow, subsystems must agree on DB structure

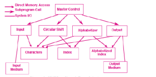


Figure 6: KWIC – Shared Data Solution

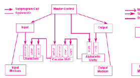


Figure 7: KWIC – Abstract Data Type Solution

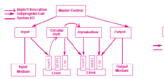


Figure 8: KWIC – Implicit Invocation Solution

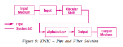


Figure 9: KWIC – Pipe and Filter Solution

3.4 Types of Cohesion

- **Functional** (best, only performs 1 calculation w/o side effects), **Layer** (related services kept together), **Communicational** (work on same data), **Sequential** (work in sequence), **Procedural** (called 1 after another), **Temporal** (same phase of execution), **Utility** (weakest, related utilities)

3.5 Types of Coupling

Type	Comment
Content (tightest)	Component modifies internal data of another
Common/global	Modules using shared DB or same global variable
Control	One module (A) directly controls the other (B) by passing information on what B should do;
Data	One module sharing data with another module; e.g., via passing parameters
External	A dependency exists to elements outside the scope of the system, such as the operating system/shared libraries/hardware
Temporal	Two actions are bundled together just because they happen to occur at the same time
Inclusion/import (weakest)	Including a tile or importing a package

3.6 Architecture Diagrams

- Shows the "big picture" of what is being built, a shared vision of the dev team, used as a "map" to navigate source code, useful for onbaording new devs to the team.
- **Common mistakes**: notation (colors, shapes) is inconsistent or not explained, purpose and meaning of element is ambiguous, relationship are missing, generic terms used, technology choices omitted, levels of abstraction mixed, too much/too little detail, no logical starting point
- "Software System" is the highest level of abstraction and represents something that delivers values to users.
- "Container" is a context or boundary inside which code is executed / data stored. Should be separately deployable
- "Component" groups related functionality and are not separately deployable units. All components inside a container typically execute in same process space. e.g. JAR files, DLL, shared lib

4 MV* Architecture

4.1 MVC

- Split up into **View** (GUI/CLI or any output), **Controller** (acts as mediator, controls what view can show) and **Model** (Persistence + Business Logic, typically most independent)
- View and controllers can be played by same object when it's very tightly coupled
- Variant where input goes through view is common (e.g. filling up a form)
- Controller acts as listener (to user request and propagate to model), View acts as observer and Model acts as observable
- Benefit of **Separation of Concern** (modularity, output separated from input handling, output and input separated from application state (model) and transaction processing) and **facilitating extensibility** (multiple views can be created from same model, new functionality can be added to the model independently of other components)

4.2 MVA (Model-View-Adapter) / Mediated MVC

- All communication between Model and View flows through Controller (**completely decouples view and model**), Controller (adapters/mediators) becomes communication hub
- Possible to have different adapters doing different business logic

4.3 MVP (Model-View-Presenter)

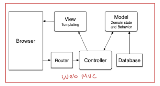
- Popular in .NET world where **Model** represents business entities/domain models/object model of business tier, **View** is a lightweight UI only component who isn't aware of the model and **Presenter** presents user actions to backend system and response to user
- View could be further divided in to **Passive View** (view doesn't know model) and **Active View** (data binding or simple code in view). Only has 1 presenter

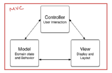
4.4 Presentation Model (PM) and Model-View-View-Model (MVVM)

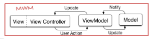
- PM is well suited for rich UI application / advances in UI technologies
- View elements are directly bound to properties on the model ("1-to-1" binding between view and model)
- **Model** represents the state of the view and might contain UI elements specific properties which will be rendered once model is constructed. **View** is lightweight and simple and is used to publish events raised by user to Presentation Model. **Presenter** receives event from view, updates the model and any state changes and call the view to render
- MVVM is very similar to PM except that it substitutes Presenter with a View-Model. Used in cases like live word count updates, loading screens and validation of form fields
- MVVM uses commands and data binding to communicate

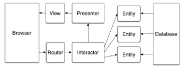
4.5 Web MVC

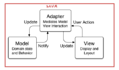
- Very similar to normal MVC except that the controller has a new responsibility of handling the initial HTTP request. Used by frameworks like Spring, Ruby on Rails, PHP, ASP.net
- Controller does not do much computation and just serves HTML pages, View could use data from Model to generate static pages, browser only renders what View sends it
- New **Single Page Application** Paradigm introduced an extra step in creating static bundles of HTML, JS and CSS for direct hosting via a **View Controller**. SPA applications could have the entire MVC in the browser and include logic for making HTTP API requests to API controllers who responds in JSON format











4.6 nginx case study

- nginx has NFRs in Performance, Load (density) and Economic resource utilization. It use a Modular, event-driven, asynchronous (want to waste less time and resource busy waiting), single-threaded, non-blocking architecture
  - Uses multiplexing (packing multiple request into 1) and event notifications and dedicates specific tasks to separate processes
- Workers accepts new requests via shared "listen" (all HTTP requests captured by listener), run-loop relies on async task handling implemented through modularity, event notifications and extensive use of callback functions. Aims to be **as non-blocking as possible** (can't be completely non-blocking due to disk I/O when disk is filled)
- No process/thread per connection (unlike web servers) which makes memory usage very conservative and efficient. Also no create-destroy cycle which save CPU cycles
- High scalability due to horizontal scaling on workers to handle more connections → scales across multiple cores
- Efficient due to caching (reduce I/O ops) in the form of hierarchical data storage on file system and is a shared memory accessed by worker, cache loader and manager

4.7 Twitter AdServer Case Study

- Key considerations: build quickly, iterate frequently, profit oriented
- Started off with monolithic as they wanted reliability, deployability and performance and wanted a Cohesive Monolithic Adserver (AdMixer + AdShart)
- Faced problems when they wanted to extend to other forms of ads (needed to replicate AdServer for each type of ad which is an Anti-Pattern since each vertical lacks cohesion and there is no reusability). Had troubles reusing existing patterns in legacy code and needed patchy updates. Also had tightly coupled codebase which increased complexity and had spaghetti code
- Moved towards microservices by identifying "tech functions" like Candidate Selection, Candidate Ranking and Analytics
- Horizontal components are now shared across verticals which led to more cohesive services, decoupling and isolation of verticals and had better separation of concerns
- Horizontal services now each serve different kind of ads and it is easily extensible which led to faster development, troubleshooting and debugging
- Downside of microservice was that it increased hardware and operational costs and had to redesign the entire system which took time.

5 Microservices

5.1 Modularization

- **Module**: deployable (if it's a library), manageable, unit of software that provides a concise interface to its clients (does not expose its internals)
  - Code that does not expose its API i.e. Dead code, should not be a module!
- Modularization is the process of **separating the functionality** of a program into independent modules, such that each contains everything necessary to **execute only one aspect of the desired functionality**

5.1.1 Modular Application

- **Monolith**: single application deployed as a single process
- **Modular Monolith**: Monolith composed of loosely coupled, highly cohesive modules
- Splitting application into independent modules allows for **easier development, better testability and maintainability** and **allows for reusability** (requires experience to know what can be reused)

5.1.2 High Cohesion, Loose Coupling

- loosely coupled module **knows as little as it needs to** about the modules with which it interacts
  - want to limit number of different types of calls between modules → chatty communication leads to performance issues and tight coupling
- want **related behaviors to be grouped together** and unrelated behavior elsewhere → leads to high cohesion
- want to find **boundaries** to ensure related stuff are in 1 module and communicates with other modules as loosely as possible

5.1.3 Design Principles

Abstraction	Encapsulation	Single Responsibility Principle
Low Coupling	High Cohesion	Liskov Substitution Principle
Information Hiding	Separation of Concerns	Open Close Principle
Interface Segregation	Dependency Inversion	Program to Interface

- **Information Hiding**: Guides you on how to define boundaries
- want to separate code that change frequently from static code
- Hide code that changes so that we can make changes internally without affecting compatibility
- **Encapsulation**: binding one or more things into 1 boundary
- Use visibility to hide parts of implementation

5.2 Command-Query-Responsibility Segregation (CQRS)

- **pattern** relying on separation of **commands** (writes) from **queries** (reads)
- **Command**: operations that change application state but returns no data, will leave side effect within application
- **Queries**: return data but don't change application state
- Models for read operations and write operations are **completely separated** (could be using 2 different tables for SQL DB or could use 2 entirely different DB e.g. SQL and NoSQL) → allows for **individual scaling/caching of reads/writes**
- **Command model will go through the domain model** since it is modifying the state (may require some manipulation of data and it requires the entire transactional entity) but Query model can **go directly to persistence layer** to fetch data
- If implemented using 2 different data stores, will **require some sort of synchronization** between write store and read store so that when you read data it is updated
- Follows the **Separation of Concern**, **Single Responsibility** and **Interface Segregation**
  - responsibility of handling writes separated from reads
  - Each model does one thing only, either read or write
  - Client either talks to read or write interface

5.3 Emergence of Microservices

- Defined as an ***independent, standalone capability*** designed as an executable that ***communicates with other microservices*** through standard lightweight inter-process communication such as HTTP, message queues, etc.

	Advantage	Disadvantage
Monolith	1. Simple to develop - many Dev Tools and IDEs support monolith 2. Simple to deploy 3. Simple to scale - run multiple copies behind load balancer	1. Large Codebase - intimidates new devs, must coordinate effort and redeployments, long time commitment to tech stack 2. Frequent deployment is difficult - have to redeploy entire app 3. Scaling can be difficult - can't scale components individually, makes caching less effective, increase I/O and memory consumption
Microservices	1. Organized around business capabilities 2. Owned by small team 3. Independently Deployable 4. Productivity remains relatively high with complexity	1. Increase complexity and larger codebase 2. Increase complexity to orchestrate deployment 3. Possibly increased network traffic from interservice communication

Domain-Driven Design	Continuous delivery	Infrastructure automation	On demand Virtualization	Small autonomous teams	Systems at Scale
•helped us understand the importance of representing the real world in our code, and • showed us better ways to model our systems.	•showed how we can more effectively and efficiently get our software into production, and • instilling in us the idea that we should treat every check-in as a release candidate.	•led us to develop better ways of having machines talk to other machines.	•allowed us to provision and resize our machines at will, and with infrastructure automation giving us a way to handle these machines at scale.	•Amazon and Google espoused the view of small teams owning the full lifecycle of their services.	•Netflix shared with us ways of building antifragile systems at a scale that would have been hard to comprehend a decade or two ago.

5.4 Characteristics of Microservices

5.4.1 Domain Driven Design (DDD)

- DDD does not dictate any specific architecture style, only **requires model to be isolated from technical complexities** so that it can **focus on domain logic concerns**
- **Domain**: critical and fundamental /foundational concept behind the business
- **Ubiquitous Language**: A shared language between domain experts and developers that uses domain-specific terminology
- **Subdomain**: is a logical "separation" of the domain, is in the problem space
  - **Core** - "must have functions", key differentiator of the business
  - **Supporting** - "nice to haves", related to business but not differentiator, outsources or implemented in-house
  - **Generic** - not specific to business, typically outsourced
- **Bounded Context**: Divide complex and large domain into separate bounded context which is a explicit boundary within which a domain model exists
  - Bounded context refers to the solution space → solves problem from sub-domain
  - 1 bounded-context can contain **multiple sub-domains** and is the **actual implementation of sub-domains**
  - Inside the boundary, **all terms and phrases of the ubiquitous language have specific meaning** → customer in a Sales Context can mean something different from a customer in the Support Context
  - Follows **SoC** and **SRP** as each part of the system has its own intelligence, data and vocabulary and is independent of each other

5.4.2 Microservices

- **verb/use cases** responsible for particular actions
- **nouns/resources** responsible for operations on entities/resources of given type
- Use **subdomains** that corresponds to different parts of the business
- Use **Bounded context** that have **physical boundaries** and are **owned by a single team**
  - One team can own multiple microservices
  - Microservices are bounded context but all bounded contexts are not microservices → bounded contexts sometimes have wide functionalities
- Each service should **ideally only have small set of responsibilities** → apply SRP
- Aim to target services that encompasses entire Bounded Context

5.4.3 Characteristics of Microservices

1. Organized around business capabilities
   - Looks at what functionality does it provide and what data does it need?
2. Loosely coupled
   - **Aggregates**: a **self-contained unit focused on a single domain** concept in the system
   - **Bounded contexts**: represents a **collection of aggregates**, with an explicit interface
   - Use **information hiding** and **async communication** when possible!!
     - **Inside-out thinking** - Specify an API for external services to interact
     - **Outside-in thinking** - External service specifies an interface for internal service to implement
3. Owned by small (cross-functional) team
   - When teams are less coupled, product also tend to be less coupled (Conway's Law)
4. Independently deployable
   - Each microservice has its own deployment, resource, scaling and monitoring requirements
   - **Service per Host** (e.g. EC2) VS **Service per Container** (e.g. Docker)

5.4.4 Services and their Database

- **Database per service pattern**: Each service has its **own database schema** which often leads to data duplication but is however needed for loose coupling. Can choose database suited for service

5.4.5 Service Communication

- Inter process communication can either be async or sync
  - **Request/sync response** - client makes request to service and waits for response
  - **Notification** (one way request) - client sends request to service but no reply is expected
  - **Request/ Async response** - client sends request to service which replies asynchronously

5.4.6 API Gateway

- API Gateway acts as a server that is the **single entry point** into the system
- It encapsulates the internal system architecture and provides an API tailored to each client → client does not have to directly communicate to the services and offloads routing workload to API gateway
- Could have other responsibilities like authentication, monitoring, load balancing and caching
- Follows the **facade pattern**!

5.4.7 Orchestration vs Choreography

- **Orchestration**: rely on a central brain to guide and drive the process
  - central brain could be a bottleneck and a single point of failure
- **Choreography**: inform each part of the system of its job, and let it work out the details
  - more decoupled but more work if monitoring is needed

5.4.8 Service Discovery

- API Gateway needs to know the IP address and port of each microservice
- Could either do Client-side Discovery, Server-side Discovery or a Service Registry
  - Client-side Discovery: **Client queries service registry** which maintains a database of services instance (if service terminates than it is removed from registry). Client then determines network location of available service using a load balancer (**couples client with service registry**)
  - Server-side Discovery: Client **makes request to service via load balancer** which queries service registry and routes each request to an available service instance (**decouple client from service registry but require manual implementation of load balancer**)
  - Service registry: is a database of services, their instances and locations, **requirement of being highly available and up to date**
- Allows services to register itself and allows services to be discovered by the registry

How small is small ? 10-100 LOC services ?	Communication Complexity	Architectural Complexity	Change Complexity	Deployment Complexity
•Size is not the primary goal. The goal: to sufficiently decompose the application to facilitate development and deployment.	•much more complex than in a monolithic application (inter-process communication – sync/async; handle partial failures)	•Partitioned database architecture (eg distributed transactions pose the challenge of keeping consistency)	•Implementing changes that span multiple services (not very common; but needs careful plan and coordination)	•Deployment complexity (many more moving parts that need to be configured, deployed, scaled, and monitored) •Need a service discovery mechanism

6 Messaging Patterns

Messaging systems or Message Oriented Middleware (MOM) are pieces of software that provides messaging capabilities. All Distributed Application Design (DAD) needs a MOM.

- 3 axes of component/application communication: **sync vs async**, **single vs multiple receiver**, **persistent vs transient**
- Persistent Async (Email), Persistent Sync (Messaging/Chat), Transient Async (UDP), Transient Sync (RPC), Sync Single (RPC, REST), Async Single (Polling, P2P), Sync Multiple (Webhooks), Async Multiple (PubSub)

6.1 Communication Types

6.1.1 Synchronous vs Asynchronous

- **Synchronous**: components exchanges information at the same time with each other. a.k.a. "1:1" or "request-reply" communication.
  - Tight coupling, connection overhead, sender has to handle failures
- **Asynchronous**: communication has lag between message sent and receiver response. Enables independent functioning of receiver and sender and 1:n communication
  - Notification need to be sent and sender needs to remember context which message was sent

6.1.2 Remote-Procedure Call (RPC)

- RPC mimics the serial thread of execution that a "normal" non-distributed system would use, each statement is executed in sequence (**sync** in nature)
- Success of one RPC call depends on success of all downstream RPC call ("all or nothing")
- Type of **inter-process communication** to make a procedure execute on another address space (typically another computer in shared network)
- Client call to procedure → Stub builds message (Marshalling) → Message sent via network → Server OS hands stub message to server stub and unpacks it (Un-marshalling) → Stub makes local call to method

6.1.3 HTTP/HTTPS

- Is a **sync request-reply** pattern
- In the event that backend processing needs to be asynchronous but requester needs a clear response, one possible way to circumvent it is to keep polling status endpoint

6.1.4 Asynchronous Message-passing

Async and persistent, with intermediate storage (message queue) while sender and receiver are not active. **Can be single** (P2P, exactly 1 consumer and message processed once)/**multiple** (pub-sub) receiver. Messages guaranteed to be inserted into the queue but no guarantee on when or if message will be read.

6.1.5 Persistent vs Transient

- Persistent (store-and-forward): messages stored in intermediate hop, receiver **guaranteed to receive message**. Has 4 stages - send, received, read, processed (e.g. Emails)
- Transient: message buffered **only for a short period of time** (while sender/receiver are executing). Discarded if cannot be delivered or host is down (e.g. TCP/IP)

6.2 Messaging Pattern

Messaging systems constructs, transports, routes, transforms, produce and consume messages. Also manage and test the system.

6.2.1 Message Construction

- Each message has a **header** (metadata), **properties** (optional) and **payload** (data).
- Encapsulates **method requests** and **data structure** to be sent. Header specifies the type of information transmitted, origin, destination, size and other structural information
- 3 types of message intent: **Command message** (telling consumer what to do), **Document message** (just sending data), **Event message** (Notification, have time constraint, cannot have too high latency)

6.2.2 Message Channel

- Connect sender and receiver using a **queue** that allows them to exchange messages
- Request/reply channel: channel transmit message in **1 direction**. If we want two-way message, need a channel for each of reply and request
- Contains **return address** (to tell replier where to send reply to), **correlation ID** (specifies which request the reply is for. Contains requestor, replier, request, reply, request ID and correlation ID. **Can be chained** → allows for retrace of message from latest reply to original request)
- **Point to Point** (P2P): request processed by single consumer→ uses message queues
- **Publish-Subscribe**: used when **multiple parties** are interested in certain messages. Messages published to a topic is **immediately received** by all subscribers (unless message filter is applied)
  - beneficial when it is important to communicate with multiple services that do work in parallel and **needs responses to be aggregated afterwords**
  - Has **Invalid Message channel** (handles message that makes no sense), **Dead Letter Channel** (handles message that cannot be delivered), **Data Type Channel** (separate channel for each type of data e.g. XML, JSON, byte array etc.)

6.2.3 Message Routing

- Consumes messages from 1 channel and reinserts them into different channels based on set of conditions
- **Simple router** which route messages from 1 inbound channel to ≥ 1 outbound channels. **Composed Routers** which combines multiple simple router to create complex message flows
- **Content-Based**: Examines message content and routes → has knowledge of all possible recipients and their capabilities. **Message filter is a special kind of content-based router** that discards messages based on content. **Decouples receiver and sender**
- **Context-based**: Decides destination based on context, e.g. load-balancing, fail-over
- **Message Splitter**: Splits a single message into multiple messages, **not a router**
- **Message Aggregator**: Aggregates correlated messages into a single message
- **Scatter Gather**: Broadcast to multiple participants and aggregates replies into 1 message

6.2.4 Message Transformation

- Transforms application layer data structures/types/representation e.g. ASCII to Unicode, TCP/IP to Sockets etc. so that systems using different data formats can communicate
- Uses a **Message Translator** which is responsible for the conversion between file formats → follows a **Channel Adapter Pattern**
- Helps to decouple applications from each other as they do not need to know about each other's data formats
- Could also use a **Canonical Data Model** which provides an additional layer of indirection → all data will be first converted to canonical data format, sent, then translated back to its own data format at receiver end
- Can contain a **Message Endpoint** which is an interface between an application and a messaging system → should be an isolated piece of code from the rest of the application
  - Endpoint can be used to either send/receive message but **1 instance does not do both**
  - Endpoint is channel specific → multiple endpoints to interface with multiple channels
  - **Polling Consumer**: controls when it consumes messages, **proactively** reads messages. **Event-Driven Consumer**: message delivery is an event that trigger receiver, **reactively** process messages

7 Design Patterns

Design pattern is a *solution* (pattern itself, general design) to a *recurring problem* (goal and constraints to be achieved, aka **forces**) in a *context* (situation in which pattern applies). A pattern consists of a name, classification, intent (what the pattern does), participants (responsibilities of classes/objects involved), Implementation/Sample Code.

7.1 GoF Patterns Category

- **Creational Patterns** - help designers handle issues with creation of objects, **encapsulate/hide details** about actual creation
- **Structural Pattern** - provide structure to relationship between objects, allows for **flexibility** in interconnecting modules
- **Behavioral Patterns** - help define how objects talk with each other, **increases communication flexibility**, **simplify flow** and **make communications more understandable**

7.1.1 Principles of GoF

- **Program to interface/abstract class** - want to depend on interfaces only so that we are decoupled from implementation → implementation can vary without affecting application, **healthy dependency relationship**
- Favor Composition/aggregation over inheritance (**has-a over is-a**) - Inheritance causes tight coupling between base class and subclass → use composition to reduce coupling
- **Encapsulate what 'varies'** - separate code that varies from code that does not vary (static code) so that we can alter the part that varies without affecting the static code

7.2 Creational Pattern

7.2.1 Builder

- useful for objects with many possible constructor parameters
- **Intent**: Separate the construction of a complex object from its representation so that the same construction process can create different representations → client separated from object creation process
- Enforces **step-by-step process** which remains the same but end products can be different
- **Participants**: **Product** (class that represent product to be created), **Director** (class that dirrects `Builder` to perform steps), **Builder** (interface to build parts), ConcreteBuilder (class that implements Builder)

7.2.2 Prototype

- used in cases when a specific resource is expensive to create (e.g. need to call outside API)
- **Intent**: create an object by cloning another as necessary
- Client does not instantiate a product directly, calls the `clone()` method of the prototype
- **Participants**: **Client**, **Prototype** (interface or abstract class that defined contract of classes that permits cloning), **ConcretePrototype** (class that provides cloning operations)

7.3 Structural Patterns

7.3.1 Adapter (aka Wrapper)

- allows interface of existing class to be used from another interface, follows **SRP** and **dependency inversion principles**
- **Intent**: convert interface into something that is compatible with what the client expects
- **Participants**: **Client**, **Target** (existing interface client communicates with), **Adaptee** (new incompatible interface), **Adapter** (class that adapts adaptee to target)

7.3.2 Facade

- **Intent**: provide unified interface to a set of interfaces in subsystem to make subsystem easier to use
- **internal subsystems still talk to each other** but client just talks with Facade
- **Participants**: **Client**, **Facade** (delegates client requests to appropriate subsystem), **Subsystems** (used by facade but not the other way round, subsystems no reference to facade)

7.4 Behavioral Pattern

7.4.1 Observer Pattern

- **Intent**: Let objects observe behavior of other objects so they stay in sync, abstraction has 2 aspect (one dependent on another), change to 1 aspect requires notifying dependents
- **Participants**: **Subject** (interface/abstract class defining operations for attaching/removing observers), **ConcreteSubject** (maintain state of object and notify observers when change detected), **Observer** (interface/abstract class, defines operations to be used to notify this object), **ConcreteObserver**
- Typically used in GUI apps where different views can be rendered for same subject
- **Pull-Model**: register with subject → subject change states → notify observers → observer pull changed data
- **Push-Model**: mechanism similar to Pull-Model except subject will just push a snapshot of its state to observers instead of notifying them when state changes i.e. observers don't have to manually pull data
- Follows **OCP**

7.4.2 Mediator

- **Intent**: Mediator encapsulates how a set of objects communicates which reduces coupling between objects. Objects delegate routing data, messages, request through mediator
- Subsystem don't talk to each other but instead talk to mediator (different from Facade)
- **Participants**: **MediatorBase** (abstract class that defines communication with Colleague objects), **ConcreteMediator** (implements Mediator, holds reference to Colleagues it serves), **ColleagueBase**(abstract class, holds reference to mediator), **ConcreteColleague**

7.4.3 Memento

- **Intent**: w/o violating encapsulation, allow client to capture an object's state, and restore
- Allows an `Originator` that can create or restore from a `Memento`, which captures the former's state. A `Caretaker` can help to safekeep `Mementos` and clean them up when the `Originator` is deallocated.

7.4.4 State

- **Intent**: allow an object to alter its behavior when internal state changes
- Prevent massive switch cases by defining a class for each `State`, and delegating `Context` behavior to those `ConcreteState` (using polymorphism). Difference with Strategy is that `ConcreteStates` are aware of each other and transition the `Context`'s `State` accordingly
- Follows **open-close principle**

7.4.5 Strategy

- **Intent**: represent a behavior that parameterizes an algorithm for behavior or performance
- Encapsulate algorithms into `Strategy`, then have the `Context` use the correct `ConcreteStrategy`. Follows **open-close principle**

7.5 Other Patterns

7.5.1 Data Transfer Object

- Bundle all data items that might be needed into a single DTO used for querying or updating attributes together. Reduce multiple remote calls into a single call which in turns reduces network traffic. DTO is generally **immutable and read-only**

8 Software Design Quality

Functionality The degree of completeness, correctness, and appropriateness of the software's functionality.	Reliability The degree of consistency, availability, fault-tolerance, and recoverability from errors of the software system.	Compatibility The degree of the software's ability to coexist and be interoperable with the existing software and hardware.
Performance (efficiency) The level of the software's response times, throughput, and utilization of hardware and software resources.	Scalability The ability of a software to handle the increased load without decreasing its performance.	Security The degree of confidentiality, integrity, accountability, and authenticity of the software.
Maintainability The degree of modularity, reusability, analyzability, modifiability, and testability of the application.	Usability The level of the software's learnability, operability, accessibility, error-protection, aesthetics.	Portability The software's adaptability, installability, and replaceability of the system.

- **Quality Perspective** refers to how good is software design in terms of product and process perspectives
- **Erosion**: overall deterioration of the engineering quality of a software system that could lead to technical debt
- Symptoms of erosion: **Rigidity** (every change forces many other changes, system hard to change), **Immobility** (hard to modularize system), **Fragility** (system break in conceptually unrelated places), **Viscosity** (doing things correctly is harder), **Opacity** (code is hard to read and understand)
- **Quality Assurance** follows preventive approach and ensures quality in SE **processes**. **Quality Control** is post-development focused, ensuring quality in the review/testing phase