CS603 SOFTWARE ARCHITECTURE & DESIGN
SHORT NOTES FOR FINAL-TERM
TOPIC 77 TO 227
PREPARED BY ALMAS AFZAL

Topic_77
Software Design & Architecture
(Modeling State Machine)

PREPARED BY ALMAS AFZAL

State Machine Modeling:

State Machine Modeling is a behavioral modeling technique used in Software Design & Architecture to represent the states an object can be in and how it transitions between these states in response to events.
This approach is especially useful for modeling reactive systems like user interfaces, devices, protocols, or game logic.

FOR MCQ:

State_____ A specific condition or situation of an object during its life cycle.
Event_____An occurrence that may trigger a transition (e.g., user input, timeout).
Transition_____A movement from one state to another triggered by an event.
Action_____A behavior that occurs due to an event or transition.
Initial State_____The default starting point of a state machine.
Final State_____The terminating point of the state machine

Why Use State Machines in Software Design?

- ❖ Clarifies complex logic through visual representation.
- ❖ Identifies all possible states and transitions.
- ❖ Simplifies testing and debugging.
- ❖ Helps in managing life-cycle of objects (e.g., orders, users, devices).

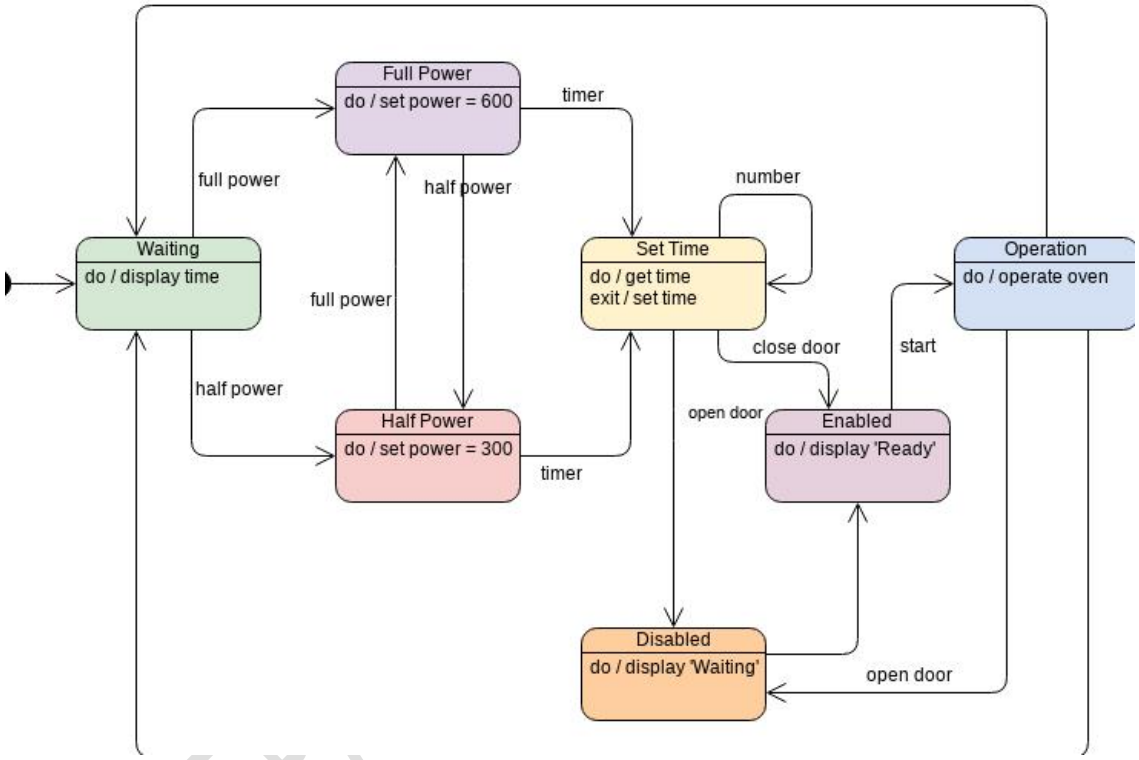
UML State Machine Diagram

A UML State Machine Diagram visually represents the states and transitions of a class or system.

Basic Syntax:

- **Rounded rectangles** represent states.
- **Arrows** represent transitions.
- **Labels on arrows** show the triggering event and action.

Sample:



Modeling Steps:

1. **Identify the object or process** to be modeled.
2. **List all possible states** the object can be in.
3. **Define events** that cause transitions between states.
4. **Draw the state diagram** using standard UML notations.
5. **Add actions or activities** that occur during transitions or inside states.

Topic_78 & 79
Software Design & Architecture
(State Design Pattern)

Definition:

The State Design Pattern is a behavioral design pattern that allows an object to alter its behavior when its internal state changes. The object will appear to change its class.

When to Use:

Use the State Pattern when:

- An object must change its behavior based on internal state.
- You want to avoid large if-else or switch statements scattered across the code.
- You have many states with distinct behaviors, and adding new states should be easy.

Advantages:

- Simplifies complex conditional logic.
- Makes state transitions explicit and organized.
- Follows Open/Closed Principle (easily add new states).

Disadvantages:

- Increases the number of classes.
- Can be overkill for simple state logic.

PREPARED BY ALMAS AFZAL

Topic_80
Software Design & Architecture
(Comparison of Strategy & State Design Pattern)

Strategy_____Defines a family of algorithms, encapsulates each one, and makes them interchangeable.
State_____Allows an object to alter its behavior when its internal state changes.

- **Strategy Pattern** is about **choosing algorithms externally.**
- **State Pattern** is about **changing behavior based on internal state.**
- Both use the same structure (Context + multiple interchangeable classes), but differ in purpose and behavior control.

Topic_81
Software Design & Architecture
(Composite Design Pattern)

PREPARED BY ALMAS AFZAL

What is the Composite Design Pattern?

The **Composite Design Pattern** is a structural design pattern used to represent **part-whole hierarchies**. It allows **individual objects** and **groups of objects (composites)** to be treated **uniformly**.

Benefits of Composite Pattern

- ✓ **Simplifies code:** clients don't need to distinguish between single or group elements
- ✓ **Supports complex structures** (tree hierarchies)
- ✓ **Follows Open/Closed Principle:** easy to add new types of components
- ✓ **Promotes recursion** for handling structures

Drawbacks

- ✓ Can make code more complex when the tree structure isn't necessary
- ✓ Might lead to a large number of small classes

Important MCQs

- Composite Pattern is a **structural pattern**
- Represents **part-whole hierarchy**
- Treats **individual & composite objects uniformly**

- Supports **recursion and tree-like structures**
- **Downside:** May cause extra complexity and many small classes

---

Topic_82
Software Design & Architecture
(Observer Design Pattern)

PREPARED BY ALMAS AFZAL

What is the Observer Design Pattern?

The **Observer Pattern** defines a **one-to-many dependency** between objects so that when **one object (Subject)** changes state, **all its dependents (Observers)** are automatically **notified and updated.**

Purpose:

To establish automatic communication between objects. One object (the Subject) maintains a list of observers and notifies them of any state changes, usually by calling one of their methods.

Scenario	Subject	Observers
YouTube channel	The channel	Subscribers
Weather station	Weather data source	Weather display screens, apps
Stock market	Stock ticker	Trading apps, investors
Social media	User profile	Followers

Important MCQs

- **Observer Pattern is a behavioral design pattern**
- It defines a **one-to-many relationship**
- **Subject notifies all Observers when state changes**
- Used where **automatic communicatio**n is needed
- **Real life examples: YouTube, Weather, Social Media**

Topic_83 & 84
Software Design & Architecture
(Refactoring Introduction)

What is Refactoring?

Refactoring is the process of restructuring existing code without changing its external behavior.
The goal is to improve the internal structure of the code, making it cleaner, more efficient, and easier to maintain, without affecting what it does.

"Refactoring is the art of improving the design of existing code without changing its observable behavior." **— Martin Fowler**

Purpose of Refactoring

Objective	Explanation
Improve code readability	Make code easier to understand for developers
Reduce complexity	Eliminate redundant or duplicate logic
Increase maintainability	Easier to modify, debug, and extend in the future
Enhance performance (sometimes)	Optimize code structure (not always for speed)
Support clean architecture	Align code with design principles (SOLID, DRY, etc.)

PREPARED BY ALMAS AFZAL

When to Refactor?

- Code is messy or hard to understand
- Adding a new feature is difficult due to tangled logic
- There is duplicate code across functions or files
- You see long methods, deep nesting, or tight coupling
- After passing all tests and confirming behavior is correct

Refactoring vs Rewriting

Refactoring	Rewriting
Keeps original behavior	May change behavior
Done gradually	Done all at once
Safer and testable	Riskier if not well-planned
Improves existing code	Replaces old code entirely

Important MCQs

- **Refactoring** = Code improvement without behavior change
- **Martin Fowler = Refactoring ke legend**
- **Goal** = Readability, maintainability, less complexity
- **Refactoring ≠ Rewriting**
- **Refactoring = Safe & gradual**
- **Rewriting = Risky & big changes**

PREPARED BY ABBAS ALI ZAL

Topic_85
Refactoring vs Rewriting And Optimization

These three activities—Refactoring, Rewriting, and Optimization—are often confused in software development. While all aim to improve software, they differ significantly in purpose, risk, and impact.

Difference

Refactoring

- ✓ Improves the internal structure of the code.
- ✓ Does not change what the code does.
- ✓ Used to make code cleaner, easier to read and maintain.

Rewriting

- ✓ Means starting from scratch.
- ✓ Old code is discarded, and a new version is built.
- ✓ Used when the old system is too messy or outdated to fix.

Optimization

- ✓ Makes the code run faster or use fewer resources.
- ✓ Focuses on performance improvements.
- ✓ Used when the system works but is slow or inefficient.

Important MCQs:

- ❖ **Refactoring**: Improve structure without changing behavior
- ❖ **Rewriting**: Start from scratch, discard old code
- ❖ **Optimization**: Improve performance, same logic
- ❖ **Refactoring** is safest, **Rewriting** is riskiest

❖ **Optimization** can sacrifice readability for speed

Topic_86
Software Design & Architecture
(Making Refactoring Safe)

What Is Safe Refactoring?

Safe refactoring means changing the internal structure of code without introducing bugs or altering its external behavior.
The goal is to improve the quality of the code while ensuring the system still works exactly the same.

Why Make Refactoring Safe?

- ❖ Prevents breaking existing features
- ❖ Protects system stability
- ❖ Builds confidence in making code improvements
- ❖ Ensures future maintainability without risks

Best Practices Checklist

- ✓ Write tests before refactoring
- ✓ Use a separate Git branch
- ✓ Commit frequently
- ✓ Test after every small change
- ✓ Document complex logic for reviewers
- ✓ Avoid changing too much at once

What to Avoid During Refactoring

- Making large changes without testing
- Changing code you don't understand
- Removing or rewriting logic without backups
- Refactoring without version control or rollback options

Topic_87
Software Design & Architecture
(Code smells___ Bad smells in Code)

MASAFZAL

What is a Code Smell?

A Code Smell is a warning sign in the source code that indicates a possible problem in design, structure, or implementation.
It does not necessarily mean the code is broken, but it suggests that the code might need refactoring to improve readability, maintainability, or flexibility.

"A code smell is a surface indication that usually corresponds to a deeper problem in the system."— **Martin Fowler**

Why Identify Code Smells?

- Helps detect poor design decisions
- Makes code easier to maintain and test
- Encourages clean, modular, and reusable code
- Prevents future bugs and technical debt

Benefits of Removing Code Smells

- ✓ Improves code readability
- ✓ Makes code easier to test and debug
- ✓ Enhances collaboration and team productivity
- ✓ Reduces future maintenance cost

Important MCQ:

- ❖ **22 Bad smells Listed in Fowler's Book.**
- ❖ **Code smell ≠ bug**
- ❖ **Code smell = potential refactoring needed**
- ❖ **Removing smells = clean, efficient systems**

Topic_88
Software Design & Architecture
(Code smells___ Bloaters)

CODED BY ALMAS AFZAL

What are Bloaters?

Bloaters are a category of **code smells** where code becomes **too big, too long, or overloaded** with **responsibilities.**
These are sections of code that have grown over time and become **hard to understand, modify, or maintain.**

Bloaters "inflate" your code unnecessarily — they don't break the code, but they reduce clarity and flexibility.

Types of Bloaters

1. Long Method

- ✓ A function grows too long (50+ lines), handling multiple concerns.
- ✓ Problem: Hard to read, reuse, or test.

✓ Refactor with: Extract Method, Replace Temp with Query

2. Large Class

✓ A class has too many responsibilities (e.g., handles UI, data access, and business logic).

✓ Problem: Violates Single Responsibility Principle.

✓ Refactor with: Extract Class, Move Method

3. Primitive Obsession

✓ Overuse of primitives like string, int, etc., instead of defining objects.

✓ Example: Using three int variables for a date instead of a Date object.

✓ Refactor with: Replace Primitive with Object, Encapsulate Field

4. Long Parameter List

✓ Methods take too many arguments (more than 3–4).

✓ Problem: Confusing, error-prone, and difficult to maintain.

✓ Refactor with: Introduce Parameter Object, Preserve Whole Object

5. Data Clumps

✓ Variables that often appear together in multiple places.

✓ Example: (name, address, phone) in multiple methods.

✓ Refactor with: Extract Class or Introduce Parameter Object

PREPARED BY ALMAS AFZAL

Topic_89
Code Smells
(Object_Orientation Abusers)

What Are Object-Orientation Abusers?

Object-Orientation Abusers are a category of code smells that violate core object-oriented programming (OOP) principles, such as:

- **Encapsulation**

- **Inheritance**
- **Polymorphism**
- **Abstraction**

These smells usually occur when **procedural code is written inside an object-oriented system**, or when OOP features are **used incorrectly or excessively.**

Common Object-Orientation Abuser Smells

Smell Name	Description
Switch Statements	Using many switch/if statements instead of polymorphism
Temporary Field	A field that is only used in certain conditions or methods
Refused Bequest	A subclass that doesn't use or ignores inherited functionality
Alternative Classes with Different Interfaces	Classes that do the same thing but use different method names
Incomplete Library Class	A library class that lacks some functionality, causing hacks in your code

PREPARED BY ALMAS AFZAL

Topic_90
Software Design & Architecture
(Code smells___ Change Preventers)

What Are Change Preventers?

Change Preventers are a group of code smells that make it **difficult to modify or extend a system.**
They **discourage or block change** because updating one area of the code requires changes in **many other places**, often breaking existing functionality.

These smells violate principles like:

- Single Responsibility Principle (SRP)
- Open/Closed Principle (OCP)
- Separation of Concerns

Types of Change Preventers

- ✓ Divergent Change
- ✓ Shotgun Surgery
- ✓ Parallel Inheritance Hierarchies

Topic_91
Software Design & Architecture
(Code smells___ Dispensables)

What Are Dispensables?

Dispensables are parts of the code that **add no real value** and can be **safely removed** to make the code **cleaner, simpler, and easier to maintain.**
They include redundant, unused, or overly complicated code elements.

If removing something doesn't break the code and improves clarity — it's a dispensable.

Common Dispensable Code Smells

1. Duplicate Code
2. Lazy Class
3. Data Class
4. Dead Class
5. Speculative Generality

MCQs

- **Dispensables** = Code that can be safely removed
- **Improve**: Readability, simplicity, maintainability
- **Types**: Duplicate Code, Lazy Class, Data Class, Dead Code, Speculative Generality

PREPARED BY ALMAS AFZAL

- **Lazy Class** = Not doing enough work
- **Data Class** = Only has fields/getters/setters
- **Speculative Generality** = Future-proof code that's never used

---

Topic_92
Software Design & Architecture
(Code smells___ Couplers)

What Are Couplers?

Couplers are code smells that indicate **excessive or improper dependency between classes or modules.**
They **violate the principle of low coupling**, which is key to building modular, flexible, and maintainable software systems.

*A "coupler" smell means **one part of the system knows too much about another.***

Common Coupler Code Smells

- ✓ Feature Envy
- ✓ Inappropriate Intimacy
- ✓ Middle Man

Topic_93
Software Design & Architecture
(Refactoring ___ The Catalog)

What is the Refactoring Catalog?

The **Refactoring Catalog** is a collection of **standard, proven techniques** for improving the internal structure of existing code **without changing its external behavior.**
These techniques are used to **clean up code smells**, **improve readability**, and **make future development easier.**

Originally popularized by **Martin Fowler** in his book "Refactoring: Improving the Design of Existing Code".

PREPARED BY ALMAS AFZAL

Why Use the Refactoring Catalog?

- Standardizes improvement process
- Makes code easier to maintain
- Reduces bugs
- Enables continuous improvement

Duplicated Code ____ A bad Smell

- Extract Method
- Extract Class
- Pull up Method
- Form Tempate Method

Topic_94
Software Design & Architecture
(Refactoring ___ Extract Method)

PREPARED BY ALMAS AFZAL

What is "Extract Method"?

Extract Method is a refactoring technique used to improve code readability, reusability, and maintainability by taking a piece of code from an existing method and putting it into a new, well-named method.

Purpose: To break down long or complex methods into smaller, focused units.

When to Use Extract Method

- A method is too long
- A section of code is repeated in multiple places
- A block of code does a separate task
- You want to isolate functionality for reuse or testing
- Code becomes difficult to read or understand

How to Apply Extract Method

✧ **Identify** the code block you want to extract

✧ **Select** the lines and create a new method

✧ **Give a clear name** that describes the purpose

✧ **Replace** the original code with a call to the new method

✧ **Test** to ensure behavior hasn't changed

Topic_95
Software Design & Architecture
(Refactoring ___ Replace Temp with Query)

PREPARED BY ALMAS AFZAL

What is "Replace Temp with Query"?

Replace Temp with Query is a refactoring technique used to eliminate temporary variables (temps) by replacing them with **method calls (queries)** that calculate the value **directly**.

Goal: Make code **more flexible, reusable, and easier to maintain** by avoiding unnecessary temporary storage.

Use this when:

✧ A temporary variable holds the result of a simple expression

✧ The variable is only used in a limited scope

✧ The same logic could be placed in a method that can be reused

✧ You want to improve encapsulation and code clarity

When Not to Use

✧ If the value changes multiple times (then use temp variable)

✧ If performance is critical and the expression is costly to compute repeatedly

✧ If the value is used across multiple scopes in different contexts

---

Topic_96
Software Design & Architecture
(Refactoring ___ Move Method)

What is “Move Method”?

Move Method is a refactoring technique used when a method **uses more data or functions from another class** than from its own.
To reduce coupling and improve cohesion, the method should be **moved to the class it uses most.**

Goal: Improve design by placing behavior where it logically belongs.

Use this when:

- ✓ A method accesses more data/methods from another class
- ✓ The method doesn’t really belong in the current class
- ✓ You want to simplify dependencies and follow Single Responsibility Principle (SRP)
- ✓ You notice Feature Envy (a code smell)

When Not to Use

- ✓ If the method strongly depends on multiple classes, splitting logic might be better
- ✓ If moving the method creates circular dependencies or tight coupling
- ✓ If moving the method breaks access to private/protected members

PREPARED BY ALMAS AFZAL

Topic_97
Software Design & Architecture
(Refactoring ___Replace Conditional with Polymorphism)

Replace Conditional with Polymorphism is a refactoring technique used to eliminate **complex if-else or switch statements** that select behavior based on an object's type or value, and instead use **polymorphism (sub-classing and method overriding).**

It allows behavior to be **defined in** sub-classes, making the code **cleaner, more modular, and easier to maintain.**

Benefits

- ✓ Removes duplication
- ✓ Improves readability
- ✓ Easier to extend
- ✓ Aligns with OOP principles

Topic_97-100
Software Design & Architecture
(Refactoring ___Introduce Null Object)

PREPARED BY ALMAS AFZAL

What Is "Introduce Null Object"?

Introduce Null Object is a refactoring technique that involves replacing null values (or empty references) with a **special object** that represents "do nothing" or "default behavior".

Instead of checking for null with if statements, we use a **Null Object** that behaves in a safe and predictable way.

Use this technique when:

- ❖ Your code contains many null checks
- ❖ A null reference is treated as a special case
- ❖ You want to avoid conditional logic and improve code safety
- ❖ You follow Polymorphism and want all objects to follow a common interface

---

Topic_101 to 107
(Skipped)

Topic_108
Software Design & Architecture
(Refactoring ___Managing Refactoring)

What Is "Managing Refactoring"?

Managing Refactoring means applying refactoring in a **planned, safe, and controlled way** during software development or maintenance.
It involves choosing **when, where, and how** to refactor, ensuring that improvements are made **without breaking functionality** or slowing progress.

*Refactoring should not be random. It should be **systematic, test-driven, and aligned with project goals**.*

Why Managing Refactoring Matters.?

- ❖ Prevents unintended side effects
- ❖ Saves time in the long run
- ❖ Aligns with business goals
- ❖ Supports continuous improvement

PREPARED BY ALMAS AFZAL

Topic_109
Software Design & Architecture
(Problems with Refactoring)

What Are the Problems with Refactoring?

While **refactoring** improves code quality, it also comes with **challenges, risks, and limitations**—especially if not managed carefully.

Refactoring **changes internal code structure**, so without proper controls, it can **introduce bugs, waste effort**, or even **slow down the team.**

Common Problems with Refactoring

1. Lack of Automated Testing
2. Unclear Objectives
3. Tight Deadlines or Time Pressure
4. Mixing Refactoring with Feature Development
5. Resistance from Team or Management

Topic_110
Software Design & Architecture
(MVC __ Challenges of Interactive Application)

PREPARED BY ALMAS AFZAL

What is MVC (Model–View–Controller)?

MVC is a software architectural pattern used for developing interactive applications by separating concerns:

Component	Responsibility
Model	Handles data and business logic
View	Displays the data (UI)
Controller	Manages user input and updates model/view

This separation helps organize code, especially in web and desktop applications.

What Are Interactive Applications?

- Interactive applications are systems where users:
- Continuously interact with the interface
- Expect real-time feedback
- Require smooth, responsive experiences

Examples: social media apps, dashboards, online editors, games, booking systems.

Topic_111

MVC __ Example Part I
(Requirements)

This is the **first part of a practical example** of how to apply the **MVC (Model–View–Controller)** architectural pattern.
In this part, we focus on defining the **requirements** of an interactive application before designing it using MVC.

Objective of This Phase

- Understand **what the application will do**
- Identify **user interactions**
- Clarify the **functional and non-functional requirements**
- Prepare for proper **separation of concerns** using MVC

Topic_112
MVC __ Example Part II
(Monolithic Console Implementation)

Purpose of This Step

Before applying MVC, we first demonstrate how the entire application logic is written in one class or file – a monolithic approach.
This helps us understand why MVC is needed and how it improves design.

In this version:

- Everything is handled in a **single class**: input, logic, data
- There is **no separation of concerns** (e.g., between UI and logic)
- It works, but is **difficult to scale or maintain**

What's Included in the Monolithic Version?

1. User input/output
2. Business Logic
3. Data storage (in-memory)

PREPARED BY ALMAS AFZAL

4. User & admin actions

Topic_113
MVC __ Example Part III
(Monolithic Mixed UI Implementation)

In this step, we build a console-based application that mixes:

- **User Interface (UI)**
- **Business Logic**
- **Data Management**

in the same functions or class, before splitting them into MVC layers.

Conclusion:

- ✓ This stage shows **why we need MVC**
- ✓ Mixing UI and logic in one place is **not scalable**
- ✓ Helps us **prepare for clean separation** in the next step

---

Topic_114 to 118
Software Design & Architecture
(Drawbacks of Monolithic UI)

What is a Monolithic UI?
A monolithic UI combines the user interface (UI), business logic, and data access all in one class, file, or module — without proper separation.
This is often seen in early-stage or small console applications before introducing patterns like MVC.

Key Drawbacks of Monolithic UI Design

- ✧ Poor separation of concerns
- ✧ Difficult to maintain
- ✧ Hard to test

PREPARED BY ALMAS AFZAL

- ✧ No reusability
- ✧ High risk of bugs
- ✧ Low scalability
- ✧ Difficult debugging
- ✧ Hard to manage user roles
- ✧ Violates design principles
- ✧ Complicated code structure
- ✧ Inflexible for changes
- ✧ Mixed logic and presentation
- ✧ Difficult team collaboration

Why You Should Avoid Monolithic UI

- ✓ It **slows down development** as the system grows
- ✓ Makes collaboration between team members harder
- ✓ Prevents you from adapting to **modern architectures** like MVC, MVP, or MVVM
- ✓ Blocks you from applying **unit testing, CI/CD**, and modular deployments

PREPARED BY ALMAS AFZAL

Topic_119
Software Design & Architecture
(What is Software Architecture)

Software Architecture refers to the high-level structure of a software system.
It defines how the components of the system are organized, how they interact, and how they meet technical and business requirements.

- ❖ **Structure** of components and their relationships
- ❖ **Design decisions** that impact performance, scalability, and maintainability

- **Guidelines** for implementation, deployment, and evolution of the system
- **Separation of concerns** to manage complexity

Main Elements of Software Architecture

Element	Description
Components	Functional parts/modules of the system
Connectors	Communication between components (e.g., APIs, events)
Configurations	Arrangement/interaction of components and connectors

Why Software Architecture Is Important.?

1. Acts as a **blueprint** for the system
2. Enables better **planning and decision-making**
3. Helps ensure **non-functional requirements** (e.g., performance, security)
4. Affects **team collaboration**, scalability, and future changes

PREPARED BY ALMAS AFZAL

Topic_120
Software Design & Architecture
(Categories of Structures in Architectural Design)

What Are Architectural Structures?

In software architecture, structures define how a system is organized, both in terms of code, components, and runtime behavior.
Each structure provides a different perspective to understand, design, and analyze a system.

Main Categories of Architectural Structures

Here are the three primary categories of architectural structures:

Structure Type	Focus Area	Examples

Structure Type	Focus Area	Examples
Module Structures	Code organization (static)	Layers, modules, packages
Component-and-Connector	Runtime interaction (dynamic)	Client-server, events, REST APIs
Allocation Structures	Mapping software to environment	Deployment diagrams, cloud instances

---

Topic_121
Software Design & Architecture
(Static Software Structure)

What is Static Software Structure?

The static software structure refers to the **organization of software components in code** — how the system is arranged at rest, without execution.

It **represents** the **logical structure** of the software, such as modules, packages, classes, and their relationships.

Key Characteristics

1) **Focuses** on the **static aspects** of the system

2) Shows how **source code elements** are organized

3) **Does not** include runtime behavior

4) Supports **modularity, abstraction, and encapsulation**

Tools Used to Model Static Structure

- ❖ UML Class Diagrams
- ❖ Package Diagrams
- ❖ Dependency Graphs
- ❖ IDE Explorers (e.g., Eclipse, IntelliJ structure views)

PREPARED BY ALMAS AFZAL

Topic_122
Components-and-Connectors and

Allocation Structures

In software architecture, different structure types give different views of a system. Two key dynamic and physical views are:

1) Component-and-Connector **(C&C)** Structures

2) Allocation Structures

Component-and-Connector (C&C) Structures

Definition:

Shows the system's runtime structure — how components interact during execution through connectors.

Examples:

- ✓ Client–server system
- ✓ Microservices calling each other via HTTP
- ✓ Event-driven systems with message queues

Allocation Structures

Definition:

Shows how software elements are assigned to physical resources (hardware, files, teams, etc.)

Used In:

- ✓ Deployment diagrams
- ✓ DevOps and system operations
- ✓ Cloud and container-based systems

PREPARED BY ALMAS AFZAL

Topic_123 & 124
Software Design and Architecture
(Quality Attributes)

What Are Quality Attributes?

Quality Attributes are **non-functional requirements** that describe **how well** a software system performs, behaves, and adapts — beyond just what it does.

They help evaluate **system performance, usability, reliability, and maintainability**, ensuring the software meets real-world expectations.

Common Quality Attributes in Software Architecture

- ❖ Performance
- ❖ Scalability
- ❖ Availability
- ❖ Reliability
- ❖ Security
- ❖ Maintainability
- ❖ Usability
- ❖ Modifiable
- ❖ Test-ability
- ❖ Portability
- ❖ Interoperability
- ❖ Flexibility
- ❖ Re-usability
- ❖ Efficiency
- ❖ Fault Tolerance

What Should NOT Be Included in Quality Attributes.?

- ✧ Functional requirements
- ✧ Programming languages or tools used
- ✧ UI design choices (e.g., colors, fonts)

PREPARED BY ALMAS AFZAL

- ✧ Project deadlines or budget
- ✧ Development tasks (e.g., writing tests, code refactoring)

Topic_125
Software Design and Architecture
(Difference b/w Architecture & Representation of Architecture)

PREPARED BY AMAS AFZAL

Aspect	Architecture	Representation of Architecture
Definition	Actual structure and design of the software system	How the architecture is expressed or visualized
Form	Conceptual and abstract	Diagrams, documents, or models
Purpose	Defines how the system is organized and functions	Communicates architecture to stakeholders
Exists As	Internal design decisions and system structure	External artifacts like UML, flowcharts, architecture views
Changes With System?	Evolves as the system evolves	Must be updated to stay in sync with architecture
Used By	Developers, architects (to build the system)	Teams, managers, stakeholders (to understand the system)

Topic_126
Software Design and Architecture
(Difference b/w Software, system & Enterprise Architecture)

Aspect	Software Architecture	System Architecture	Enterprise Architecture

Aspect	Software Architecture	System Architecture	Enterprise Architecture
Scope	Focuses on software systems or applications	Focuses on the entire system (hardware + software + network)	Focuses on the entire organization and its IT alignment
Level	Low to mid-level	Mid-level	High-level (strategic)
Concerned With	Code structure, components, interactions	System components, platforms, interfaces	Business goals, IT strategy, processes, and technologies
Examples	Web app architecture, micro-services	Embedded systems, IoT setup	Organization-wide ERP, cloud migration strategy
Stakeholders	Developers, software architects	System engineers, infrastructure architects	CIOs, enterprise architects, business managers
Key Output	Component diagrams, design patterns	Network diagrams, deployment plans	Business capability models, architecture

---

PREPARED BY ALMAS AKAL

Topic_127
Software Design and Architecture
(Architectural Views)

What Are Architectural Views?

Architectural views are different **perspectives or representations** of the system's architecture, each highlighting specific concerns for specific stakeholders.

They help to **organize, communicate, and analyze** different aspects of a complex software system.

Common Types of Architectural Views

1. Logical View
2. Development View
3. Process View
4. Physical View
5. Scenarios (Use Cases)

4+1 View Model (by Philippe Kruchten)

A popular framework that organizes architectural views into:

✓ **4 Views:** Logical, Development, Process, Physical

✓ **+1:** Scenarios (Use cases) – validate the architecture

Topic_128
Software Design and Architecture
(4+1 View Model of Software Architecture)

What is the 4+1 View Model?

The 4+1 View Model of software architecture, proposed by **Philippe Kruchten**, organizes software architecture into **five interrelated views** to address the concerns of **different stakeholders**.

It helps describe, document, and analyze complex software systems from multiple perspectives.

1) Logical View
2) Development View
3) Process View
4) Physical View
5) Scenarios (Use Cases)

Topic_129
Software Design and Architecture
(Module Structures)

What Are Module Structures?

Module Structures describe the **static organization** of software into modules or components.
They represent how the **code is organized** at a high level — such as files, packages, libraries, or subsystems — and how these relate to each other.

Think of module structures as the **"blueprint" of code organization** that supports modularity, reuse, and separation of concerns.

Common Types of Module Structures

✓ Decomposition Structure
✓ Uses Structure
✓ Layered Structure
✓ Class Structure

PREPARED BY ALMAS AFZAL

Purpose of Module Structures

- Improve **code organization and maintainability**
- Promote **modular and scalable design**
- Help manage **complexity** by separating responsibilities
- Enable **team-based developmen**t by assigning modules to developers

---

Topic_130
Software Design and Architecture
(Component & Connector Structures)

What Are C&C Structures?

Component & Connector (C&C) Structures describe the system's **runtime architecture** — how components interact through connectors while the system is executing.

It focuses on the **dynamic aspects** of the system: control flow, communication, and coordination between components.

Purpose of C&C Structures

Model **runtime interactions** of a system

Help in analyzing **performance, reliability, and scalability**

Useful for understanding **concurrency, distribution, and communication mechanisms**

Topic_131
Software Design and Architecture
(Allocation Structures)

What Are Allocation Structures?

Allocation Structures describe how **software elements (modules, components, processes)** are **mapped to external environments**, such as:

- **Hardware** (machines, servers, devices)
- **File systems** (storage locations, directories)

PREPARED BY ALMAS AFZAL

❖ **Development teams** (who owns or builds what)

It focuses on the physical deployment and operational environment of the software system.

Common Types of Allocation Structures

1. Deployment Allocation
2. Implementation Allocation
3. Work Assignment Allocation

Purpose of Allocation Structures

✓ Ensure proper **deployment planning and operations**

✓ Help in **resource management** and **team coordination**

✓ Support **performance, scalability**, and **reliability** goals

---

Topic_132
Software Design and Architecture
(Structures & Quality Attributes)

Software structures (Module, Component-and-Connector, and Allocation) directly influence **quality attributes** like performance, modifiability, scalability, etc.

Choosing the right architectural structure is essential to **achieve and support** specific **quality goals.**

Topic_133
Software Design and Architecture
(Relating Structures to each other)

Software architecture uses **multiple structures** (Module, Component-and-Connector, and Allocation) to describe different views of a system.
These structures are **interrelated**, and understanding their relationship helps in **consistency**, **traceability**, and **system integrity.**

PREPARED BY ALMAS AFZAL

Topic_134
Software Design and Architecture
(Choosing an Appropriate Structure)

Choosing the **right architectural structure** is essential to meet a system's **functional and quality goals**.
Different structures highlight different concerns — so selection should depend on project **requirements, constraints, and priorities.**

Key Factors in Choosing a Structure

- **System Requirements** (functional + non-functional)
- **Quality Attributes** (e.g., performance, scalability, modifiability)
- **Team Size and Skills**
- **Deployment Environment** (cloud, mobile, embedded, etc.)
- **Technology Stack**
- **System Complexity and Size**
- **Re-usability and Maintainability Needs**

PREPARED BY ALMAS AFZAL

Topic_135
Software Design and Architecture
(Architectural Pattern_Overview)

What is an Architectural Pattern?
An **Architectural Pattern** is a **general, reusable solution** to a common problem in software architecture.
It describes the **overall structure and interaction** style of a software system.

Think of it as a **blueprint** for designing systems that ensures **best practices, consistency**, and **scalability**.

Key Characteristics of Architectural Patterns

- ✓ **High-level design guidance**
- ✓ **Solves recurring architectural problems**
- ✓ Focuses on **system-wide structure and behavior**
- ✓ Defines **components**, their **responsibilities**, and **interaction**
- ✓ Independent of specific technologies or languages

Why Use Architectural Patterns?

- ✓ To **ensure standardized and proven solutions**
- ✓ To **improve communication** between architects and developers
- ✓ To help achieve specific **quality attributes (like scalability, modifiability, etc.)**
- ✓ To reduce **risk and complexity** in design decisions

Topic_136
Software Design and Architecture
(What Makes a Good Architecture)

PREPARED BY ALMAS AFZAL

A **good architecture** is one that effectively supports both the **functional requirements** (what the system should do) and the **non-functional requirements** (how well it should perform).

It should be **robust, scale-able, maintainable, and aligned** with business goals and quality expectations.

Characteristics of a Good Architecture

1. Meets Functional Requirements
2. Supports Quality Attributes (e.g., performance, security, scalability)
3. Modular and Reusable
4. Easy to Understand and Communicate

5. Change-tolerant (Flexible & Maintainable)
6. Well-documented and Traceable
7. Cost-effective

Topic_137
Software Design and Architecture
(Process Recommendations)

Process Recommendations refer to the **best practices, steps, and guidelines** followed to ensure that software architecture is designed effectively and evolves safely over time.

The goal is to produce an architecture that is **fit for purpose, adaptable**, and **aligned with business and technical needs.**

Topic_138
Software Design and Architecture
(Product Recommendations)

Product recommendations in software architecture refer to **tools, platforms, and technologies** that assist architects, developers, and teams in **designing, modeling, analyzing, implementing, and managing** software architecture effectively.

Topic_139
Software Design and Architecture
(Architectural Drivers)

What Are Architectural Drivers?
Architectural drivers are the **key influencing factors** that shape the **architecture** of a software system.
They define the **needs, constraints**, and **priorities** that the architecture must satisfy.

These are the **primary inputs** that guide the architect in making structural and design decisions.

PREPARED BY M.S AFZAL

Types of Architectural Drivers

1) Functioanl Requirements
2) Quality Attributes
3) Constraints
4) Business Goals

Topic_140
Software Design and Architecture
(The Architectural Meta-Frame)

PREPARED BY ALMAS AFZAL

What is the Architectural Meta-Frame?

The **Architectural Meta-Frame** is a **conceptual framework** that provides a **structured way to think about and define software architecture.**
It outlines the **core elements** and their **relationships** involved in architecture decision-making.

It acts as a **mental model or checklist** to guide architects through consistent and complete architectural design.

Purpose of the Meta-Frame

- Provides a **systematic approach** to architecture
- Ensures that all **important elements and concerns** are addressed
- Helps in **organizing knowledge, communicating with stakeholders**, and **reusing best practices**

Key Components of the Architectural Meta-Frame

- Architectural Drivers
- Architectural Views & Structures
- Architectural Patterns & Styles
- Design Decisions
- Architectural Documentation

❖ Evaluation & Validation

Topic_141
Software Design and Architecture
The Architectural Meta-Frame
(Quality Attributes___Introduction)

Introduction to Quality Attributes

1. Define **how well** a system performs (not what it does)
2. Also known as **non-functional requirements**
3. Drive **architectural decisions and trade-offs**
4. Essential to system's **usability, maintainability, and performance**
5. Commonly include **performance, security, scalability, availability, modifiability**, etc.

Topic_142
Software Design and Architecture
The Architectural Meta-Frame
(Architecturally Significant Quality Attributes)

What Are Architecturally Significant Quality Attributes?

These are the **non-functional requirements** that have a **major impact on architectural decisions.**
They **shape the structure** and behavior of the system more than others.

Common Architecturally Significant Quality Attributes

- Performance
- Scalability
- Security
- Availability

PREPARED BY ALMAS AFZAL

- Maintainability
- Modifiability
- Reliability
- Usability
- Interoperability

*These attributes typically **conflict**, so architects must make **trade-offs** (e.g., security vs. usability, performance vs. maintainability).*

Topic_143
Software Design and Architecture
The Architectural Meta-Frame
(Guidelines Quality Attributes)

Guidelines for Handling Quality Attributes

These guidelines help ensure that quality attributes are properly addressed in architectural decisions.

1) **Identify quality attributes early** in the architecture process
2) **Prioritize** attributes based on stakeholder needs and project goals
3) **Define measurable scenarios** for each quality attribute (e.g., response time < 2s)
4) Use **quality attribute scenarios** to evaluate architectural choices
5) Understand **trade-offs** (e.g., performance vs. security)
6) Apply **tactics and patterns** to address specific attributes
7) **Document decisions** made to support quality goals
8) Continuously **validate and test** quality attribute fulfillment throughout development

PREPARED BY ALMAS AFZAL

Topic_144

Software Design and Architecture
The Architectural Meta-Frame
(Quality Attributes___Description)

Description of Common Quality Attributes

- Performance
- Scalability
- Availability
- Security
- Modifiability
- Maintainability
- Usability
- Reliability
- Interoperability
- Portability

Topic_145
Software Design and Architecture
The Architectural Meta-Frame
(Requirements & Constraints)

PREPARED BY ALMAS AFZAL

In software architecture, **Requirements and Constraints** are key **architectural drivers** that shape the overall design and decisions.

They help the architect understand:

1. **What the system must do** (requirements)
2. **What limits or conditions must be followed (constraints)**

Requirements

A. Functional Requirements

Define the features and behaviors the system must support
(e.g., user login, payment processing, report generation)

B. Non-Functional Requirements (Quality Attributes)
Define how well the system performs
(e.g., performance, scalability, security)

Constraints
Constraints are restrictions or rules that the architecture must follow.

Types of Constraints:

- ✓ **Technical Constraints** – Must use a specific language, database, or framework
- ✓ **Business Constraints** – Deadline, budget, organizational policies
- ✓ **Regulatory Constraints** – Compliance with laws or standards (e.g., GDPR, HIPAA)
- ✓ **Resource Constraints** – Limited hardware, skilled personnel, or tools
- ✓ **Time Constraints** – Project must be delivered within a fixed time

---

Topic_146
Software Design and Architecture
The Architectural Meta-Frame
(Application Types)

PREPARED BY ALMAS AFZAL

Different types of applications require **different architectural approaches** depending on their goals, users, environments, and complexity.

These application types help architects choose suitable **architectural patterns, technologies, and structures.**

Common Application Types

- ❖ Web Applications
- ❖ Mobile Applications
- ❖ Desktop Applications
- ❖ Enterprise Applications

- Embedded Systems
- Real-Time Systems

Topic_147
Software Design and Architecture
The Architectural Meta-Frame
(Deployment Strategy)

A **deployment strategy** defines **how and where** software is delivered and run in its production environment. It ensures that the system is **available, scalable, reliable**, and **maintainable** after development.

Common Deployment Strategies

- Single-Server Deployment
- Client–Server Deployment
- Multi-Tier Deployment
- Cloud-Based Deployment
- Microservices Deployment
- Container-Based Deployment (e.g., Docker, Kubernetes)
- On-Premises Deployment
- Hybrid Deployment (Cloud + Local)
- Serverless Deployment
- Edge Deployment (for IoT and latency-sensitive apps)

PREPARED BY ALMAS AIZAL

Topic_148 & 149
Software Design and Architecture
The Architectural Meta-Frame

(Architectural Styles)

Architectural styles are **standardized approaches** to organizing system architecture. Each style defines a set of rules about how components are arranged and interact.

They help in choosing **appropriate structure and behavior** for different types of systems.

Common Architectural Styles

- Layered Style
- Client–Server Style
- Event-Driven Style
- Microservices Style
- Service-Oriented Architecture (SOA)
- Pipe-and-Filter Style
- Monolithic Architecture
- Peer-to-Peer (P2P) Style
- Publish–Subscribe Style
- Broker Style
- Component-Based Architecture
- Server-less Architecture
- Hexagonal Architecture (Ports and Adapters)
- Space-Based Architecture

PREPARED BY ALMAS AFZAL

Topic_150
Software Design and Architecture
The Architectural Frame
(Cross-Cutting Concerns___Part I)

Cross-Cutting Concerns are aspects of a system that **affect multiple layers or components** of an application and are not limited to a single module.
They are often **non-functional requirements** that apply **system-wide.**

These concerns **"cut across"** the typical separation of concerns in a system's architecture.

Examples of Cross-Cutting Concerns

- ✓ Logging
- ✓ Security
- ✓ Authentication & Authorization
- ✓ Caching
- ✓ Exception Handling
- ✓ Performance Monitoring

Topic_151
Software Design and Architecture
The Architectural Frame
(Key Cross-Cutting Concerns___Part II)

*This part focuses on the **most critical cross-cutting concerns** that must be consistently handled across all architectural layers and modules.*

Key Cross-Cutting Concerns

- ✓ Security
- ✓ Logging
- ✓ Error & Exception Handling
- ✓ Configuration Management
- ✓ Monitoring & Observability
- ✓ Transaction Management
- ✓ Caching

PREPARED BY ALMAS AFZAL

- ✓ Data Validation
- ✓ Access Control (Authentication & Authorization)
- ✓ Localization / Internationalization

Topic_152
Software Design and Architecture
The Architectural Meta-Frame
Key Engineering Decisions___Part I
(Authentication & Authorization)

PREPARED BY ALMAS AFZAL

Authentication and Authorization are critical engineering decisions that affect system **security, access control, and user experience.**
These decisions must be carefully planned as part of the architecture, especially in distributed, web, and cloud-based systems.

Key Considerations

Authentication (Who are you?)

- ❖ Credential Storage Strategy
- ❖ Password Encryption (e.g., crypt, hashing)
- ❖ Multi-Factor Authentication (MFA)
- ❖ OAuth / OpenID Connect
- ❖ Single Sign-On (SSO)
- ❖ Identity Providers (e.g., Google, Auth0, Keycloak)

Authorization (What are you allowed to do?)

- ❖ Role-Based Access Control (RBAC)
- ❖ Attribute-Based Access Control (ABAC)
- ❖ Access Tokens (e.g., JWT)

- ❖ Policy Engines (e.g., OPA – Open Policy Agent)
- ❖ Centralized vs. Decentralized Authorization
- ❖ Endpoint and Resource-Level Permissions

Topic_153
Software Design and Architecture
The Architectural Meta-Frame
Key Engineering Decisions___Part II
(Catching & State)

Caching and state management are crucial architectural decisions that impact **performance, scalability, and user experience.**
These decisions define **where data is stored, for how long**, and **how it's synchronized across components.**

Caching

1. Client-Side vs. Server-Side Caching
2. In-Memory Caching (e.g., Redis, Memcached)
3. Cache Invalidation Strategies (e.g., TTL, LRU)
4. CDN Caching (e.g., Cloudflare, Akamai)
5. Cache Consistency Models

State Management

1) Stateless vs. Stateful Architecture
2) Session Storage (In-Memory, Database, Cookies)
3) Distributed State Management
4) External State Stores (e.g., Redis, DB)
5) Eventual Consistency vs. Strong Consistency

Topic_154

PREPARED BY ALMAS AFZAL

Software Design and Architecture
The Architectural Meta-Frame
Key Engineering Decisions___Part III
(Communication)

Communication is a key engineering concern in system architecture, especially for **distributed systems, micro-services, and cloud-native applications.**
It defines **how components interact,** exchange data, and remain reliable and consistent.

Topic_155
Software Design and Architecture
The Architectural Meta-Frame
Key Engineering Decisions___Part IV
(Concurrency & Transaction)

PREPARED BY ALMAS AFZAL

Concurrency and **transaction management** are critical in designing systems that handle **simultaneous operations**, ensure **data consistency**, and maintain **system reliability**, especially in **multi-user** or **distributed environments**.

Topic_156
Software Design and Architecture
The Architectural Meta-Frame
Key Engineering Decisions___Part V
(Data Access)

Data Access decisions define how the application interacts with data storage systems. These choices impact **performance, scalability, consistency, and flexibility** of the architecture.

Data Access Decisions

- Database Type (Relational vs. NoSQL vs. NewSQL)

- Data Access Patterns (e.g., Repository, DAO)
- ORM Usage (e.g., Hibernate, Entity Framework)
- Query Strategy (Raw SQL, LINQ, Query Builders)
- Connection Pooling
- Caching for Data Access

Topic_157
Software Design and Architecture
The Architectural Meta-Frame
Key Engineering Decisions___Part V
(User Experience UX)

User Experience (UX) decisions influence how users **interact with and perceive** the system.
Architectural decisions must support **responsive, accessible, and consistent** interfaces that meet usability goals.

Topic_158
Software Design and Architecture
(Agility & Architecture Design)

Agility in architecture design refers to creating software architectures that are **flexible**, **responsive to change**, and aligned with **Agile development principles**.

It focuses on balancing **long-term structure** with **short-term adaptability**, allowing teams to **deliver value continuously** without over-engineering.

Topic_159
Software Design and Architecture
(Key Architecture Principles)

Architecture principles are **fundamental guidelines** that help ensure the architecture is **robust**, **scalable**, **maintainable**, and aligned with **business and technical goals**.

PREPARED BY ALMAS AFZAL

They guide decision-making throughout the software lifecycle.

Key Architecture Principles

- Separation of Concerns
- Single Responsibility Principle
- Modularity
- Reusability
- Loose Coupling
- High Cohesion
- Scalability

Topic_160
Software Design and Architecture
(Incremental & iterative Approach to Architectural Design)

The **Incremental & Iterative approach** to architectural design emphasizes building architecture **gradually**, with **continuous feedback and improvement**, rather than attempting to finalize the entire architecture up front.

This approach aligns with **Agile** and **modern development methodologies**.

Topic_161
Software Design and Architecture
(Baseline & Candidate Architectures)

What is Software Architecture?

- Software Architecture is the overall structure or blueprint of a software system.
- It shows how different parts (modules) of the system work together.
- Like a building plan for software.

PREPARED BY ALMAS AFZAL

What is a Baseline Architecture?

- Baseline Architecture is the starting point or the current state of the system's design.
- It shows what exists now.
- Used as a reference to compare with future versions.

Example:

A university's current student portal is your baseline architecture. It has login, course registration, and profile features.

What is a Candidate Architecture?

- Candidate Architecture is a new proposed design.
- It is what the software might look like in the future.
- We use it to solve new problems or add new features.

Example:

You design a better version of the student portal with online payments and chatbot support — this is your candidate architecture.

Comparasion:

Feature	Baseline Architecture	Candidate Architecture
Meaning	Current design	Future/Proposed design
Purpose	Reference point	Improve system
Change	No change yet	Includes changes
Used for	Comparison	Planning new system

PREPARED BY ALMAS AFZAL

Topic_162
Software Design and Architecture
(Architectural Spikes)

What is a Spike in Software Development?

- A spike is a quick, small experiment to learn something.
- It helps developers explore an idea, test a concept, or solve a tricky problem before fully building it.

What is an Architectural Spike?

- An Architectural Spike is a type of spike focused on software architecture.
- It is used to test different design choices for the software's structure.
- Helps in deciding which architecture will work best before committing to it.

Why Use Architectural Spikes?

- To reduce risk of bad design.
- To try out multiple options for system structure.
- To discover problems early.
- To improve decisions in big or complex systems.

Example (Simple):
Suppose you're building a social media app and you're not sure whether to use Microservices or Monolithic architecture.
You can create small test versions (spikes) of both and check:

- Which is faster?
- Which is easier to scale?
- Which is more secure?

Then choose the better one for your final design.

MCQs

- Spikes are **experiments**, not final code.
- Architectural spikes are used **before** making big design decisions.
- Saves **time, money**, and **effort** in the long run.

Topic_163
Software Design and Architecture
(Architecturally Significant Use Cases)

What is a Use Case?

A use case describes how a user interacts with the system to achieve a goal.

Example: "User logs into the system" or "Admin deletes a user."

What is an Architecturally Significant Use Case (ASUC)?

PREPARED BY ALMAS AFZAL

These are special use cases that strongly affect the software architecture.

They are important because they influence design decisions like:

1. How to structure the system
2. What technologies to use
3. What performance or security features are needed

Why are ASUCs Important?
Help to identify:

- ❖ Risks
- ❖ Design challenges
- ❖ Core system requirements

Guide the selection of:

- ❖ Architectural patterns
- ❖ Frameworks and tools

PREPARED BY ALMAS AFZAL

Topic_164
Software Design and Architecture
(Reference Application Architecture)

What is Reference Architecture?

A Reference Architecture is like a **template** or **standard model** for building software systems.
It gives a **common structure, best practices,** and **design guidelines** that can be reused.

What is a Reference Application Architecture?

It is **a ready-made example** of a working application based on a reference architecture.

It **shows how to apply** the architecture in a **real project.**

Developers use it as a **guide** to build similar applications.

Why Use a Reference Application Architecture?

- **Saves time –** No need to design everything from scratch.
- **Improves quality –** Based on proven solutions.
- **Makes communication easier –** All developers follow the same model.
- **Reduces risk –** Fewer chances of design errors.

Topic_165
Software Design and Architecture
(Software Architecture patterns___Introduction)

What is a Software Architecture Pattern?

- ✓ An **architecture pattern** is a **general solution** to a **common design problem** in software.
- ✓ It helps in **organizing** and **structuring** the system in a good way.
- ✓ Think of it like a **blueprint** or **design style** for your **software's structure.**

Why Use Architecture Patterns?

To build software that is:

- ✓ Easy to understand
- ✓ Easy to maintain
- ✓ Scalable (can grow with more users or data)
- ✓ Secure and efficient

*These patterns are **tried and tested** in many projects.*

Topic_166
Software Design and Architecture

PREPARED BY ALMAS AFZAL

(D/f b.w Software Architecture patterns & Design Patterns)

1. Definition

Software Architecture Patterns:
These are high-level structures that define the overall layout and communication between major parts (components) of a software system.
➢ **Example:** Layered Architecture, Client-Server, Microservices

Design Patterns:
These are low-level solutions to common programming problems within a specific part of a system.
➢ **Example:** Singleton, Observer, Factory Pattern

2. Purpose

Architecture Patterns: Help to organize the whole system for scalability, performance, and maintenance.

Design Patterns: Help to solve specific coding problems during software design and development.

3. Level of Abstraction

Architecture Patterns: High-level – Focus on the structure of the entire application.

Design Patterns: Low-level – Focus on code-level design and object interactions.

4. Application Stage

Architecture Patterns: Used early in the SDLC, especially in system design phase.

Design Patterns: Used later in development, especially during implementation and detailed design.

5. Re-usability

Architecture Patterns: Reused for whole system structure.

Design Patterns: Reused in smaller parts of the code (e.g., class or method level).

Conclusion

Architecture patterns define how the system is structured, while design patterns define how to solve coding problems inside the structure.
Both are important for building efficient, reusable, and scale-able software.

PREPARED BY ALMAS AFZAL

Topic_167
Software Design and Architecture
(Types of Architecture Structures)

There are three categories of structures in Architecture Design:

1. Static Structures
2. Dynamic Structures
3. Deployemnet Structures

1. Static Structures

Focus: System components and their relationships (what the system is) Shows:

- Classes
- Modules
- Packages
- Data types
- Their dependencies and organization

Example: Class diagram in UML.
Purpose: Helps in understanding code organization and module relationships.

2. Dynamic Structures

Focus: Behavior of the system during execution (how the system acts) Shows:

- How components interact at runtime
- Control flow and data flow
- Sequence and collaboration

Example: Sequence diagrams, Activity diagrams.
Purpose: Useful for designing features, use cases, and real-time behavior of the system.

3. Deployment Structures

Focus: Physical deployment of software on hardware nodes Shows:

- Servers
- Devices
- Network configurations
- Where and how components are deployed

PREPARED BY ALMAS AFZAL

Example: Deployment diagram in UML.
Purpose: Helps in planning performance, scalability, and networking.

Topic_168
Software Design and Architecture
Module Patterns
(Layered Pattern_I__ Introduction)

What is Layered Architecture Pattern?

Layered architecture is a design pattern that breaks the software system into multiple layers, where:

- Each layer performs a specific task
- One layer communicates only with the layer directly below or above it

Layer No.	Layer Name	Description
1	**Presentation Layer**	User interface (UI), handles user interaction.
2	**Application Layer**	Coordinates application activities and workflows.
3	**Domain Layer**	Contains core business logic and rules.
4	**Data Access Layer**	Handles database operations (read/write).
5	**Database Layer**	Actual data storage (e.g., MySQL, MongoDB).

Topic_169
Software Design and Architecture
Module Patterns
(Layered Pattern_II__Open vs Closed Layers)

In **Layered Architecture**, each layer is designed to communicate with other layers in a specific way.

There are **two main types of layer access control**:

1. Closed Layer

Definition:
A closed layer can only be accessed by the layer directly above it.
Upper layers are not allowed to skip layers and access lower ones directly.

PREPARED BY ALIAS AFZAL

Example:

- ❖ Presentation → Application → Domain → Data Access → Database
- ❖ The Presentation layer cannot directly call the Data Access layer.

Advantages:

- ➢ Maintains strict structure
- ➢ Easy to maintain and test
- ➢ Promotes separation of concerns

Disadvantages:

May result in slightly slower performance due to strict flow.

3. Open Layer

Definition:
An open layer can be accessed by any upper layer directly.
Layers can bypass intermediate layers if needed.

Example:
The Presentation layer can directly access the Data Access layer

Advantages:

- ➢ More flexible
- ➢ Can improve performance in some cases

Disadvantages:

- ➢ Reduces structure and discipline
- ➢ Makes debugging and maintenance more difficult

Topic_170
Software Design and Architecture
Module Patterns
(Layered Pattern_III__Tiers vs Layers)

In software architecture, the terms “tiers” and “layers” are often used, and they sound similar — but they are not the same.

Understanding the difference between tiers and layers is important in designing scalable and maintainable systems.

PREPARED BY ALMAS AFZAL

1. What is a Layer?

- A layer is a logical separation of code based on responsibilities.
- Layers are about how the code is organized within the application.
- All layers may still run on the same machine.

Example Layers:

- Presentation Layer (UI)
- Business Logic Layer
- Data Access Layer

2. What is a Tier?

- A tier is a physical separation of system components across different machines or servers.
- Tiers deal with deployment and network communication.
- Each tier can be on a different computer or server.

Example Tiers:

- Web server (Presentation Tier)
- Application server (Business Logic Tier)
- Database server (Data Tier)

Topic_171
Software Design and Architecture
Module Patterns
(Layered Pattern_IV__Advantages of Layered Architectures)

PREPARED BY ALMAS AFZAL

The **Layered Architecture Pattern** is one of the most widely used patterns in software design.
It divides the system into separate layers, each responsible for specific tasks.
This structure brings many advantages in terms of development, maintenance, and scalability.

Advantages of Layered Architecture

1. Separation of Concerns
– Each layer handles a specific responsibility.

2. Easy to Maintain and Modify
– You can change one layer without affecting others.

3. Re-usability

– Layers (like business logic) can be reused in other projects.

4. Test-ability
– Layers can be tested separately (unit testing).

5. Scalability
– System can be scaled by deploying layers on different servers.

6. Team Collaboration
– Teams can work on different layers independently.

7. Flexibility in Technology
– Different layers can use different technologies.

8. Better Security Management
– Security policies can be applied layer by layer.

9. Improved Code Organization
– Code is more structured, readable, and manageable.

Topic_172
Software Design and Architecture
Module Patterns
(Layered Pattern_V__Dis-Advantages of Layered Architectures)

PREPARED BY ALMAS AFZAL

While the Layered Architecture Pattern is widely used for its simplicity and structure, it also has some limitations and disadvantages, especially in complex or performance-critical systems.

Disadvantages of Layered Architecture

1) Performance Overhead
– Each request passes through multiple layers, which can slow down performance.

2) Tight Coupling Between Layers
– If not designed properly, layers may become dependent on each other, reducing flexibility.

3) Hard to Skip Layers When Needed
– The strict rule of moving layer-by-layer can be a problem in real-time or urgent processes where skipping is needed.

4) Difficult in Large Projects

– Managing many layers with complex logic in large systems can become hard and confusing.

5) Code Duplication
– Sometimes, similar logic may be written in multiple layers, leading to redundant code.

6) Not Suitable for All Applications
– It is not ideal for small apps or systems that require high speed and flexibility.

7) Delayed Communication Between Layers
– Communication takes more time when data has to pass through all layers, even if the task is simple.

8) Increased Development Time
– Strict layering and separation may require more time to design and implement.

Topic_173
Software Design and Architecture
Module Patterns
(Layered Pattern_V)
(Client-Server Architecture____two-tier Architecture)

PREPARED BY AIMAS AFZAL

What is Client-Server Architecture?
Client-Server Architecture is a two-tier architecture where the system is divided into two main parts:

Client: The part that sends requests (usually the user interface or front-end).
Server: The part that processes the request and returns a response (usually contains logic and data).

Structure of Two-Tier Architecture

Client (Tier 1) ↔ Server (Tier 2)

Client Tier:
– Handles user input and displays results.
– Example: Web browser, desktop app.

Server Tier:
– Handles business logic and database access.
– Example: A server running PHP or Java with MySQL.

Features of Client-Server Architecture

- **Two tiers only:** User interface (client) and logic + database (server) are separated.
- **Direct communication:** Client connects directly to the server using a network.
- **Simple design:** Easy to build and deploy for small-to-medium applications.

Advantages
- Easy to implement for small systems
- Faster communication between client and server
- Centralized control and security
- Low cost and less complexity

Disadvantages
- Not scalable for large systems
- Heavy load on the server
- Difficult to manage when many clients connect
- Client must be compatible with server settings and updates

Topic_174
Software Design and Architecture
Module Patterns
(Layered Pattern_VI___n-tier Architecture)

PREPARED BY ALMAS AFZAL

What is n-tier Architecture?
n-tier architecture is an extension of the client-server model, where the system is divided into more than two tiers (layers), usually 3 or more.
Each tier is physically separated and performs a specific role.

"n" refers to the number of tiers, such as 3-tier, 4-tier, etc.

Common Tiers in n-tier Architecture
Presentation Tier
– User interface layer
– **Example:** Web browser, mobile app

Application/Business Logic Tier
– Handles business rules and processing
– **Example:** PHP, Java, Python backend

Data Tier
– Handles data storage and retrieval
– **Example:** MySQL, Oracle database

Optional additional tiers may include:
4. Integration Tier – APIs or services that connect to external systems

5. Security Tier – Handles authentication and authorization

Advantages

- ❖ High scalability – Each tier can be scaled independently
- ❖ Improved maintainability – Easy to update one tier without affecting others
- ❖ Better security – Sensitive data can be isolated in separate tiers
- ❖ Re-usability – Business logic can be reused for multiple interfaces
- ❖ Supports large applications – Suitable for enterprise-level systems

Disadvantages

- ❖ Complex to design and manage
- ❖ Slower performance due to network communication between tiers
- ❖ More cost and resources required (servers, infrastructure)

Topic_175
Software Design and Architecture
Module Patterns
(Variations in Layered Patterns)

The Layered Architecture Pattern is flexible and can be modified in different ways to suit different types of software systems. These modifications are called variations of the layered pattern.

Why Variations Are Needed?

- ❖ Not all applications are the same
- ❖ Some systems need more performance
- ❖ Some require flexibility or fast communication
- ❖ So, developers adjust the layers based on project needs

Common Variations in Layered Patterns

PREPARED BY ALMAS AFZAL

1. Skipping Layers
2. Collapsing Layers
3. Duplicated Layers
4. Plug-in Layers
5. Micro-kernel or Minimal Core

Topic_176
Software Design and Architecture
Component & Connector Patterns
(Broker Pattern_I___Introduction)

PREPARED BY ALMAS AFZAL

What is the Broker Pattern?

The Broker Pattern is a software architecture pattern used to connect different parts of a distributed system.
It is especially useful when components are on different machines and need to communicate through a central coordinator.
The main idea of the pattern is to separate communication between clients and services using a broker.

Where is it Used?

In distributed systems where multiple components are running on different computers or networks
Example: Remote Procedure Calls (RPC), Web Services, CORBA, Java RMI

Main Components of Broker Pattern

Client

– Sends requests to use a service.

Broker

– The central communication handler.
– Receives client requests and sends them to the correct service.

Server (Component/Service)

– Performs the actual work or task requested by the client.

Stub/Proxy

– A helper component that hides the details of the network communication.

How It Works (Basic Flow)

- Client makes a request
- Request is sent to the Broker
- Broker locates the appropriate server

- Broker forwards the request to that server
- Server processes the request and returns the response via Broker

Conclusion

The Broker Pattern is a powerful architectural solution for building distributed systems. It hides the complexity of network communication and helps manage services and requests efficiently across machines.

Topic_177
Software Design and Architecture
Component & Connector Patterns
(Broker Pattern_II___Advantages & Dis-Advantages)

PREPARED BY ALMAS AFZAL

Advantages of Broker Pattern

- Supports distributed systems
- Loose coupling between components
- Location transparency
- Scalability
- Centralized management
- Improved re-usability
- Interoperability between different technologies

Disadvantages of Broker Pattern

- Broker becomes a bottleneck
- Complexity in implementation
- Single point of failure
- Security risks
- Debugging difficulties
- Latency overhead

Topic_178
Software Design and Architecture
Component & Connector Patterns
(MVC Patterns)

What is MVC Pattern?

The MVC Pattern is a design architecture that separates a software application into three main components:

Model – Handles the data and business logic
View – Manages the user interface (UI)
Controller – Handles user input and controls interaction between Model and View

This separation improves modularity, test-ability, and maintainability.

Advantages of MVC Pattern

- Separation of concerns
- Easy to maintain and test
- Reusable components
- Parallel development (UI and logic separately)
- Supports multiple views for the same data

Disadvantages of MVC Pattern

- More complex to design initially
- Tight coupling between components if not designed properly
- Overhead for small/simple applications

Topic_179
Software Design and Architecture
Component & Connector Patterns
(Pipe & Filter Patterns)

PREPARED BY ALMAS AFZAL

What is Pipe & Filter Pattern?

The Pipe and Filter Pattern is an architectural style where data flows through a sequence of processing steps (called filters), connected by pipes.

Each filter processes the input data and passes the output to the next filter via a pipe.

Advantages of Pipe & Filter Pattern

- Re-usability
- Modularity
- Simplicity
- Easy Testing
- Supports Parallel Processing

Disadvantages of Pipe & Filter Pattern

- Performance Overhead
- Debugging Difficulty
- Not Ideal for Complex Data Flow
- Latency

Topic_180
Software Design and Architecture
Component & Connector Patterns
(Pipe & Filter Patterns___Strength & Weakness)

PREPARED BY ALMAS AFZAL

Strengths

- Modularity
- Re-usability
- Easy Maintenance
- Test-ability
- Supports Parallelism
- Scalability
- Clear Data Flow

Weaknesses

- Performance Overhead
- Debugging Difficulty
- Lack of Flexibility
- High Latency
- Not Suitable for Interactive Systems

Topic_181
Software Design and Architecture
Component & Connector Patterns
(Client-Server Pattern)

What is Client-Server Pattern?

The Client-Server Pattern is a common architectural model where the system is divided into two main parts:

Client – The user-side that makes requests
Server – The provider-side that processes requests and sends responses

*The client and server communicate over a **network**.*

Structure:
Client ⇄ Server

Topic_182
Software Design and Architecture
Component & Connector Patterns
(Shared-Data Pattern)

What is Shared-Data Pattern?
The Shared-Data Pattern (also called Repository Pattern) is an architectural pattern where multiple components access and share a central data store (repository).

Each component performs different operations but works with the same shared data source.

Main Components
1. Repository (Data Store)
– Central storage where data is kept
– Can be a file system, database, or memory structure

2. Clients / Components
– Independent modules that read from and write to the repository
– Do not directly communicate with each other

Topic_183
Software Design and Architecture
Component & Connector Patterns
(Shared-Data Pattern___Strengths & Weaknesses)

PREPARED BY ALMAS AFZAL

Strengths

1) Centralized data management
2) Promotes data consistency
3) Easy data backup and security
4) Components are loosely coupled
5) Easy to add or remove components

Weaknesses

1. Repository becomes a performance bottleneck
2. Single point of failure
3. Difficult concurrency control
4. Complex synchronization and access control
5. Indirect communication between components

Topic_184
Software Design and Architecture
Allocation Patterns
(Multi-Tier Pattern)

What is the Multi-Tier Pattern?

The Multi-Tier Pattern is an architectural pattern used to organize and deploy software systems across multiple physical tiers (layers).
Each tier runs on a separate server or machine and handles a specific responsibility.
It is commonly used in web and enterprise applications.

Common Tiers in Multi-Tier Architecture

1) Presentation Tier

– User interface
– Handles user input/output
– **Example**: Web browser or mobile app

2) Application (Logic) Tier

– Contains business logic and processing
– **Example**: Server-side scripts (PHP, Java, Python)

3) Data Tier

– Stores and manages data
– **Example**: MySQL, Oracle database

PREPARED BY ALMAS AFZAL

Topic_185 & 186
Software Design and Architecture
Monolithic Architecture

What is Monolithic Architecture?

Monolithic Architecture is a software design where the entire application is built as a single, tightly connected unit.

All components — UI, business logic, and database access — are part of one codebase and run together.

Key Characteristics

- One single application package or executable
- All modules are deployed together
- Tight coupling between components
- Common in traditional/legacy systems

Advantages of Monolithic Architecture

- Simple to develop
- Easy to deploy
- Performance is good
- Easier debugging

Disadvantages of Monolithic Architecture

- Hard to maintain as the code-base grows
- Not scale-able (difficult to handle high traffic)
- Tight coupling makes changes risky
- Slow development in large teams
- Difficult to adopt new technologies for parts of the system

---

Topic_187
Software Design and Architecture
Web Application Architecture____Overview

PREPARED BY ALMAS AFZAL

What is Web Application Architecture?

Web Application Architecture defines the structure and interaction of all components involved in a web-based system.

It explains how the front-end (browser) and back-end (server, database) communicate to process requests and deliver responses.

Main Goal:

To ensure smooth, secure, and efficient interaction between users, servers, and data.

Basic Components

1. Client (Frontend)

– The user interface (browser/mobile)
– Built using HTML, CSS, JavaScript

2. Web Server (Backend)
– Processes user requests
– Built using PHP, Node.js, Python, Java, etc.

3. Database Server
– Stores and retrieves application data
– Example: MySQL, MongoDB

Common Web Architecture Models
1) Two-Tier Architecture
(Client ↔ Server with database)

2) Three-Tier Architecture
(Client ↔ Application Server ↔ Database)

3) n-Tier Architecture
(Multiple layers like security, APIs, etc.)

Topic_188
Software Design and Architecture
Web Application Design Considerations
Part I_____Maintainability

PREPARED BY ALMAS AFZAL

What is Maintainability?
Maintainability means how easy it is to update, fix, or improve a web application after it has been developed and deployed.

A maintainable application can be changed or improved with less effort, less time, and fewer bugs.

Why is Maintainability Important?
- ❖ Web applications often need updates (e.g., new features, security fixes)
- ❖ A poorly designed system is hard to understand and modify
- ❖ Good maintainability saves cost, time, and developer effort

Key Factors That Improve Maintainability
Modular Design
– Code is divided into small, independent parts (modules)

Clear Code Structure
– Code is well-organized and easy to read

Consistent Coding Standards
– All developers follow the same style and naming rules

Documentation
– Comments and guides help future developers understand the system

Low Coupling, High Cohesion
– Components are independent but focused on specific tasks

Benefits of Maintainability
- ✧ Faster bug fixing
- ✧ Easier feature upgrades
- ✧ Better team collaboration
- ✧ Long-term cost savings
- ✧ Reliable system performance over time

Conclusion
Maintainability is a critical part of web application design.
It ensures that your app can grow, improve, and stay secure over time without becoming too complex or costly to manage.

Topic_189
Software Design and Architecture
Web Application Design Considerations
Part II_____Performance

PREPARED BY ALMAS AFZAL

What is Performance in Web Applications?
Performance refers to how fast and efficiently a web application responds to user actions, processes data, and loads content.

A good performance means the website is fast, smooth, and responsive, even with many users.

Why is Performance Important?

Users expect quick page load times **(within 2–3 seconds)**
Slow websites lose users and traffic
Impacts SEO ranking, user experience, and revenue

Key Factors That Improve Performance
Efficient Code
– Write optimized backend and frontend code

Caching
– Store frequently used data temporarily to avoid repeated processing

Minimize Server Requests
– Reduce the number of files (CSS, JS, images) loaded on each page

Database Optimization
– Use proper indexing and avoid unnecessary queries

Content Delivery Network (CDN)
– Deliver static content (images, CSS, JS) from servers closer to users

Load Balancing
– Distribute user requests across multiple servers

Benefits of High Performance
- Better user satisfaction
- Higher conversion rates
- Lower bounce rates
- Improved search engine ranking
- Handles more users at once (scalability)

Conclusion
Performance is a key part of web application design.
A fast and responsive web app leads to better user experience, SEO, and business success.
Designers and developers must plan for performance from the start of the project.

Topic_190
Software Design and Architecture
Web Application Design Considerations
Part III_____Security

PREPARED BY ALMAS AFZAL

What is Security in Web Applications?
Security means protecting the web application, its users, and its data from unauthorized access, hacking, or misuse.

A secure application ensures that only trusted users can access or modify data and that sensitive information is safe.

Why is Security Important?
- Prevents data theft, hacking, and attacks
- Protects user privacy and company reputation
- Ensures compliance with legal and data protection laws (e.g., GDPR)
- Builds user trust

Common Security Threats

1. SQL Injection – Attacker runs harmful database queries
2. Cross-Site Scripting (XSS) – Attacker injects malicious scripts
3. Cross-Site Request Forgery (CSRF) – Attacker tricks user to perform unwanted actions
4. Authentication Bypass – Unauthorized access to user accounts
5. Data Breaches – Sensitive data exposed due to poor security

Benefits of Strong Security

✧ Keeps user data safe
✧ Protects business reputation
✧ Reduces legal risks
✧ Increases user confidence
✧ Prevents system downtime due to attacks

Topic_191
Software Design and Architecture
Web Application Frame_____Introduction

What is a Web Application Framework?

A Web Application Framework is a pre-built structure or toolkit that helps developers create web applications faster and more easily.

It provides ready-made code, libraries, and tools to handle common tasks like:

❖ Routing (handling URLs)
❖ Database connection
❖ Session management
❖ Security
❖ Page rendering

Why Use a Framework?

❖ Instead of writing everything from scratch, a framework:
❖ Saves time and effort
❖ Follows best practices and design patterns
❖ Makes code cleaner, modular, and maintainable
❖ Helps build secure and scale-able applications

PREPARED BY ALMAS AFZAL

Topic_192
Software Design and Architecture
Web Application Frame_____Authentication

What is Authentication?

Authentication is the process of verifying the identity of a user.
It ensures that the person trying to access the application is who they claim to be.
Example: Logging in with a username and password.

Role of Frameworks in Authentication

Web application frameworks provide built-in tools and libraries to implement authentication easily and securely.

They save time and reduce the risk of security mistakes.

Common Authentication Features in Frameworks

User Login & Logout Functions

– Handle user sessions automatically

Registration & Password Management

– User sign-up, password reset, password hashing

Session Management

– Track logged-in users using cookies or tokens

Middleware Support

– Check if user is logged in before allowing access to pages

Two-Factor Authentication (2FA) (Optional)

– Adds extra security layer using mobile code or email

Topic_193
Software Design and Architecture
Web Application Frame_____Authorization

PREPARED BY ALMAS AFZAL

What is Authorization?

Authorization is the process of controlling what a user is allowed to do after they have been authenticated (logged in).

It answers the question:

"Does the user have permission to access this resource or perform this action?"

Authentication vs Authorization (Quick Difference)

Authentication	Authorization
Confirms **who** the user is	Controls **what** the user can do
Comes **first**	Happens **after authentication**

Authentication	Authorization
Example: Login	Example: Access to admin panel

Types of Authorization

Role-Based Access Control (RBAC)

– Users are assigned roles (e.g., Admin, User, Guest)
– Permissions are based on roles

Permission-Based Access

– Specific actions allowed based on individual user permissions

Policy-Based Access

– Rules and conditions decide what access is allowed (e.g., age > 18)

Topic_194
Software Design and Architecture
Web Application Frame_____Caching

PREPARED BY ALMAS AFZAL

What is Caching?

Caching is the process of storing frequently used data in a temporary storage (cache) so that it can be quickly accessed later without reloading from the original source (like a database).

Caching improves the speed and performance of a web application.

Why is Caching Important?

- ❖ Reduces loading time
- ❖ Decreases server load
- ❖ Improves user experience
- ❖ Saves bandwidth and processing power

Types of Caching

Page Caching

– Stores entire pages for faster loading

Data/Object Caching

– Caches specific data like user profiles, product lists

Query Caching

– Saves results of frequent database queries

Browser Caching

– Static files (images, CSS, JS) stored in the user's browser

Opcode Caching (e.g., PHP)
– Stores compiled script code to avoid re-parsing on every request

Benefits of Caching
- ❖ Faster response time
- ❖ Better scalability
- ❖ Lower database usage
- ❖ Smooth performance under high traffic

Topic_195
Software Design and Architecture
Web Application Frame_____Exception Management

PREPARED BY ALMAS AZAL

What is Exception Management?
Exception Management is the process of detecting, handling, and logging errors (called exceptions) that occur during the execution of a web application.

The goal is to prevent application crashes, show friendly error messages, and help developers fix bugs quickly.

Why is Exception Management Important?
- ➢ Ensures application stability and reliability
- ➢ Protects against unexpected behavior
- ➢ Improves user experience
- ➢ Helps developers debug and fix issues faster
- ➢ Prevents security risks from unhand-led errors

Types of Exceptions in Web Applications
- ❖ **Runtime Errors** – Division by zero, null pointer, etc.
- ❖ **Database Errors** – Connection failures, query errors
- ❖ **Validation Errors** – Invalid user input
- ❖ **Network Errors** – Timeouts, server not found
- ❖ **Security Errors** – Unauthorized access, token errors

Topic_196
Software Design and Architecture
Web Application Frame_____Logging & Instrumentation

What is Logging?

Logging means recording important events or messages during the execution of a web application.
It helps developers track what the system is doing and find problems when something goes wrong.

What is Instrumentation?

Instrumentation is the process of measuring and monitoring the performance and behavior of a web application.
It includes collecting data like response time, resource usage, errors, etc., for analysis and debugging.

Why Logging & Instrumentation Are Important

- ❖ Help in debugging errors
- ❖ Detect security threats or system failures
- ❖ Monitor performance and usage trends
- ❖ Provide insights for maintenance and improvements
- ❖ Required for audit trails and compliance

Common Things Logged

1. User logins and activity
2. Errors and exceptions
3. API requests and responses
4. Database access issues
5. Server warnings and crashes

Logging Levels

1) **INFO** – General events (e.g., user logged in)
2) **DEBUG** – Developer-level details
3) **WARNING** – Something went wrong, but app can continue
4) **ERROR** – Serious issue, needs fixing
5) **CRITICAL** – System failure, immediate attention needed

Topic_197
Software Design and Architecture
Web Application Frame_____Navigation

What is Navigation in Web Applications?

Navigation refers to how users move between different pages or sections of a web application.
It ensures users can easily find information and perform tasks without confusion.

A good navigation structure improves usability, accessibility, and user satisfaction.

Types of Navigation

PREPARED BY ALMAS AFZAL

Top Navigation (Header Bar)
– Appears at the top of every page
– Contains links like Home, About, Contact

Side Navigation (Sidebar)
– Vertical menu on the left or right side
– Used in dashboards and admin panels

Footer Navigation
– At the bottom of the page
– Contains secondary links (e.g., Terms, Privacy Policy)

Breadcrumb Navigation
– Shows the user's path (e.g., Home > Products > Phones)

Dropdown Menus / Hamburger Menus
– Collapsible menus for mobile or compact UI

Benefits of Good Navigation
1. Improves user experience
2. Increases engagement and usability
3. Reduces bounce rate
4. Helps users complete tasks faster
5. Supports accessibility for all devices and users

Conclusion
Navigation is a critical part of web application design.
Web frameworks offer tools to build structured, dynamic, and user-friendly navigation, making it easier for users to find their way and interact with the application smoothly.

Topic_198
Software Design and Architecture
Web Application Frame_____Page Layout

PREPARED BY ALMAS AFZAL

What is Page Layout in Web Applications?
Page Layout refers to the arrangement of visual elements (like headers, sidebars, content areas, footers) on a web page.
It defines how content is organized, aligned, and displayed to the user.

A good page layout improves readability, navigation, and user interaction.

Key Elements of a Typical Web Page Layout
Header
– Usually contains logo, navigation menu, search bar

Sidebar (optional)
– Used for navigation, filters, or extra options

Main Content Area
– Displays the primary page content (text, forms, tables, etc.)

Footer
– Contains copyright, links, contact info

Banner/Notifications
– Shows alerts or updates to users

Common Layout Structures
- ❖ Single Column – All content in one vertical flow
- ❖ Two Column – Sidebar + Main Content
- ❖ Three Column – Left Sidebar + Content + Right Sidebar
- ❖ Grid-Based Layout – Divides page into rows and columns for flexible design
- ❖ Responsive Layout – Automatically adjusts for mobile, tablet, and desktop screens

---

Topic_199
Software Design and Architecture
Web Application Frame_____Page Rendering

PREPARED BY ALMAS AFZAL

What is Page Rendering?
Page Rendering is the process of generating and displaying the web page content in the browser.
It transforms code (HTML, CSS, JS) into a visible and interactive page that users can see and use.

Types of Page Rendering
Server-Side Rendering (SSR)
– The server generates the full HTML page and sends it to the browser.
– Example: PHP, Laravel, Django

Client-Side Rendering (CSR)
– The browser downloads a blank HTML page and uses JavaScript (usually with frameworks like React, Angular) to fill in the content.

Static Rendering
– Pre-built HTML pages served as-is from the server.
– Fastest, often used with static site generators.

Hydration / Hybrid Rendering

– Combines SSR and CSR: initial page is server-rendered, then becomes interactive via JavaScript.
– Example: Next.js

Benefits of Efficient Page Rendering
- ❖ Faster page load time
- ❖ Improved user experience
- ❖ Better search engine visibility (SEO)
- ❖ Reduced server load (CSR)
- ❖ Smooth interactivity (CSR + hydration)

Topic_200
Software Design and Architecture
Web Application Frame_____Presentation Entity

PREPARED BY ALMAS AFZAL

What is a Presentation Entity?
A Presentation Entity is a type of data structure or object that is specifically designed to transfer data between the backend and the frontend (user interface).

It helps format, filter, and organize the data in a way that is suitable for display on the web page.

Why Use Presentation Entities?
- ❖ To separate business logic (actual data processing) from the presentation layer (UI)
- ❖ To avoid sending raw database models directly to the user
- ❖ To customize or reshape data before displaying it
- ❖ To improve security by hiding sensitive information
- ❖ To simplify frontend rendering

Where Are Presentation Entities Used?
They are used in:

1. **MVC Frameworks:** as "View Models" or "DTOs (Data Transfer Objects)"
2. **API Responses:** to format JSON output
3. **Templates:** to feed data into HTML views

Topic_201
Software Design and Architecture
Web Application Frame_____Request Processing

What is Request Processing?

Request Processing refers to the step-by-step handling of a user's request by the web application.
It begins when a user clicks something in the browser (like a button or link), and ends when they receive a response (like a web page or message).

Benefits of Proper Request Processing

- ❖ Faster and efficient responses
- ❖ Easier debugging and maintenance
- ❖ Clear structure and flow
- ❖ Better scalability and flexibility
- ❖ Secure and consistent handling of user actions

Conclusion

Request Processing is a core part of how web frameworks work.
It helps transform user actions into meaningful system responses.
Modern frameworks simplify this process and make it easy to build well-structured, responsive applications.

Topic_202
Software Design and Architecture
Web Application Frame_____Session Management

PREPARED BY AMAS AFZAL

What is Session Management?

Session Management is the process of tracking a user's activity across multiple requests during a visit to a web application.

A session stores temporary data (like login info) for a user while they are using the site.

Why Is Session Management Important?

1. Keeps users logged in across pages
2. Stores user-specific data (cart items, preferences)
3. Maintains state in stateless HTTP protocol
4. Helps in authentication and authorization
5. Prevents unauthorized access

Common Data Stored in a Session

- ❖ User ID or username
- ❖ Login status
- ❖ User roles/permissions
- ❖ Cart data (in e-commerce apps)
- ❖ Temporary form data

Topic_203
Software Design and Architecture
Web Application Frame_____Validation

What is Validation?
Validation means checking user input to ensure that it is correct, safe, and complete before processing or saving it in the system.

Example: Making sure an email address field contains a valid email, or that a password is not empty.

Why is Validation Important?
- ❖ Prevents invalid or harmful data from entering the system
- ❖ Protects against security threats like SQL Injection or XSS
- ❖ Helps maintain data integrity
- ❖ Improves user experience by giving instant feedback
- ❖ Ensures application behaves correctly

Types of Validation
1. Client-Side Validation
– Done in the browser using JavaScript or HTML5
– Quick feedback but not secure alone

2. Server-Side Validation
– Performed on the server before saving data
– Essential for security

3. Form Validation
– Checks required fields, data format, matching fields, etc.

4. Custom Validation
– Business logic-specific rules (e.g., age must be > 18)

Benefits of Validation
- ➢ Keeps application safe and secure
- ➢ Ensures reliable and clean data
- ➢ Enhances user trust and experience
- ➢ Prevents unexpected errors and crashes

PREPARED BY ALMAS AFZAL

Topic_204
Software Design and Architecture
Web Application Frame

(Presentation Layer Considerations)

What is the Presentation Layer?

The Presentation Layer is the part of a web application that users see and interact with. It includes all UI components, such as:

- Forms
- Buttons
- Menus
- Tables
- Layouts
- Visual feedback (messages, loaders, alerts)

Purpose of the Presentation Layer

- To display data to users clearly and attractively
- To receive user input and pass it to the back-end
- To ensure easy navigation and good user experience

Key Considerations for the Presentation Layer

1. User Experience (UX)

– Easy to use
– Clear flow and actions
– Helpful feedback and tooltips

2. Consistency

– Same design style across pages
– Reusable components (e.g., same button styles, headers)

3. Responsiveness

– Layout should adjust on mobile, tablet, desktop

4. Accessibility

– Supports screen readers
– High contrast, keyboard navigation

5. Performance

– Load pages quickly
– Use optimized images, lazy loading, and minimal JS/CSS

6. Validation & Feedback

– Show errors and success messages properly
– Real-time input checks

7. Separation of Concerns

– Keep logic (back-end) and presentation (front-end) separate
– Use template engines or front-end frameworks

PREPARED BY ALMAS AFZAL

8. Security
– Avoid exposing sensitive data
– Escape content to prevent XSS attacks

Topic_205 to 207
Software Design and Architecture
Web Application Frame
(Business Layer Considerations)

PREPARED BY ALMAS AFZAL

What is the Business Layer?
The Business Layer (also called Logic Layer) is the part of the application that handles the core logic, rules, and processing.

It connects the Presentation Layer (UI) with the Data Layer (database).

Purpose of the Business Layer
- ❖ Processes user input
- ❖ Applies business rules
- ❖ Coordinates data flow between UI and database
- ❖ Ensures correct calculations, decisions, and workflows

Key Considerations for the Business Layer
1. Separation of Concerns
– Keep business logic separate from UI and data code
– Makes the system easier to maintain

2. Re-usability
– Write logic in reusable functions or services
– Useful in multiple places (e.g., order total calculation)

3. Scalability
– Business logic should support growth (more users, features)

4. Test-ability
– Keep logic in units (classes/functions) that can be easily tested

5. Security
– Enforce rules like access permissions, validation, and workflows
– Never rely only on UI checks

Topic_208
Software Design and Architecture

Web Application Frame
(Testing & Test-ability Considerations)

What is Testing in Web Applications?

Testing is the process of checking if the web application works correctly and meets user requirements.

It ensures the system is free of bugs, performs well, and behaves correctly in different scenarios.

What is Test-ability?

Test-ability means how easily and effectively you can test a web application.
A well-designed system is easier to test, debug, and fix.

Why Are Testing & Test-ability Important?

- ❖ Catch errors before deployment
- ❖ Ensure reliable and stable performance
- ❖ Make it easier to update or modify the system
- ❖ Support automated testing
- ❖ Improve developer confidence

PREPARED BY ALMAS AFZAL

Types of Testing in Web Applications

1. Unit Testing

– Test small pieces of code (like functions, methods)

2. Integration Testing

– Test how modules work together (e.g., front-end with back-end)

3. Functional Testing

– Test user features (e.g., login, signup)

4. UI/End-to-End Testing

– Simulate full user behavior in the browser

5. Performance Testing

– Test speed and load capacity

6. Security Testing

– Find vulnerabilities and data risks

(Performance Considerations)

What is Performance in Web Applications?
Performance means how fast, responsive, and efficient a web application is while handling user actions, loading pages, and processing data.

Why Performance Matters.?
1) Improves user experience
2) Increases conversion and engagement
3) Reduces bounce rate
4) Saves server resources
5) Helps with SEO (Google ranks faster websites better)

Topic_210
Software Design and Architecture
Web Application Frame
(Security Considerations)

What is Security in Web Applications?
Security means protecting your web application from unauthorized access, data theft, and malicious attacks.
It ensures only trusted users can access data and perform actions.

Why Security is Important.?
- Protects user data and company information
- Prevents hacking and cyber attacks
- Builds user trust
- Required for legal and compliance reasons
- Avoids financial and reputation loss

Topic_211
Software Design and Architecture
Web Application Frame
(Deployment Considerations)

What is Deployment in Web Applications?
Deployment means making a web application live on a server so users can access it online.
It includes setting up servers, transferring code, configuring environment, and monitoring the system.

PREPARED BY ALMAS AFZAL

Why Deployment is Important.?

- Makes the application available to real users
- Ensures correct environment setup
- Supports scalability, performance, and uptime
- Involves security, backups, and monitoring

Topic_212
Software Design and Architecture
Web Application Frame
(Load Balancing)

What is Load Balancing?

Load Balancing is the process of distributing incoming network traffic across multiple servers.
This ensures no single server gets overloaded and helps keep the web application fast, reliable, and available.

Why Load Balancing is Important.?

- Increases availability (handles more users)
- Improves performance (faster response times)
- Provides fault tolerance (if one server fails, others continue)
- Supports scalability (easier to add/remove servers)

Topic_213
Software Design and Architecture
(Cloud Computing____Introduction)

What is Cloud Computing?

Cloud Computing means using the internet to access computing resources like servers, storage, databases, and software instead of owning and managing physical hardware.

It allows users and companies to use IT services on-demand, like renting instead of buying.

Key Idea:

"Pay only for what you use" *– No need to buy big servers, just use online services and pay monthly or hourly.*

Main Characteristics of Cloud Computing

1. **On-Demand Self-Service**

– Users can get services anytime without human interaction.

2. **Broad Network Access**

PREPARED BY ALMAS AFZAL

– Services are available via the internet from anywhere.

3. Resource Pooling
– Servers and resources are shared among many users (multi-tenant model).

4. Rapid Elasticity
– Resources (like storage or CPU) can scale up/down quickly based on need.

5. Measured Service
– Usage is monitored and billed automatically.

Types of Cloud Models

Model	Description	Example
Public	Owned by cloud providers, shared with others	AWS, Google Cloud, Azure
Private	Used by one organization only	Company's internal cloud
Hybrid	Mix of public + private	Cloud + on-premises setup

PREPARED BY ALMAS RIZAL

Service Models of Cloud Computing

Service Model	What It Offers	Example
IaaS	Infrastructure (VMs, storage, networks)	AWS EC2, Azure VM
PaaS	Platform for app development	Google App Engine, Heroku
SaaS	Ready-to-use software over the internet	Gmail, Dropbox, Zoom

Benefits of Cloud Computing
✧ **Cost-Effective** – No need to buy hardware
✧ **Scale-able** – Grows with your needs
✧ **Reliable** – Backup and recovery options
✧ **Accessible** – Available from anywhere
✧ **Fast Deployment** – Services can be set up quickly

Topic_214
Software Design and Architecture
(Cloud Computing____Basic Definitions)

1. Cloud Computing

Cloud computing is the delivery of computing services like servers, storage, databases, and software over the internet.
It allows users to access resources on-demand and pay only for what they use.

2. Cloud Provider

A cloud provider is a company that offers cloud services.
Examples include:

- ✧ Amazon Web Services (AWS)
- ✧ Microsoft Azure
- ✧ Google Cloud Platform (GCP)

3. IaaS (Infrastructure as a Service)

Provides basic computing infrastructure such as virtual machines, storage, and networks.
Users manage the operating system and applications.
Example: AWS EC2

4. PaaS (Platform as a Service)

Provides a platform for developers to build, test, and deploy applications.
The cloud provider manages the infrastructure.
Example: Google App Engine, Heroku

5. SaaS (Software as a Service)

Ready-to-use software accessed through the internet.
Users do not manage the infrastructure.
Example: Gmail, Dropbox, Microsoft 365

6. Virtual Machine (VM)

A software-based simulation of a physical computer.
It allows multiple operating systems to run on one physical server.

7. Scalability

The ability of the cloud system to increase or decrease resources based on demand.

8. Elasticity

Automatic adjustment of cloud resources depending on traffic or usage.
More users = more resources; fewer users = fewer resources.

9. Multi-Tenancy

A single cloud system is shared by multiple users (tenants), while keeping each user's data secure and separate.

10. Cloud Deployment Models

- ❖ Public Cloud: Services are shared and accessible via the internet
- ❖ Private Cloud: Services are used by one organization only

PREPARED BY ALMAS AFZAL

❖ Hybrid Cloud: A combination of public and private cloud models

11. Pay-as-you-go
Users only pay for what they use — no need to buy full servers or software in advance.

12. Service Level Agreement (SLA)
A formal agreement between the cloud provider and the customer that defines expected performance, availability (uptime), and support.

13. Cloud Storage
A service that allows users to store and access data online instead of using physical drives.
Example: Google Drive, Dropbox

Topic_215
Software Design and Architecture
Cloud Computing
(Service Models & Deployemnet Options)

PREPARED BY ALMAS AFZAL

1. Cloud Service Models
These are the ways cloud services are provided to users, based on what level of control and responsibility they need.

Feature	IaaS	PaaS	SaaS
Control Level	High	Medium	Low
User Manages	OS, apps	Apps only	Just uses app
Used By	Developers	Developers	End-users
Example	AWS EC2	Heroku	Gmail, Zoom

2. Cloud Deployment Models
These define where the cloud infrastructure is hosted and who can access it.

Model	Access	Control Level	Cost	Example Use
Public Cloud	Shared (Public)	Low	Low	Startups, websites
Private Cloud	Single Org	High	High	Banks, Government
Hybrid Cloud	Mixed	Medium	Medium	Enterprises with sensitive +

Model	Access	Control Level	Cost	Example Use
				public data

Topic_216
Software Design and Architecture
Cloud Computing (Multi-tenancy)

What is Multi-Tenancy?

Multi-tenancy is a cloud architecture where a single software system or infrastructure serves multiple users (called tenants).
Each tenant's data and usage are kept separate, even though they use the same system.

Who are Tenants?

Tenants can be:

1. Individual users
2. Companies
3. Clients using the same application

Key Features of Multi-Tenancy

Resource Sharing

– Servers, databases, and code are shared among all tenants

Data Isolation

– Each tenant's data is kept separate using IDs or schemas

Single Codebase

– Only one version of the application needs to be updated or maintained

Cost Efficiency

– Cloud providers can serve many users using fewer resources

Advantages of Multi-Tenancy

- ➢ Lower cost per user
- ➢ Easier updates (update once for all users)
- ➢ Scalable (can handle many users)
- ➢ Centralized maintenance

Challenges in Multi-Tenancy

- ✧ Data Security & Privacy must be strong
- ✧ Performance isolation (one user should not slow down others)
- ✧ Complex access control and customization

PREPARED BY ALMAS AFZAL

Topic_217
Software Design and Architecture
Architecting in Cloud Environment

What Does "Architecting in Cloud" Mean?

It means designing software applications to run efficiently, securely, and reliably in a cloud-based infrastructure (like AWS, Azure, GCP).

Cloud architecture is different from traditional architecture because it must support scalability, flexibility, and on-demand services.

Why It's Important.?

- ❖ Cloud systems must handle dynamic workloads
- ❖ Must be cost-effective, scale-able, and highly available
- ❖ Supports global users, fail-over, and auto-scaling

Key Principles of Cloud Architecture

Scalability

– Automatically add/remove resources based on demand

Elasticity

– Grow or shrink resources dynamically (pay-as-you-go)

Fault Tolerance

– Continue working even if part of the system fails (failover setup)

High Availability

– Application stays online 24/7 using backups and replication

Security

– Data encryption, access control, and safe APIs

Loose Coupling

– Components should work independently (micro-services)

Automation

– Use scripts, templates, and CI/CD tools for deployment and scaling

PREPARED BY ALMAS AFZAL

Topic_218
Software Design and Architecture
Service-Oriented Architecture____Introduction

What is Service-Oriented Architecture (SOA)?

SOA is a software design approach where an application is built by combining loosely coupled services.

Each service is a small, independent program that does a specific task, and these services communicate with each other over a network (usually via HTTP or messaging systems).

Key Idea

Break down the software into independent services that can be reused, replaced, or updated easily.

Examples

An e-commerce website may use SOA like this:

User Service → handles login and registration
Order Service → handles order placement
Payment Service → handles billing
Notification Service → sends email/SMS

All these services talk to each other but work independently.

Main Characteristics of SOA

Loose Coupling

– Services are independent; changing one doesn't break others

Re-usability

– Services can be reused in different applications

Discover-ability

– Services are registered and can be located by other systems

Standardized Interfaces

– Services communicate using standard protocols like HTTP, SOAP, or REST

Interoperability

– Services can be written in different programming languages and still work together

Benefits of SOA

- ❖ Easier to maintain and update parts of the system
- ❖ Supports integration across platforms (Java, .NET, PHP, etc.)
- ❖ Enables scalability and re-usability
- ❖ Good for enterprise-level applications with multiple teams

PREPARED BY ALMAS AFZAL

What Makes Service-Oriented Architecture Different from other Distributed Solutions.?

What is a Distributed System?

A distributed system is a system where components are located on different networked computers, but work together to perform a task.
Examples: client-server apps, RPC systems, CORBA, RMI, web services.

What is SOA?

Service-Oriented Architecture (SOA) is a special type of distributed system where the system is built using independent, reusable services that communicate over a network.

Feature	Traditional Distributed Systems	Service-Oriented Architecture (SOA)
Coupling	Tightly connected components	Loosely connected services
Technology	Works in same language/tech	Supports different languages/technologies
Re-usability	Hard to reuse parts	Services are reusable
Communication	Uses custom protocols (e.g., RMI)	Uses standard web protocols (e.g., HTTP, SOAP)
Flexibility	Difficult to change	Easy to modify or replace services
Integration	Limited	Easy to integrate across platforms

PREPARED BY ALMAS KHAN

Topic_220 & 221
Software Design and Architecture
Benefits & Challenges Service-Oriented Architecture

Benefits:

- Re-usability
- Scalability
- Flexibility
- Interoperability
- Faster Development
- Easy Maintenance
- Better Integration
- Cost Effectiveness

- Improved Security
- Cloud Compatibility

Challenges:

- Complexity in Design
- High Initial Cost
- Security Management
- Performance Overhead
- Service Versioning Issues
- Difficult Testing
- Network Dependency
- Governance and Standardization
- Inter-Service Communication Failures
- Skilled Resource Requirement

Topic_222
Software Design and Architecture
Micro-Service Architecture_____Introduction

What is Micro-service Architecture?

Micro-service Architecture is a software design style where an application is built as a collection of small, independent services, each responsible for a specific feature or task.

Each micro-service runs independently, communicates through APIs (like HTTP/REST), and can be developed, deployed, and updated separately.

Key Idea:

Break a big application into many small parts (micro-services) that can work on their own and together.

Technologies Used

- APIs (REST/GraphQL) for communication
- Docker, Kubernetes for containerization and orchestration
- CI/CD pipelines for automation
- Service Discovery, Load Balancers, Monitoring tools

Topic_223
Software Design and Architecture
Characteristics of Micro-Service Architecture

PREPARED BY ALMAS AFZAL

Characteristics of Micro service Architecture

Single Responsibility
– Each service focuses on one specific task or business function.

Independent Deployment
– Services can be updated, deployed, or restarted without affecting others.

Decentralized Data Management
– Each micro-service manages its own database (no shared DB).

Technology Diversity
– Different services can use different languages or frameworks.

Scalability
– Individual services can be scaled separately based on their load.

Fault Isolation
– If one service fails, it doesn't crash the whole system.

Topic_224
Software Design and Architecture
Designing Polyglot Services

PREPARED BY ANAS AFZAL

What are Polyglot Services?
Polyglot Services means building different services of a system using different programming languages, tools, or technologies — each chosen based on the specific needs of that service.

Why Use Polyglot Services?
Some languages are better suited for specific tasks.
Example:

1. Use Python for AI/ML
2. Use Java for enterprise apps
3. Use Node.js for real-time apps
4. Gives teams freedom to choose the best tool for their service.

Where is it Used?
Polyglot services are common in Micro-service Architecture, where each service is independent and can be written in any language.

Benefits of Polyglot Services
Right tool for the right job
Technology flexibility

Easier to adopt modern tools
Improved performance (for specific tasks)
Better team productivity (teams use familiar languages)

Challenges of Polyglot Services
Complex deployment and testing
Harder to maintain and integrate
Need for multi-skill teams
More effort in monitoring and logging
Version and compatibility issues

Topic_225
Software Design and Architecture
Stateless VS Statefull Micro-Services

PREPARED BY ALMAS AFZAL

1. Stateless Microservices

- Do not remember anything from previous requests.
- Each request is independent and self-contained.
- No data is stored in the service between calls.

Example:
A weather API that returns temperature when you give a city name – no need to remember previous calls.

Characteristics:

- Easier to scale
- Better for cloud and load balancing
- Simple to deploy and manage

2. Stateful Micro-services

Remember information between requests.
Store session data or user context inside the service.

Example:
A shopping cart service that stores items until the user checks out.

Characteristics:

- Harder to scale (need to manage user sessions)
- May require sticky sessions or shared storage
- Needs more memory and care during failure or restart

Topic_226
Software Design and Architecture
Micro-Services Architecture (Challenges)

Challenges:

- System Design Complexity
- Deployment Complexity
- Data Management Issues
- Inter-Service Communication Problems
- Difficult Debugging and Monitoring
- Complex Testing
- Dependency and Deployment Order Issues
- Security Management
- Need for Skilled Teams
- High Operational Overhead

Topic_227
Software Design and Architecture
Summing it up…..
What every Software Architecture Should Know?

What Every Software Architect Should Know.?

1. Understand Business and Technical Requirements
2. Focus on Quality Attributes (Performance, Security, etc.)
3. Use Modular and Layered Designs
4. Prefer Loose Coupling and High Cohesion
5. Choose the Right Architecture Style (e.g., Microservices, Layered)
6. Apply Suitable Design Patterns
7. Embrace Automation and DevOps (CI/CD)
8. Document the Architecture Clearly
9. Plan for Scalability and Change
10. Always Balance Trade-offs (Cost, Speed, Simplicity, etc.)

COMPLETED

PREPARED BY ALMAS AFZAL

BEST OF LUCK!

PREPARED BY ALMAS AFZAL