# Design 1

Architectural Patterns


# Introduction to Design

# Definition

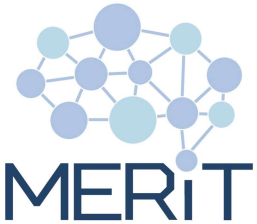
- The design is the process of **translating the requirements** into a **design** that will **guide the implementation**.
- It serves as the **bridge between requirements and code**.

# Purpose

- Shape the **development strategy**.
- Building the **architecture**.
- Convert the requirements into **technical blueprints**.
- Support **maintainability** and **scalability**.
- Minimize **implementation risks**.

# Key Activities

- Select Architecture Type
- Design High-Level Component
- Select Technology Stack
- Design Interfaces
- Address Non-Functional Aspects
- Apply Design Patterns
- Design Logic and Data Structures

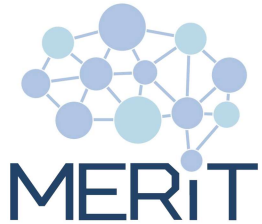


# Key Deliverables

- Architectural Design Document
- Technology Stack Document
- Design Constraints and Assumptions Log
- Data Design Artifacts
- Component and Module Specifications
- Interface Design Specifications
- Non-Functional Design Specifications

# Key Deliverables

- Design Pattern Application Records
- Low-Level Design Documents



# Key Roles

- Software Architect: Owns the overall system architecture and structure.
- Tech Lead: Works closely with architects and developers to maintain design consistency.
- Developer / Software Engineer: Create low-level designs.
- Database Engineer: Designs data models, storage schemas, and access strategies.

# Key Roles

- Integration Engineer: Specifies system interfaces and integration logic.
- DevOps Engineer: Advises on infrastructure-related design aspects.
- Test Lead: Reviews the design for testability and coverage.
- QA Engineer: Begins mapping test scenarios to design elements.

# Key Roles

- UI/UX Designer: Defines user interface structure, navigation, and layout.
- Business Analyst: Assists in ensuring functional coverage in the design.
- Product Owner: Approves trade-offs, deferrals, or technical constraints impacting features


# Abstraction Levels

# Definition

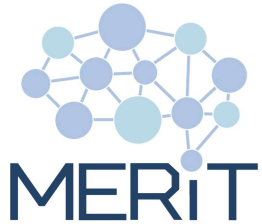
- Abstraction levels represent the **layers of detail** in design.
- Each level of abstraction **builds on the previous one**, refining the system with **greater detail** and guiding the **transition from analysis to implementation**.

# Abstraction Levels

- Architectural Design
- Logical Design
- Detailed Design

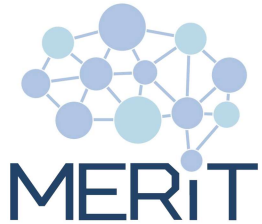

# Architectural Design

# Definition




- Architectural design is the **high-level design** of a system that defines 1) the **components** of the system, 2) the **responsibilities** of the components, and 3) the **interactions** between the components and the external environment.

# Steps



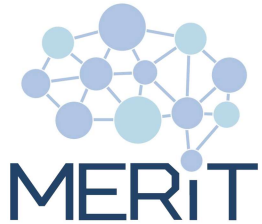

- Review the **analysis**.
- Select the **architecture type**.
- Design **components** and **their responsibilities**.
- Design **interactions**.
- Select **technology stack**.
- Define **deployment strategy**.


# Logical Design

# Definition

- Logical design is the **mid-level design** of a system that defines 1) the decomposition of the components into **modules**, 2) the **responsibilities** of the modules, and 3) the **interactions** between modules.

# Steps




- Refine components into **modules**.
- Allocate **features and requirements** to modules.
- Design internal and external **interfaces**.
- Design **data flow**.
- Address **non-functional responsibilities**.

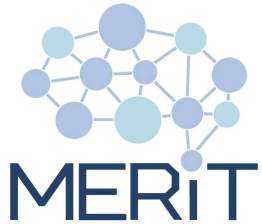

# Detailed Design

# Definition

- Detailed design is the **low-level design** of a system that defines **implementation-ready specifications**.

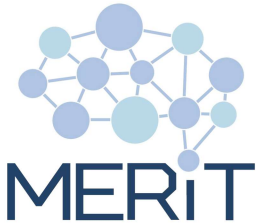
# Steps

- Select **design patterns**.
- Refine modules into **units**.
- Design **logic**.
- Design **data structures**.
- Model **state transitions** and **interactions**.
- Plan **error handling**.




# Architectural Patterns

# Architectural Patterns



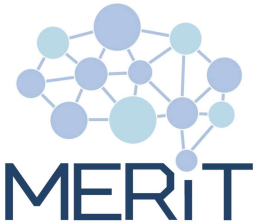

- Monolithic Architecture
- Layered Architecture
- Client-Server Architecture
- Service-Oriented Architecture (SOA)
- Event-Driven Architecture (EDA)
- Microservices Architecture


# Monolithic Architecture

# Definition

- Monolithic architecture is a **single-tier** architecture where all components are **tightly coupled** and act as a **single unified unit**.
- Era: 1960s–1970s
- Analogy: A unicellular organism.

# Emergence

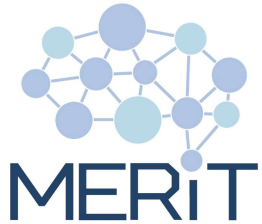



- Monolith architecture emerged out of necessity because **computers** at that time were **slow and costly**, and software had to be as centralized and compact as possible to run effectively.

# Advantages

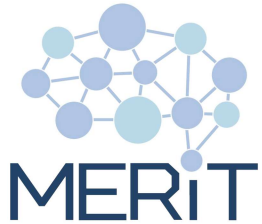
- Simple to develop for small teams
- Easy to test in early stages
- Efficient for small-scale applications

# Disadvantages




- Hard to scale
- A small bug can cause the entire application to crash
- Difficult to adopt new technologies

# Common Use Cases



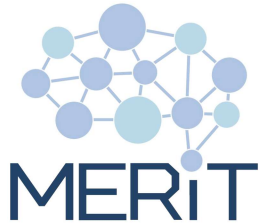

- Legacy systems
- Early-stage MVPs and prototypes
- Small to medium web or desktop applications
- Internal business tools


# Layered Architecture

# Definition

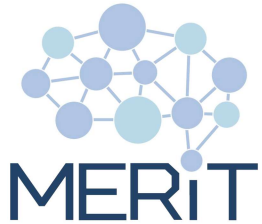
- Layered architecture is a **n-tier** architecture where the application is organized into **logical layers** with **distinct** responsibilities.
- Layers are **stacked** such that each layer only **interacts with its adjacent layers** via well-defined interfaces or abstractions.
- Era: 1970s–1980s

# Emergence



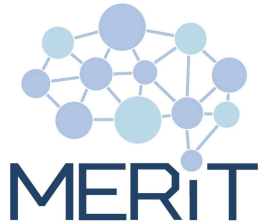

- Layered architecture gained popularity as **software complexity increased**, encouraging modular design to enhance code organization, testability, and separation of concerns.

# Advantages



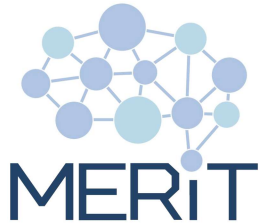

- Clear separation of concerns
- Easier to maintain and test individual layers
- Reusability across projects

# Disadvantages




- May lead to performance overhead
- Too rigid if communication is strictly one-directional

# Common Use Cases




- Traditional enterprise software (e.g., ERP, HRM)
- Banking systems with structured business rules
- Web applications using MVC frameworks (e.g., Django, ASP.NET, Spring)
- University or government systems built for maintainability


# Client-Server Architecture

# Definition

- Client-server architecture is a **distributed system** where clients issue service requests to a **centralized server** over a network protocol (typically HTTP or TCP/IP), and the server responds with the required **resources or computations**.
- Era: 1980s–1990s

# Emergence

- Client-server architecture emerged with the widespread **adoption of personal computers** and local area networks (LANs), enabling user interface (UI) clients to interact with centralized servers over networks.

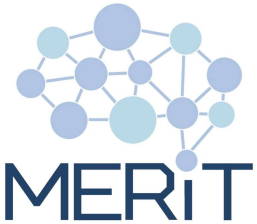
# Advantages

- Centralized control and data storage
- Multiple clients can use the same server
- Better resource management on the server side

# Disadvantages

- Server is a single point of failure
- Not ideal for very large-scale systems

# Common Use Cases



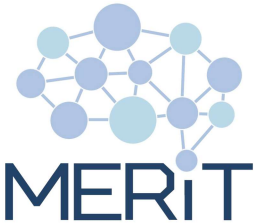

- Web applications with browser frontends and backend APIs
- Mobile apps communicating with RESTful servers
- Multiplayer games (client app - game server)
- Email systems (SMTP, IMAP, POP3)
- Remote desktop or file-sharing applications


# Service-Oriented Architecture (SOA)

# Definition

- Service-oriented architecture is a paradigm in which services **(self-contained and reusable units of business functionality) expose their interfaces using network-accessible protocols** (such as SOAP and WSDL) and are orchestrated via middleware (like the ESB) to achieve system-level workflows.
- Era: 1990s–2000s

# Emergence

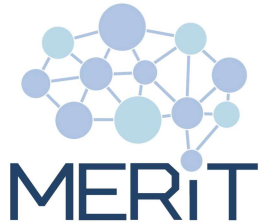



- Service-oriented architecture emerged to help large enterprises **integrate heterogeneous systems** using **reusable**, network-accessible services with formal communication protocols.

# Advantages

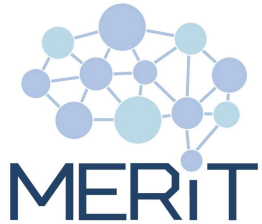
- Platform and language agnostic
- Reusability across business domains
- Scalable in large enterprise environments

# Disadvantages




- Complex service orchestration
- Performance overhead from XML/SOAP
- Governance and versioning are hard

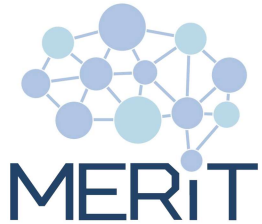
# Common Use Cases




- Large enterprise systems requiring integration across departments
- Government or telecom systems that need to expose shared services
- Interoperable B2B platforms with third-party access
- Complex workflows using orchestration engines


# Event-Driven Architecture (EDA)

# Definition




- Event-driven architecture is a reactive system in which **event producers** publish events to an event bus, and **event consumers** subscribe to react to those events, often **asynchronously** and **decoupled** from the event source.
- Era: 2000s–2010s

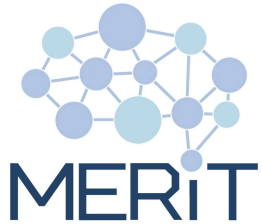
# Emergence

- Event-driven architecture emerged as systems needed to be more **reactive** and **scalable**. It became essential to decouple components via asynchronous messages to effectively manage real-time data and fluctuating workloads.

# Advantages

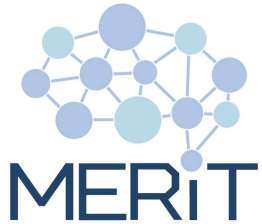
- Highly decoupled and scalable
- Responsive to real-time changes
- Supports asynchronous flows

# Disadvantages




- Hard to debug due to loose coupling
- Event storms and eventual consistency issues

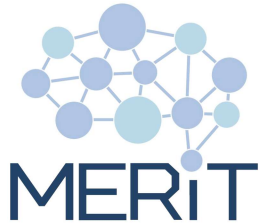
# Common Use Cases




- Real-time analytics platforms
- IoT systems
- Order processing in e-commerce
- Monitoring and alerting systems
- Gaming backends where state changes trigger actions

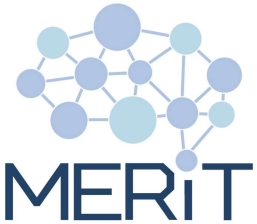

# Microservices Architecture

# Definition




- Microservices architecture structures an application as a **collection of independently deployable services**, each responsible for a **single bounded context**.
- Services communicate using **lightweight protocols** (like HTTP/REST or gRPC) and are **deployed in isolated** containers or virtual machines.
- Era: 2010s–2020s

# Emergence

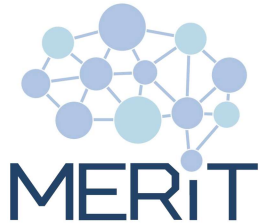



- Microservices architecture emerged as organizations aimed to **accelerate development cycles**, **scale components separately**, and empower independent teams. This approach also leverages cloud infrastructure, containerization, and DevOps practices to build systems that are more flexible, resilient, and responsive to change.

# Advantages

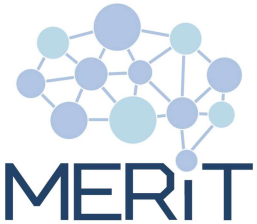
- Scalability and fault isolation
- Technology independence
- Team autonomy

# Disadvantages




- Complex inter-service communication
- Distributed system challenges such as monitoring and deployment
- Requires a strong DevOps culture

# Common Use Cases




- Large-scale SaaS platforms (e.g., Spotify, Netflix, Amazon)
- Financial platforms
- Cloud-native applications on Kubernetes or serverless platforms
- E-commerce platforms
- API-first ecosystems and multi-tenant applications