



# COSI - A Framework for Understanding Software Architecture

## What is Software Architecture?

- Software Architecture is a high-level blueprint of a software system.

## Differentiation from Software Design

- Software Design
  - Focus on low-level structure and components → code
  - Concerned with "how" the system performs a specific set of tasks. How is your code organized? How are things coupled?
  - For example: Choice of how functions and classes are organized, database schema design, changing the design of your code to remove duplication.
  - Taking decisions about software design are therefore more tactical and more implementation-focused. And because they're more on the micro-level, they're generally not that expensive to change. For example, deciding to use a Strategy pattern to improve the design of code that has some coupling probably only requires minor refactoring work. It's not free, but it's not that expensive.
  - Software design evolves frequently - it's fluent. Code is always changing, libraries you use will be updated, you'll reorganize things as you write your software and add new features or improve existing ones.
  - Software design is mostly done by the developers, testers, engineering manager. Normally, customers are not involved in software design.
- Software Architecture
  - High-level structural framework. This can be for a single system or project. Or it can be business-wide and associated with multiple systems and linked to a long-term strategy. Disclaimer: in this course, I'll focus on single systems to keep things manageable.
  - Concerned with "what" components are part of the system and how they interact.

- Decision to use microservices or a monolith, choice of cloud vs. on-premise, whether you'll use a SQL or NoSQL database and whether you're hosting that yourself or you're using a Database-as-a-Service.
- Take decisions is more strategic and on a higher level as well. And because of that, these decisions are often more expensive to change. For example, if you want to change from a SQL to a NoSQL database. That's a lot of work: rewrite all the backend code, migrate data to a new database without breaking things for your customers, business logic might change due to differences between NoSQL/SQL, etc.
- Architecture is generally quite stable - not frequently changed.
- Software architecture: Broad range of stakeholders, including business analysts, product managers, and even customers to some extent.

Takeaway: Architecture focuses on **macro-level**, design focuses on **micro-level**. Typically, design focuses on the **code**, architecture focuses on the **system**.

## Introduction to COSI Framework

- What does COSI stand for?
- Communication, Organization, Storage, Implementation
- Cross-cutting concerns: security, privacy, development and deployment processes.

## Communication

- How do modules/services communicate?
- Effects on scalability, performance, security, etc.

## Examples

- Directly in the code (calling a function or method and passing data as an argument)
- HTTP single messages - RESTful (JSON or XML data)
- GraphQL
- WebSockets (setup connection first, then full duplex)
- gRPC (layer on top of HTTP/2).
  - define a schema that specifies object types and the procedures that are exposed
  - faster than REST
  - not built for the web, might be complicated to setup

# Organization

- How are modules/services organized?
- Architectural patterns
- Impacts maintainability, extensibility.

## Examples

- Monolith
- Services
  - Microservices
  - SOA
- Hexagonal architecture (port & adapters)
  - Onion architecture
  - Clean architecture
- Event-driven architecture
  - Pub-sub
  - Scheduling
- Pipeline architecture
- Multitier/multilayer architecture
  - Layers and tiers are used interchangeably, but they are different: layer is a logical structuring mechanism vs tier is a physical structuring mechanism
  - 4 layers
    - Presentation layer (a.k.a. UI layer, view layer, presentation tier in multitier architecture)
    - Application layer (a.k.a. service layer or GRASP Controller Layer)
    - Business layer (a.k.a. business logic layer (BLL), domain logic layer)
    - Data access layer (a.k.a. persistence layer, logging, networking, and other services which are required to support a particular business layer)
  - 3 layers
    - presentation, business logic, data
    - MVC, MVP, MVVM
- Client-server
  - The Client-Server architecture involves at least two parties: a client that requests resources and a server that provides those resources.
  - The server is usually a centralized system that stores resources, databases, and services that are accessed by multiple clients.
  - Multiple clients can connect to a single server, and servers can also be clustered or load-balanced to handle more clients.
- Peer-to-peer
  - Client-Server has a centralized server, while P2P is decentralized. In Peer-to-Peer (P2P) architecture, all nodes (or "peers") are equal, meaning any node can serve as both a client and a server.

- Unlike Client-Server, there's no centralized server that stores all the data or provides all the resources. Resources are distributed across the network.
- P2P networks can be very scalable because adding new nodes increases the overall capacity of the network.
- P2P networks can be more fault-tolerant and have more redundancy. If one peer goes down, the resource can usually be found on another peer.
- All nodes can contribute resources, but this also means that all nodes need to have the capability to serve resources, which can be resource-intensive.
- The lack of centralized control can make them more susceptible to certain types of attacks or unauthorized data sharing. Client-Server architectures, being centralized, are often easier to secure and control than P2P systems.
- Blackboard architecture
  - The Blackboard architecture centers around a common knowledge base, known as the "blackboard." Different subsystems or "agents" interact with this central storage to solve a problem collaboratively.
  - Multiple specialized agents read from and write to the blackboard. These agents work asynchronously and are loosely-coupled, focusing on solving specific aspects of the overall problem.
  - The architecture is data-centric, meaning the agents respond to changes in the data on the blackboard rather than explicit commands.
  - This architecture is especially useful for complex, large-scale problems where a monolithic approach isn't effective. Different agents can have different areas of expertise, contributing to a more efficient problem-solving process.
  - The architecture allows for incremental and iterative problem-solving, as agents can contribute partial solutions that get refined over time.
- Broker architecture (communication via central broker)
  - The Broker architecture serves as a mediator between different system components, facilitating indirect communication. Unlike the centralized knowledge base in the Blackboard architecture, the Broker deals more with coordinating messages and requests among components.
  - Broker is generally used to distribute computational tasks and serve multiple clients in distributed systems. On the other hand, Blackboard is used for collaborative problem-solving, where multiple agents contribute to a single, often complex, task.
  - In Broker architecture, the components are usually more loosely coupled than in Blackboard. In Blackboard, agents need to know the structure of the common knowledge base to interact with it effectively.

# Storage

- How is information stored and persisted?
- Data integrity, speed, security.

## Examples

- SQL database
- NoSQL database
- Vector database
- Knowledge graph
- Key-value store (often used for caching)
- Object store
- Memory/RAM
- File

# Implementation

- How is the system implemented?
- Effects on cost, scalability, flexibility, deployment.

## Examples

- Serverless (scalable cloud functions, no infrastructure at all)
- Kubernetes (scalable, still infrastructure knowledge needed but on a more abstract level)
- Dedicated server(s) that each run a specific part of the system (need full knowledge of infrastructure)
- Depending on what you want, you can run each of these things yourself on-premise, in a datacenter offered by any of the cloud providers, or you can use a PaaS solution (often good idea for database or cache)

# How to Use the COSI Framework

- Analyzing Existing Systems: Evaluate based on the four dimensions.
- Planning New Systems: Define objectives in each COSI area.
- You don't have to pick one for each dimension of COSI!